

## JAVA SYLLABUS

### Basic Concepts :-

what is java?

what is programming language?

Levels of languages?

java independent

Features of java

JDK architecture

command prompt:

Structure of class

print & println statement

### Part - 1

#### TOKENS

↳ keywords

↳ Identifiers

↳ Separators

Data types

Variables

operators

Decision Making statement

Loops

methods

### Part - 2

static

JVM memory allocation

object

non-static

Object loading process

this - keyword

this() - statement

OOPS:

### Encapsulation

- ↳ private, getter, setter

### Relationship

- ↳ has-a relationship

- ↳ is a relationship

- ↳ super() statement

- ↳ upcasting + downcasting

### Polymorphism

- ↳ method overloading

- ↳ constructor overloading

- ↳ shadowing

- ↳ method overriding

### Final

### Abstraction:

concrete class

Interface

part 3:

packages

modifiers

protected

dynamic reader (scanner class)

### Arrays

### Linear Search

Bubble sort

Object class

String class

Exception handling

wrapper class

collection framework

8/3/22

## INTRODUCTION OF JAVA:

\* Java was developed by James Gosling from Sun microsystem in the year of 1991. Java is open source software.

\* Now Java was owned by Oracle

\* Java first version is released in the year of 1995. which is Java 1.0. we are having Java version till Java 15

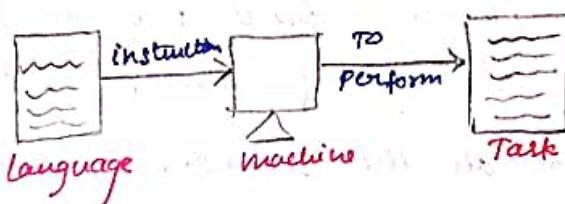
\* The most stable version is above Java 8.

what is Java?

Java is a programming language

Programming language:

A programming language is language is used to instruct a machine to perform some specific task



Types of language:

The language are classified into three types based on machine understanding

1) Binary language / machine language / low level language

2) Assembly language / middle level language

3) High level language / programmer friendly language

## Machine level language:

\* machine level language is language which is only understandable by a machine. Hence, it is known as machine level language.

\* machine can understand 0's and 1's known as binary digits.

\* Language which is making use of binary digit known as binary language.

## Binary language:

0 → low (logical low)

1 → high (logical high)

**NOTE:** humans can not understand machine level language therefore it is very difficult for a programmer to write instruction in binary language. hence programmer will never use machine level language for developing SW application

\* we can convert all the character such as:

→ alphabets [ Uppercase : A, B, ..., Z ]  
[ Lowercase : a, b, ..., z ]

→ Decimal Number System (0, 1, ..., 9)

→ Special character (@, %, \$, #, !, ...)

\* Special character can be converted to machine understandable language with the help of ASCII code

American Standard Code for Information Interchange

A = 65 to Z = 90

a = 97 to z = 122

0 = 48 to 9 = 57

eg:

$$B = 66$$

$$\begin{array}{r} 2 | 66 \\ 2 | 33 - 0 \\ 2 | 16 - 1 \\ 2 | 8 - 0 \\ 2 | 4 - 0 \\ 2 | 2 - 0 \\ 1 - 0 \end{array}$$

$1000010$  is the binary equivalent representation of B.

### Assembly level language:

\* Assembly level language is a language which is easily understandable by microprocessor and microcontroller (i.e.)

\* In assembly level language there are some predefined words such as add, sub, mul, div, mov, load, jump known as mnemonics which is converted into low level language with the help of a software called Assembled machine level language

### Disadvantage :

- Decision making
- looping

### High Level language:

\* The language which is programmer friendly is known as high level language

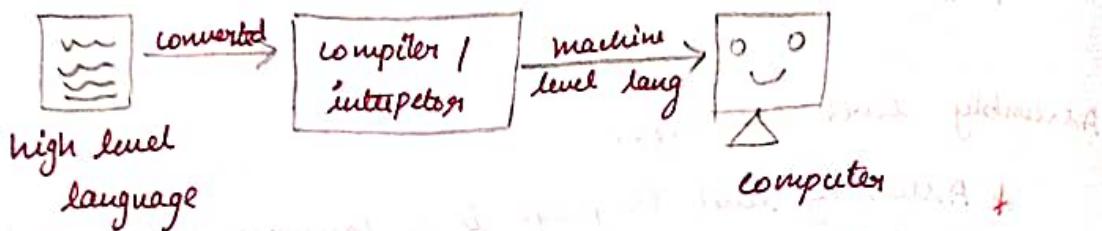
\* It was very easy to understand, learn and write the instruction

\* High level language are generally used to develop software application

eg: Java, C++, C#, Python, perl, ruby, & ...

### NOTE:

- \* Machine can not understand high level language we will a software called compiler / interpreter to convert from high level language to machine understandable language
- \* every high level language will have its own compiler



### PLATFORM:

A platform is a hardware or Software which is used to run an application.



### Operating System:

It is a system software that manages both hardware and software resources and provides common service for computer programs.

(or)

It manages how application access hardware and software resource.

32 bit

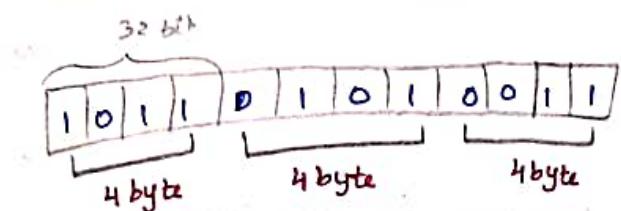
64 bit

word size, memory concept

4GB RAM, 64 bit processor

## word size:

How much bit of data a processor read at once is known as word size.



$$1\text{-byte} = 8\text{-bit}$$

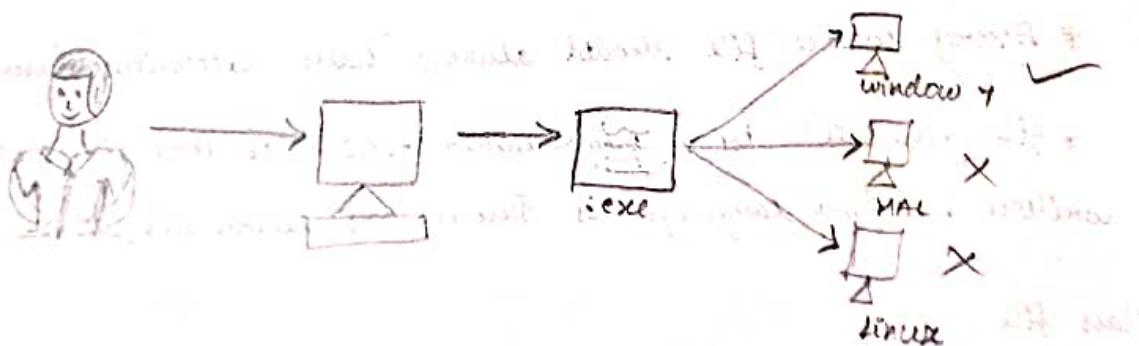
$$4\text{byte} = 4 \times 8 = 32\text{ bit}$$

platform is divided into two types

- \* Platform dependent
- \* Platform independent

### Platform dependent :

If a software developed in one platform can be used only on the same platform not on another platform is known as platform dependent.

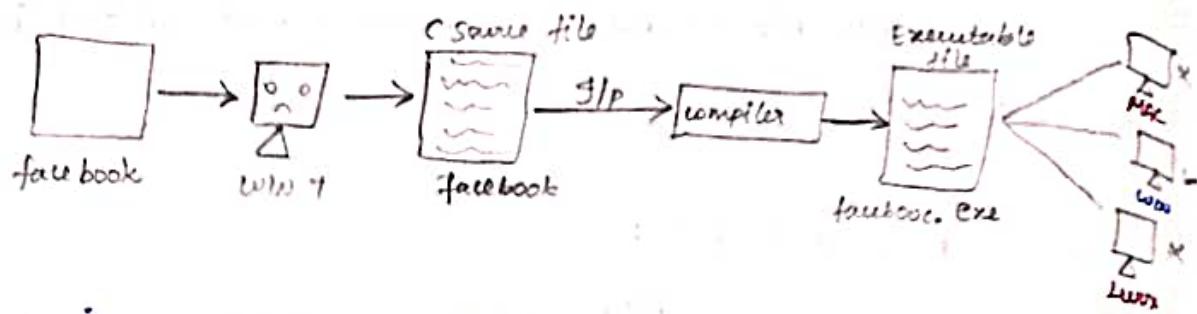


If a software developed on one platform and can be executed on any platform is known as platform independent.

### NOTE:

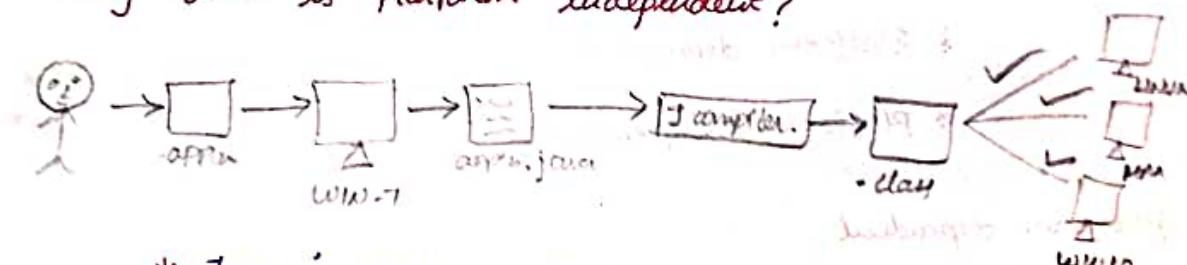
The software developed using Java is platform independent

C, C++ are platform dependent:



C is a one step compilation language i.e. the compiler directly convert the source file into native machine understandable language (.exe file). This makes C a platform dependent language.

why Java is platform independent?



- \* Java is platform independent but it is dependent on JRE
- \* Two step compilation.

Source file:

- \* Every source file should always have extension .java
- \* file generated by a programmer which consists instruction written in java language is known as source file

class file:

- \* A file which is generated by java compiler is known as class file
- \* class file contains instruction written in bytecode
- \* The extension of class file is .class

### ByteCode:

- \* It is an intermediate language created by java compiler. It is neither understandable by a programmer nor by a machine.
- \* It is only understandable by JRE (Java runtime environment)

How platform independent is achieved in Java?

- \* Java compiler does not translate java instructions directly into machine understandable language instead it generates an intermediate language known as byte code.
- \* Byte code generated is neither understandable by a programmer nor by a machine it is understandable only in a machine in which has JRE on it (JRE installed) we have JRE available for different platform.

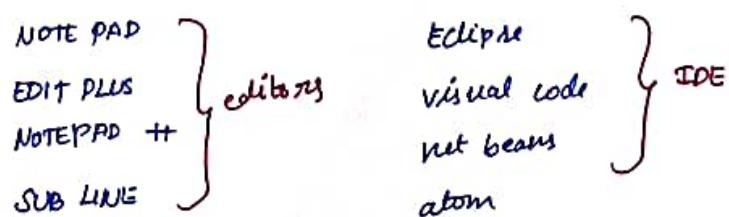
\* This design makes an application developed using Java is a platform independent but JRE dependent.

Step to create and execute Java source file.

Step 1: Create source file.

Java source file is nothing but a text file.  
File name can be anything like .java  
name.java

To create source file we need editors



\* To create java source file we can use a software called editors or an IDE

\* The source file must have name as well as extension (ie: Name.java)

Step 2: Generates class file

Requirement :

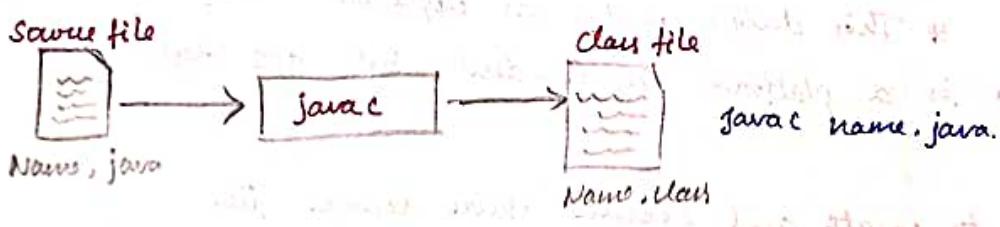
→ Source file must be ready

→ Java compiler must be installed

\* Java source file will be given as an input to the java compiler

\* Java compiler will check the syntax of instructions if there is no syntax error java instruction is translated into bytecode and class file will be generated.

\* command to call java compiler is javac



NOTE:

\* If there is any syntax error then translation will not be done and class file will not be generated.

\* The extension of class file is always .class

### Step 3: Execute classfile

Requirement:

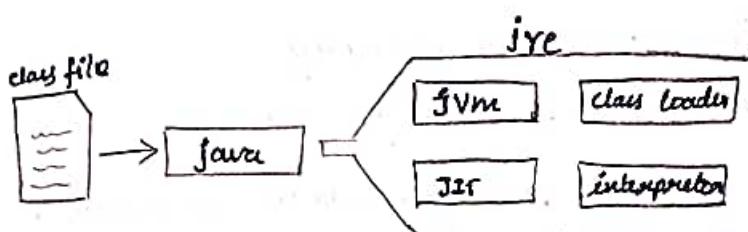
→ class file should be generated

→ JRE must be installed

\* class file should be given as an input to JRE

\* we have command called java classname which is used to create the instance of java runtime environment

\* java class-Name / file name



Source file will be stored in secondary memory

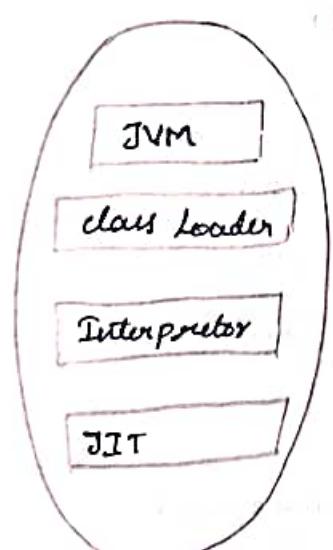
\* jre will be stored in primary memory

\* Entire memory will not be allocated for jre (only a portion of memory will be allocated)

without source file can we execute class file.

Yes, classfile will be generated from source file

### JRE (Java Runtime Environment)



+

### Build in Libraries

↳ java.lang

↳ java.util

↳ java.io

components available  
in jre

jre consists of components such as jvm, class loader, interpreter, JIT etc., as well as built in packages, built-in-libraries such as `java.lang`, `java.util`, `java.io` etc., which are essential for interpreting & executing byte code.

### JVM (Java virtual machine):

It is the main software which drives on all other components.

class loader: jre → primary memory

class file → secondary memory

It load the class file from secondary memory to primary memory jre.

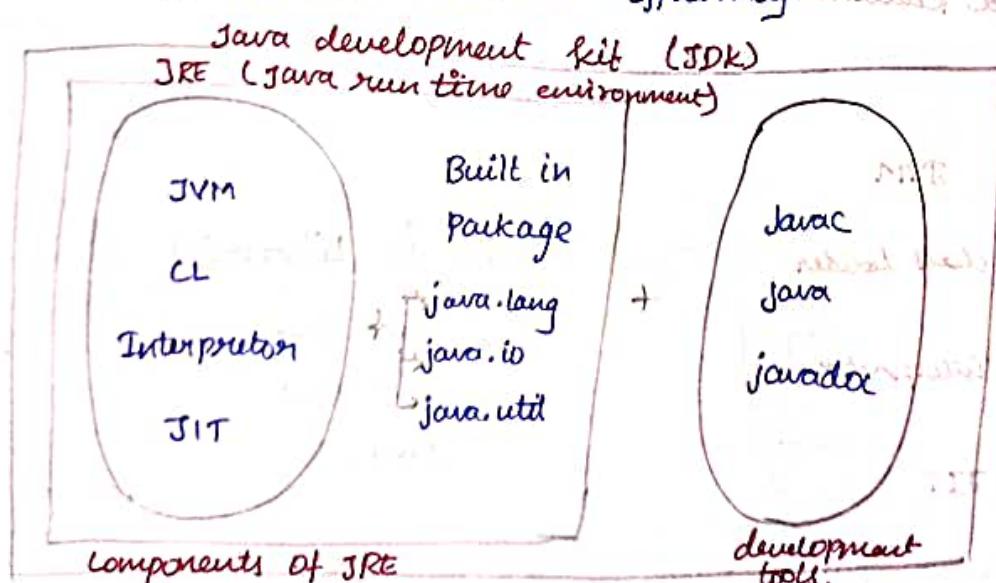
### Interpreter:

It convert bytecode instruction to native machine understandable language.

(therefore java is a compiled and interpreted language)

### JIT (just in time):

To increase run time efficiency



\* Javac is used to compile the source file

\* Java is used to Execute the class file

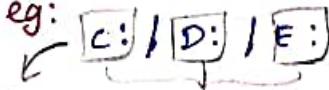
\* JDK is a package which consists of java development tools as well as JRE, which is essential for a programmer to compile and execute a java application.

③ It act like IDE to execute the program <sup>also called as output</sup> Command prompt (we can access only by Hard disk: <sup>① Consol</sup> <sup>② (It is CLS)</sup> ie command line interface commands)

\* Hard disk is divided into portion each portion is called as drive.

\* File structure always starts with drive name

\* Every drive will have a name and it is represented as drive name.

eg: 

Root folder:

\* Root folder is always represented by '\'.  
Sub folder can be created

→ C:\user

→ C:\programfiles

→ C:\users\Lavanya → path to each Lavanya folder.

Note:  
why we are partitioning the memory?

If all the folders/documents are stored in single drive. If that drive got deleted all the data present in that drive will get deleted including periodical data.

User's Home directory: → also called folder

A folder which is created in the name of user  
inside user folder it is known as user's home  
directory

e.g.: c:\user\Lavanya → is the user name

i.e. folder is created in the name of Lavanya inside  
user folder

NOTE:

Everytime we open command prompt user's home  
directory location will be loaded.  
we have only one user home directory  
working directory:

The folder which are currently in use is known  
as working directory.

NOTE:

All the commands will executed by default in  
working directory

we have more than one working directory.

c:\users\asp (asp is the folder which is currently  
in the working)

commands of command prompt:

making directory / create directory → (mkdir)  
it is used to create a folder inside working directory

Syntax:

mkdir folder-name1 and folder-name2 foldername..

change directory → cd used to change the directory

change directly → rmdir folder-name (It is removes the available folder in the working directory).

clear the screen → cls (It clears the command/screen).

absolute path:

\* If the path starts from driver name. Then, it is called as absolute path.

\* It does not depends on working folder

eg: C:\user\lavaranya  
(or)

D:\JAVA\program\session2

Relative Path:

\* If the path starts from folder name. Then, it is called as Relative path.

\* Relative path always depends on working folder

eg: JAVA\program\session2

dir:

To list the files in the folder.

∴ It is used to check the existence of file or folder.

## STRUCTURE OF JAVA PROGRAM:

Java instruction are always written inside the class

```

class className
{
    public static void main(String[] args)
    {
        // statements
    }
}

```

eg: as human having  
 - some structure or feature  
 where it having brain,  
 eye, ear, nose, skin,  
 etc.

Class or is Block  
 where it contains  
 members

**File Name:** className.java

### NOTE:

Every class in java must have a name, it is known as className

Every class has a block, it is known as class block.

In class Block we can create

- \* Variables

- \* Methods

- \* Initializers

These are said to be a members of a class

### Variables :

Variable is a container which is used to store data

### Methods

It is a block of instructions which is used to perform a task.

## Initializers:

QUESTION PAPER

Initializers are used to execute the start-up instructions.

### NOTE:

A class in java can be executed only if main method is created as follows.

Syntax to create main method:

```
public static void main(String[] args)
```

//statements.

```
}
```

### NOTE:

We can create a class without main method. It is compile time success. And class file is generated but we can't execute that class. Because, execution starts from main method and ends at main method.

QUESTION PAPER  
DIFFERENCE BETWEEN PRINT AND PRINTLN STATEMENT.

Println statements:

```
System.out.println(data)
```

\* println statement is used to print data as well as create a new line

\* We can use the println statement without passing any data, it is just used for printing new line

Eg:

```
System.out.println("Hi");
```

```
System.out.println("Laila");
```

```
System.out.println();
```

## OUTPUT SCREEN:

Hi  
Laila  
- (no data)

## Print Statement:

System.out.print(data)

- \* Print statement is used only to print the data.
- \* We can't use the print statement without passing any data, if we use then we will get compile time error (CTE)

Eg :

```
System.out.print("Hi");  
System.out.print("Laila");  
// System.out.print(); // CTE
```

## OUTPUT SCREEN:

Hi Laila -

## Executing Hello world Program

```
class Program1  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello world");  
    }  
}
```

## OUTPUT:

Hello world

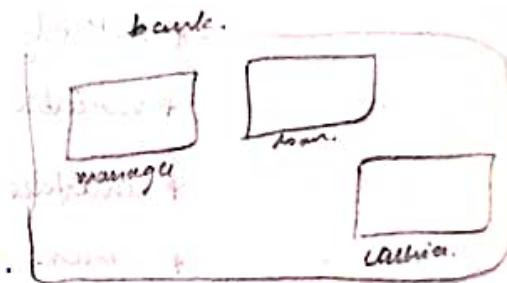
## TOKENS :

The smallest unit of programming language which is used to compose instruction is known as token  
(or)

tokens are the smallest unit of programming language. with the help of tokens we can built the program.

### Types :

- \* keywords
- \* Identifiers
- \* Literals / Data / values.



### Keywords :

\* A predefined words which the java compiler can understand is known as keyword

\* Every keyword in java is associated with a specific task

\* A programmer can't change the meaning of keyword  
(can't modify the associated task)

Eg:

we have got keywords in java

class, public, static, void etc.

### RULE :

Keywords are always written lower case.

Eg: class      write      take      give }  
                while }  
                { can't change the  
                meaning it is  
                already defined

## Identifiers:

The name given to the components of java by the programmer is known as identifiers.

Eg: to identify come on wear having name.

### List of components:

- \* class
- \* method
- \* variable
- \* interface
- \* enum
- \* constructor
- \* package etc.,

### NOTE:

A programmer should follow the rules and conventions of an identifier

### Rules of an identifier:

- \* Identifiers should never start with a number
- \* Identifier should not have special characters except \$ and \_
- \* character space is not allowed in identifier
- \* we can't use keyword as an identifier

### conventions:

The coding or industrial standard to be followed by the programmer is known as convention.

### Note:

: Standard

\* Compiler doesn't validate the convention, therefore if convention is not followed then we won't get compile time error.

\* It is highly recommended to follow the convention.

### Convention for class Name / Interface:

**Single word** - The first <sup>character</sup> word should be in upper case remaining in lower case

eg: Addition, Calculator, Sum, etc.,

**Multiword** - The first character of every word should be in upper case remaining in lower case.

eg: SquareRoot, PowerOfDigit, etc.,

### Convention for Method

**Single word** - Should be in lower case :

eg: addition, calculator, sum, etc.,

**Multiword** - First word should be in lowercase remaining words should start with uppercase.

eg: squareRoot, powerOfDigit, factorialOfDigits, etc.,

## Literals :

Literals are also called as Data or Values

### Types:

- \* Number
- \* character
- \* boolean
- \* String

The data is generally categorized into two types

- \* Primitive values
- \* Non-primitive values

### Primitive values:

- \* Single valued data is called as primitive value.  
eg: number, character, boolean

### Non-Primitive values:

- \* Multi valued data is called as non-primitive value.  
eg: String, object reference

### Primitive value:

#### Number Literals:

##### Integer number literals

eg: 1, 4, 67, 28, 37, etc.,

##### Floating number literals

eg: 1.5, 2.8, 31.25, ...

## Character Literals:

Anything which is enclosed within a single quote ('') is considered as a character literal.

The length of character literals should be one.

Eg: 'a', 'G', '1', '\$', etc.,

## Boolean Literals:

Boolean literals are used to write logical values.

We have two Boolean literals

- true
- false

## Non-primitive values:

Reference of an object/address is known as non-primitive value (group of data)

Eg: String, details of objects; etc.,

## String Literals:

Anything enclosed within a double quote ("") is known as String literals. The length of the String literal can be anything.

They are case sensitive

Eg: "Hello", "true", "a", "hello@", "", "1.1" etc.

## Session 2: Activity :

1. write a java program with a class name starting with a number
2. WAJP with a class name as a keyword
3. WAJP with a class name starting with \$
4. WAJP with a class name starting with \_
5. WAJP with a class name as multiword and the first letter of each word should be in upper case
6. WAJP to print the account no, IFSC, branch name, bank name and available balance.

15/3/22

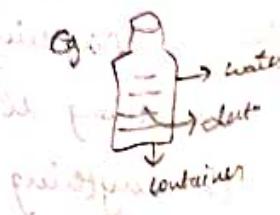
### Variables :

\* Variable is a container which is used to store a single value.

\* We have two types of variables. They are

→ Primitive variable

→ Non primitive variable



### Primitive Variable:

\* The variable which is used to store a primitive values such as numbers, characters, boolean is known as primitive variable.

\* We can create primitive variable with the help of primitive data type

Syntax to create primitive variable:

data type identifier<sub>1</sub>, identifier<sub>2</sub>, ...;

Primitive datatype identifier<sub>1</sub>, identifier<sub>2</sub>, ...;

Eg:

int a; → Primitive Variable of int type

boolean b; → Primitive Variable of boolean type.

Non Primitive Variable:

\* The variable which is used to store a reference is known as Non-primitive variable.

\* It is also known as reference Variable.

Syntax to create Non-primitive Variable:

Non primitive datatype identifier<sub>1</sub>, identifier<sub>2</sub>, ...;

Eg:

String s = new String();

Datatypes:

\* Datatypes are used to create variables of specific type.

\* In Java datatypes are classified into two types.

→ Primitive datatype

→ Non primitive datatype

eg: bottle can be  
plastic, glass, copper  
etc, based on its type.

Primitive datatype:

The datatype which is used to create a variable to store primitive value such as numbers, characters, boolean is known as primitive data type.

NOTE:

All primitive data types are keyword in Java.

PRIMITIVE VALUES	PRIMITIVE DATA TYPES	DEFAULT VALUE	SIZE
Number	Integer	byte short int long	1 byte 2 byte 4 byte 8 byte
	floating	float double	4 byte 8 byte
		char	8 byte
		boolean	1 byte
Character			
Boolean			

### NOTE:

The number data type in increasing order of the capacity  
 byte < short < int < long < float < double

byte < short < char < int < long < float < double.

### Non-primitive data type :

The data type which is used to create a non-primitive variable to store the reference is known as non-primitive data type.

### NOTE:

Every class name in java is non-primitive datatype

### Scope Of Variables :

\* The visibility of a variable is known as scope of a variable

\* Based on scope of a variable we can categorize variable in three types

→ Local variable

→ Static variable

→ Non Static variable (instance)

**Local Variable:**

The variable declared inside a method block or any other block except class block is known as local Variable

**characteristics of Local Variable:**

\* we can't use local variables without initialization, if we try to use local variable without initialization then we will get compile time error.

\* Local variables will not be initialized with default values

\* The scope of the local variable is nested inside the block whenever it is declared, hence it can't be used outside the block.

**Type Casting:**

\* The process of converting one type of data into another type is known as Type casting

\* There are two types of type casting:

Primitive type casting

widening (implicitly)

Narrowing (Explicitly)

Non-primitive type casting

Upcasting (implicit)

downcasting (Explicitly)

**Primitive type casting:**

The process of converting one primitive value into another primitive value is known as primitive type casting

widening:

\* The process of converting smaller range of primitive data type into large range of primitive data type is called widening

- \* In widening process there is no data loss
- \* Since, there is no data loss, compiler can implicitly do widening hence it is also known as auto widening

### Narrowing:

- \* The process of converting larger range of primitive data type into smaller range of primitive data type is known as Narrowing
- \* In narrowing process there is a possibility of data loss
- \* Since there is a possibility of data loss, compiler does not do narrowing implicitly
- \* It can be done explicitly by the programmer with the help of type cast operator

### Type Caste Operator:

- \* It is a unary operator
- \* Type caste operator is used to explicitly convert one datatype into another datatype.

## Operators:

\* Operators are predefined symbol which is used to perform specific task on the given data.

\* The data given as a input to the operator is known as operand.

Based on the number of operands operators are further classified into the following

\* Unary operator

\* Binary operator

\* Ternary operator

### Unary operator:

The operator which can accept only one operand is known as unary operator

### Binary operator:

The operator which can accept only two operand is known as binary operator

### Ternary operator:

The operator which can accept three operand is known as Ternary operator.

### Types of operators: (based on function)

The operators can also be classified based on the task.

\* Arithmetic operator (binary operator)

\* Assignment operator (unary operator)

\* Relational operator (binary operator)

\* Logical operator (binary operator)

\* Increment/decrement operator (unary operator)

\* conditional operator (ternary operator)

\* Miscellaneous (Type cast & instanceof) (unary operator)

\* Bitwise operator

## Arithmetic Operator:

These are the operators which can accept two operands. So, arithmetic operators are binary operators.

+	addition
-	Subtraction
*	Multiplication
/	division (Quotient)
%	modulus (Remainder)

Type 1	Type 2	RESULT
Byte	Byte	int
short	short	int
int	int	int
long	long	long
float	float	float
double	double	double
char	char	int

### NOTE:

If operation done with combination of different type  
the result must be higher data type only.

Except for byte and short.

e.g.

$$\text{int} + \text{float} = \text{float}$$

## Assignment Operator:

This operator is used to assign/give/store the value inside the variable.

e.g.:  $a=5$ ; // here 5 is stored inside a

## Compound assignment operator:

\* There are the operators which perform arithmetic operation and store the data in the existing/current container. i.e., it just update the value of the container with new value.

\* The combination of arithmetic and assignment operator is called compound assignment operator.

operator	Example	equivalent to
$+=$	$a+b$	$a=a+b$
$-=$	$a-b$	$a=a-b$
$*=$	$a*b$	$a=a*b$
$/=$	$a/b$	$a=a/b$
$\% =$	$a \% b$	$a=a \% b$

## Example 1:

```
class Assignment Operator
{
    public static void main(String [] args)
    {
        int a=20;
        a+=10;
        System.out.println(a);
    }
}
```

O/p: 30

example 2:

```
class AssignmentOperator2  
{  
    public static void main(String[] args)  
}
```

```
    int a=20;
```

```
    a+=20; // 20+20 = 40
```

```
    a-=1; // 40-1 = 39
```

```
    a*=3; // 39 * 3 = 117
```

```
    a/=2; // 117/2 = 58
```

```
    a+=17; // 58+17 = 75
```

```
    System.out.println(a);
```

```
}
```

```
}
```

O/P: 75

### Relational operator:

- \* It is binary operator

- \* The return type of relational operator is boolean i.e, it returns either true or false

operators

Example

>

a>b

<

a<b

>=

a>=b

<=

a<=b

==

a==b

!=

a!=b

### Example 1:

```
class Relational Operator
{
    public static void main (String [ ] args)
    {
        int a = 5;
        int b = 2;
        boolean c = a > b;
        System.out.println (c);
    }
}
```

Output: true

### Logical Operators:

These are the operators which can accept boolean type of data. i.e., the expression or data given to the logical operator is of boolean type.

Eg:  $\frac{5>2}{\downarrow}$   $\&$   $\frac{2>1}{\downarrow}$   
exp (return boolean)      exp (return boolean)

#### OPERATOR

$\&&$  (Logical AND)

#### OPERATION

If all the expression returns true. Then this operator will return true

#### || (Logical OR)

If any of one of the expression return true then this operator will return true

#### ! (Logical NOT)

If the result is true then it will return false & vice versa.

example 1: (for AND & & )

(cont'd)

## class Logical Operators

1

```
public static void main(String[] args)
```

{

int a = 80;

int b = 30;

int  $c = 70;$

boolean res =  $a > b \& \& b > c;$   
(f)  $80 > 30 \rightarrow 30 > 20$

System.out.println(res);

3

3

olp: false

(an) (ii)

## class Logical Operator

1

```
public static void main(String[] args)
```

1

int a=80;

int b = 30;

**int c=70;**

boolean res = a < b & b > c;

802 30

False → it won't check

System.out.println(res); and operand because it is 18

(FBI) operator

1

3

olp: false

### case (iii)

class Logical Operator

{

public static void main (String [] args)

{

int a=80;

int b=30;

int c=70;

boolean res = !(a < b && b > c);

System.out.println(res);

}

}

O/p: true.

examp 2 : ( for OR || )

case(i)

class Logical Operator

{

public static void main (String [] args)

{

int a=80;

int b=30; -> (a < b) is 2 (2nd || ok<2) = true operand

int c=70;

boolean res = a < b || b < c;

System.out.println(res);

}

}

O/p: true

Case (ii)

```
class LogicalOperator
{
    public static void main(String [] args)
    {
        int a=80;
        int b=30;
        int c=70;

        boolean res = a>b || b<c;
        System.out.println(res); → straightly print
                                     true because it is OR operator,
    }
}
```

O/P: true

Example 3: (for both && and ||)

```
class LogicalOperator
{
    public static void main(String [] args)
    {
        int a=80;
        int b=30;
        int c=70;

        boolean res = (a>b || b<c) && (a>b && b<c);
        System.out.println(res);
    }
}
```

Annotations for the code:

- Annotations for the first part of the expression:  $a > b$  is labeled "80 > 30 (T)" and "won't check".
- Annotations for the second part of the expression:  $b < c$  is labeled "30 < 70 (T)" and "will check".
- Annotations for the final result: "True" is labeled "True" and "and check".

O/P: true.

conditional operator:

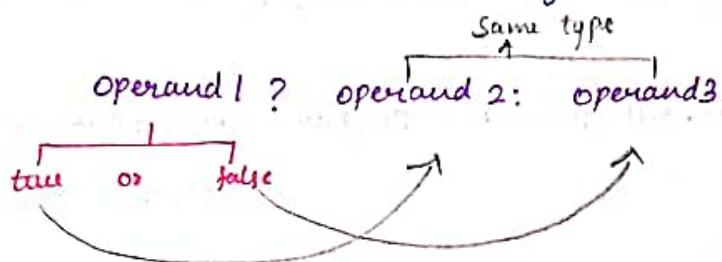
It is a ternary operator.

Syntax to create conditional operator:

operand1 ? operand2 : operand3  
condition ? statement1 : statement2

Operation:

- \* The return type of operand1 must be Boolean
- \* If the condition returns true, statement1 will get executed else statement2 will get executed.



example :

WAP to find greatest of two numbers.

class GreatestOfTwoNumbers

{

    public static void main(String[] args)

{

    int a=20;

    int b=10;

    int res = a>b?  $\frac{20}{20} > 10$  :  $\frac{10}{20}$  ;

    System.out.println(res);

}

3

O/P: 20

example 2:

WAP to find the given number is even or odd

class EvenOrOdd

{

public static void main(String[] args)

{

int given\_num = 7;

String res = (given\_num % 2 == 0) ? "Even" : "Odd";

System.out.println("The given\_num " + given\_num + " is " + res + " number");

}

}

O/P: The given\_num 7 is odd number.

example 3:

WAP to find the given char is alphabet or not

class Alphabet

{

public static void main(String[] args)

{

char ch = 'S';

String res = (ch >= 65 && ch <= 90) || (ch >= 97 && ch <= 122) ?

"Yes" : "No";

System.out.println("The given character " + ch + " is alphabet");

}

}

O/P: Yes The given character S is alphabet

WAP to find the greatest of three numbers:

```
class GreatestNumber
{
    public static void main(String[] args)
    {
        int num1 = 7;
        int num2 = 9;
        int num3 = 2;
        // first check num1 is greater than num2 if true then check num1 is
        // greater than num3 if both are true print num1 else check num2 and num3.
        int res = ((num1 > num2) && (num1 > num3)) ? num1 :
            (num2 > num3) ? num2 : num3;
        System.out.println("The given number "+res+" is greatest
                           number");
    }
}
```

O/P: The given number 9 is greatest number

WAJP to find the smallest of three numbers:

```
class SmallestOfThree
{
    public static void main(String[] args)
    {
        int num1 = 7;
        int num2 = 9;
        int num3 = 2;
        int res = num1 < num2 ? (num1 < num3 ? num1 : num3) : (num2 < num3
            ? num2 : num3);
        System.out.println("The given number "+res+" is smallest
                           number");
    }
}
```

O/P: The given number 2 is smallest number

Increment and decrement operators: (It is unary operator)

### Increment operators:

- \* It is used update the variable by 1
- \* They are two types of increment operators:
  - \* Pre-Increment
  - \* Post-Increment

#### Pre-Increment:

- \* It is denoted as  $++i$

#### Operation:

- Increase the value by 1 and update
- Substitute the updated value
- Use the substituted value

#### Post-Increment:

- \* It is denoted as  $i++$

#### Operation:

- Substitute the original value
- Increase the value by 1 and update
- Use the substituted value

### Decrement operators:

- \* It is used to decrease update the variable by decreasing 1.

- \* They are two types of decrement operator

- \* Pre-decrement

- \* Post-decrement

## Pre-decrement :

- \* It is denoted as  $--i$
- Operation:

- Decrease the value by 1 and update
- Substitute the updated value
- use the substituted value

## Post-Decrement :

- \* It is denoted as  $i--$

### Operation:

- Substitute
- Decrement the value by 1 and update
- use the substituted value.

### NOTE :

each time the value get updated while doing increment / decrement .

### Example :

```
class Increment
```

```
{
```

```
    public static void main(String[] args)
```

```
{
```

```
    int a=10;
```

```
    int b=a++;
```

```
    System.out.println(a);
```

```
    System.out.println(b);
```

```
}
```

```
}
```

O/P:  
11  
10

### Example :

```
class IncrementAndDecrement
```

```
{
```

```
public static void main(String [] args)
```

```
{
```

```
int a=7;
```

```
int b=8;
```

```
int res = a + --a + ++b - a++;
```

```
System.out.println(res); // 16
```

```
System.out.println(a); // 7
```

```
System.out.println(b); // 9
```

```
}
```

```
}
```

O/P : 16

7

9

### Example :

```
class IncrementAndDecrement
```

```
{
```

```
public static void main (String [] args)
```

```
{
```

```
int a=7;
```

```
int b=7;
```

```
a=a++; // here first + is used and the
```

```
value is stored in a then
```

```
b=++b; // 2 it update the value
```

```
System.out.println(a);
```

```
System.out.println(b);
```

```
}
```

```
}
```

O/P : 7

8

17/3/22

## Decision Statements : Indicates the flow of program

Decision statements help the programmer to skip the block of instructions from the execution if the condition is not satisfied.

### Types of Decision Statements :

\* if statement

\* if - else statement

\* if - else if statement

\* switch

### If Statement :

The instruction written inside the if block will execute if the condition returns true. else it will not execute the instruction inside the block.

#### Syntax :

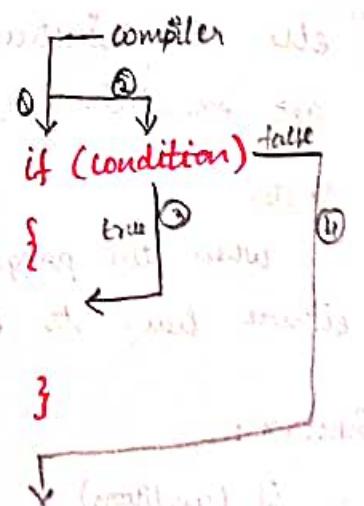
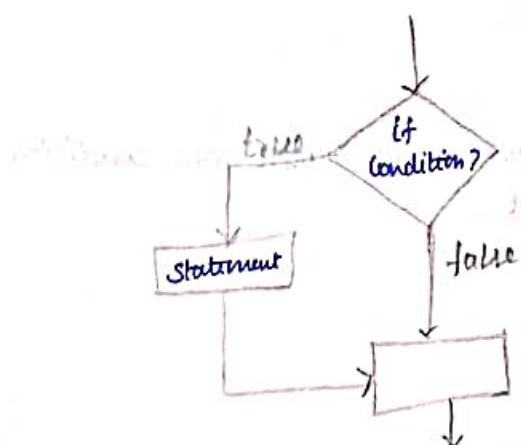
if (condition)

    {     // If condition is true, statements will be executed

        statements or a block of statements will be executed

    }     // If condition is false, statements will not be executed

#### Work Flow :



WAP to print the even number if the number is even.

```
class EvenNumber  
{  
    public static void main(String[] args)  
    {  
        int num = 7;  
        if (num % 2 == 0)  
        {  
            7 % 2 = 1 // 1 != 0 (F)  
            System.out.println(num);  
        }  
    }  
}
```

Op: nothing

if else statement:

The instruction written inside the if block will get executed only if the condition is satisfied else the instruction written inside the else block will get executed.

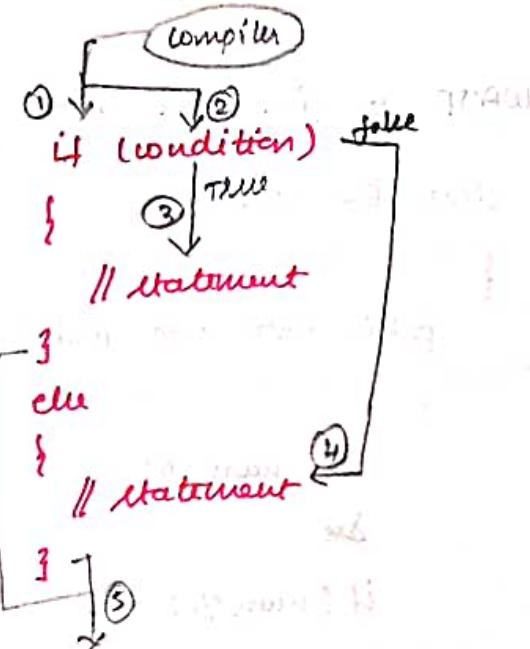
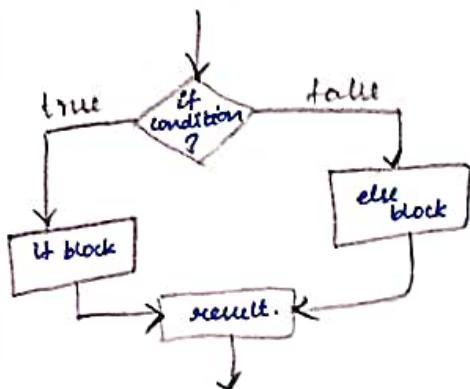
NOTE:

When the program wants the result with one condition either have to execute or not.

Syntax:

```
if (condition)  
{  
}  
else  
{  
}  
}
```

## Workflow:



WAP to find the greatest of two numbers.

```
class GreatestOfTwo
```

```
{ public static void main(String[] args)
{
    int num1 = 55;
    int num2 = 17;
    if (num1 > num2)
    {
        System.out.println("The greatest number is " + num1);
    }
    else
    {
        System.out.println("The greatest number is " + num2);
    }
}
```

Output of program: The greatest number is 55

Output of program: The greatest number is 17

WAP to find the given number is even or odd.

class EvenOrOdd

```
{  
    public static void main(String[] args)  
    {  
        int num = 6;  
        if (num % 2 == 0)  
        {  
            System.out.println("The given number " + num + " is even");  
        }  
        else  
            System.out.println("The given number " + num + " is odd");  
    }  
}
```

O/P: The given number 6 is even

WAP to find the given number is divisible of 2 & 3.

class Divisible

```
{  
    public static void main(String[] args)  
    {  
        int num = 6;  
        if (num % 2 == 0 && num % 3 == 0)  
        {  
            System.out.println("The given number is divisible of 2 & 3");  
        }  
        else  
            System.out.println("The given number is not divisible of 2 & 3");  
    }  
}
```

O/P: The given number is divisible of 2 & 3

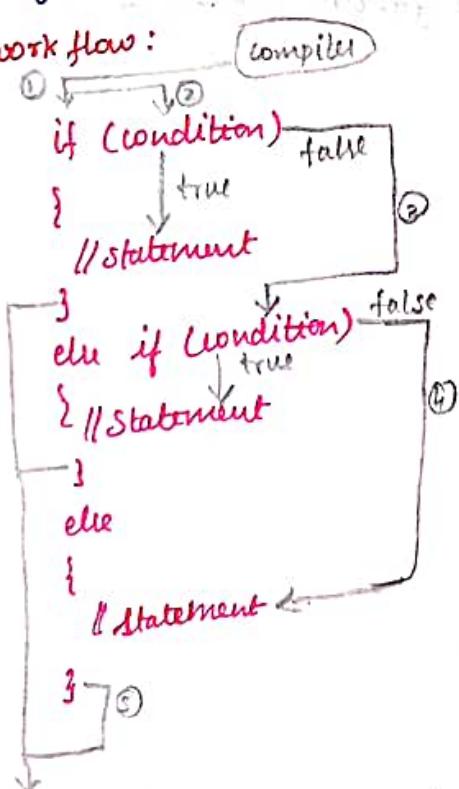
## If - else if ladder :

If the condition is satisfied then the instruction written inside the if block gets executed if not satisfied, condition is checked in the else if block from top to bottom order and if the condition is satisfied in any of the else if block then, only that else if block is gets executed. If not satisfied else block gets executed remaining blocks are skipped.

Syntax :

```
if (condition)
{
}
else if (condition)
{
}
else if (condition)
{
}
else
{
}
```

work flow :



WAP to find the greatest of Three numbers.

class GreatestOfThree

{

public static void main(String [] args)

{

int num1=366;

int num2=34;

int num3=568;

if( num1>num2 && num1>num3)

{

System.out.println("The greatest number is "+num1);

}

else if ( num2>num3)

{

System.out.println("The greatest number is "+num2);

}

else

{

System.out.println("The greatest number is "+num3);

}

}

}

O/P:

The greatest number is 568

WAP to find the smallest of 5 numbers:

class SmallestOfFive

{

    public static void main(String [] args)

{

        int n1=7;

        int n2=8;

        int n3=1;

        int n4=-3;

        int n5=0;

        if (n1 < n2 && n1 < n3 && n1 < n4 && n1 < n5)

            System.out.println("The smallest number is "+n1);

        else if (n2 < n3 && n2 < n4 && n2 < n5)

            System.out.println("The smallest number is "+n2);

        else if (n3 < n4 && n3 < n5)

            System.out.println("The smallest number is "+n3);

        else if (n4 < n5)

            System.out.println("The smallest number is "+n4);

        else

            System.out.println("The smallest number is "+n5);

    }

}

}

o/p:

The smallest number is -3

WAP to find the type of character.

```

class TypeOfChar
{
    public static void main(String [] args)
    {
        char ch = '$';
        if (ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z')
            System.out.println ("The given char is a Alphabet");
        else if (ch >= '0' && ch <= '9')
            System.out.println ("The given char is a number");
        else
            System.out.println ("The given char is a special character");
    }
}

```

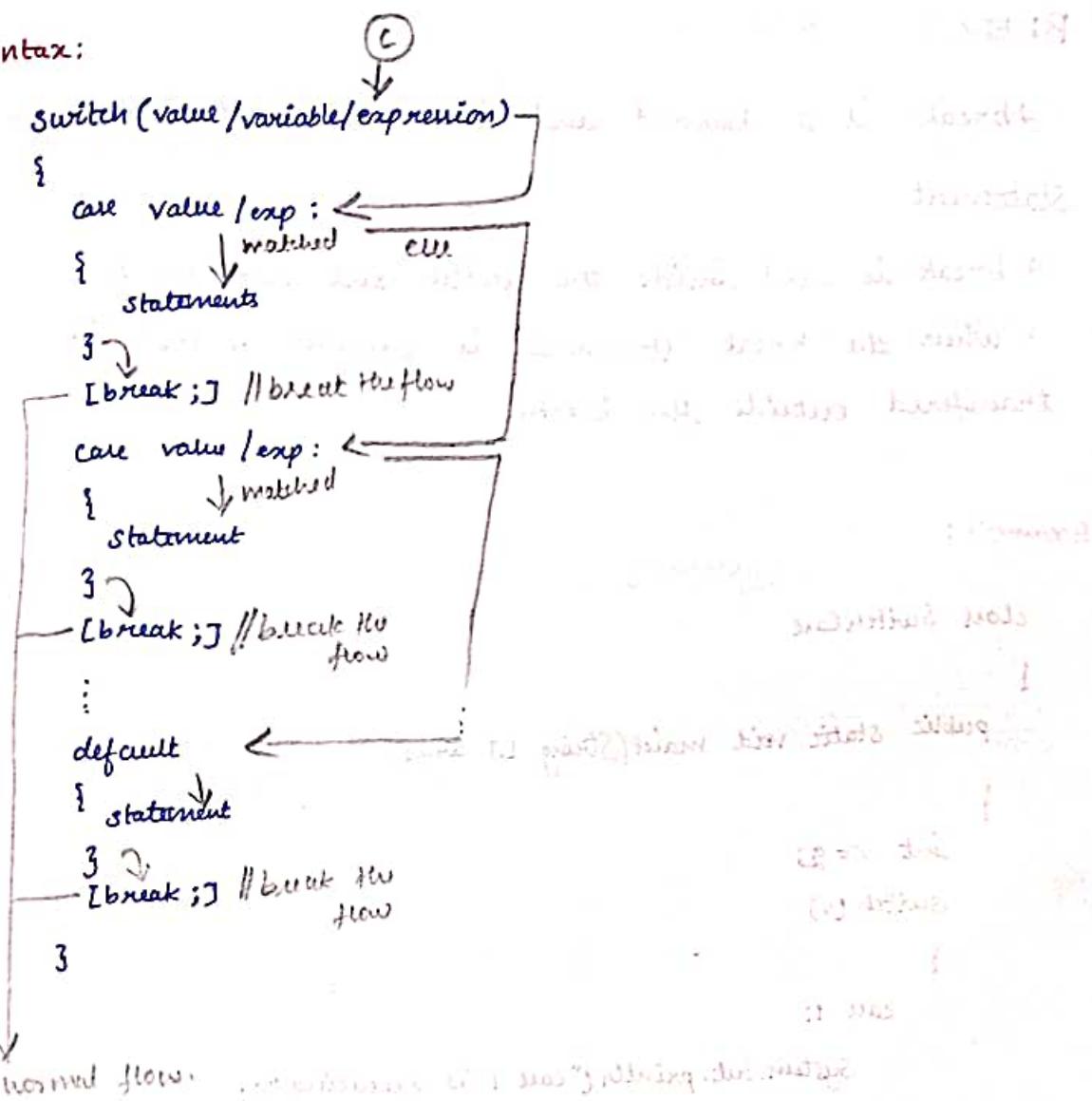
O/P:

The given char is a special character

### Switch Statement:

- \* Switch Statement is used for "pattern Matching"
- \* If the value is matching with the case means it will execute the block and control is transfer to the below cases also.
- \* In order to stop the flow of execution we have to use "break".

Syntax:



work flow:

- \* The value/variable/expression passed in the switch gets compared with value passed in the case.
- \* If any of a case is satisfied, the case block is executed and all the blocks present below gets executed
- \* If no case is satisfied then default gets executed
- \* For a case we can use a break statement which is optional

NOTE :

- For a switch we can't pass long, float, double, boolean
- For a case we can't pass variable.

## BREAK :

\*break is a keyword and it is a control transfer Statement

\* break is used inside the switch and loop block

\* when the break statement is executed control is transferred outside the block.

## Example :

```
class SwitchCase
{
    public static void main(String [] args)
    {
        int v=2;
        switch(v)
        {
            case 1:
                System.out.println("case 1 is executing");
                break;
            case 2:
                System.out.println("case 2 is executing");
                break;
            case 3:
                System.out.println("case 3 is executing");
            default:
                System.out.println("default case is executed");
        }
    }
}
```

Op:

case 2 is executing.

TC 1: v=1

with break:

case 1 is executing

without break:

case 1 is executing

case 2 is executing

case 3 is executing

default case is executed

TC 2: v=3

with break:

case 3 is executing

without break:

case 3 is executing

default case is executed.

WAP to design a Game: (by grouping)

class Game

{

    public static void main(String[] args)

{

        int task = 3;

        switch (task)

{

            case 1:

            case 4:

            case 6:

                System.out.println("sing");

                break;

            case 2:

            case 7:

                System.out.println("Dance");

                break;

            case 3:

            case 5:

                System.out.println("Truth");

                break;

            case 8:

                System.out.println("Story");

                break;

}

}

O/P: Truth

Example :

```
class LogicalFinder
{
    public static void main(String[] args)
    {
        int num1 = 5;
        int num2 = 68;
        int choice = 2;

        System.out.println("You have chosen " + choice + " function");

        switch (choice)
        {
            case 1:
                System.out.println("You have chosen addition");
                int c = num1 + num2;
                System.out.println("The addition of two numbers is " + c);
                break;

            case 2:
                System.out.println("You have chosen subtraction");
                int d = num1 - num2;
                System.out.println("The subtraction of two numbers is " + d);
                break;

            case 3:
                System.out.println("You have chosen multiplication");
                int e = num1 * num2;
                System.out.println("The multiplication of two numbers is " + e);
                break;
        }
    }
}
```

case 4:

```
{    System.out.println("you have chosen division");  
    int c = num1/num2;  
    System.out.println("The division of two number is "+c);  
}  
break;  
default  
    System.out.println("choose the correct function");  
}  
}  
}
```

Output:

T. case(i) choice = 2

you have chosen 2 function

you have chosen Subtraction

The subtraction of two number is -63

T. case(ii) choice = 1

you have chosen 1 function

you have chosen Addition

The addition of two number is 73

T. case(iii) choice = 4

you have chosen 4 function

you have chosen division

The division of two number is 0

T. case(iv) choice = 7

you have chosen 7 function.

choose the correct function

**Example:**

WAP to find the given year is leap year or not

```
class LeapYearOrNot
{
    public static void main(String [] args)
    {
        int year = 1700;
        if((year%4==0 && year%100!=0) || year%400==0)
            System.out.println (year + " is a leap year");
        else
            System.out.println (year + " is not a leap year");
    }
}
```

**O/P:**

1700 is not a leap year

**Example:**

WAP to find the given number is positive or negative or zero

```
class CheckInteger
{
    public static void main(String [] args)
    {
        int num=5;
        if (num>0)
            System.out.println ("The given number "+num+" is a positive number");
        else if (num<0)
            System.out.println ("The given number "+num+" is a negative number");
        else
            System.out.println ("The given number is 0");
    }
}
```

O/P:

The given number 5 is a positive number

Example:

Want to find the given character is vowel or consonant

class Vowel

{

public static void main (String [] args)

{

char ch = 's' ;

switch (ch)

{

case 'A':

case 'E':

case 'I':

case 'O':

case 'U':

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

System.out.println ("The given character is vowel");

break;

default:

System.out.println ("The given char is consonant");

}

}

O/P:

The given char is consonant

18/3/22

## Scanner (Read value from User)

## Dynamic Read :

The process of reading a data from the user during execution of a process is known as dynamic read.

## Step to Achieve Dynamic Read :

Step 1: Import Scanner class from java.util package

```
import java.util.Scanner;
```

Step 2: Create an object for the Scanner class.

```
Scanner input = new Scanner(System.in)
```

Step 3: call the method of Scanner class to read the data from the user

```
input.method();
```

## Methods in a Scanner Class :

Types of Data	Method signature	Return type
byte	nextByte()	byte
short	nextShort()	short
int	nextInt()	int
long	nextLong()	long
float	nextFloat()	float
double	nextDouble()	double
boolean	nextBoolean()	boolean
char	next().charAt(0)	char
String (s.w)	next()	String
String (n.w)	nextLine()	String

WAP to perform multiplication of two numbers by using Scanner.

```
import java.util.Scanner;
class Multiplication
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter the number");
        int num1 = s.nextInt();
        System.out.println("enter the 2nd number");
        int num2 = s.nextInt();
        System.out.println("The multiplication of two numbers is " + num1 * num2);
    }
}
```

O/P:  
enter the number

4

enter the 2nd number

10

The multiplication of two numbers is 40

WAP to display name, age and gender by reading value from the user.

```
class
import java.util.Scanner;
class Details
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Please enter your name");
        String name = sc.nextLine();
```

```
System.out.println("Please enter your age");
int age = sc.nextInt();

System.out.println("Please enter the gender");
char gender = sc.next().charAt(0);

System.out.println("The details are:");
System.out.println(name + " " + age + " " + gender);
}
```

}

Output:

Please enter your name

Lavanya L

Please enter your age

23

Please enter the gender

female

The details are:

Lavanya L 23 f

WAP to find the addition of two character.

```
import java.util.Scanner;
class AdditionChar
{
    public static void main(String [] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter the 1st char");
        char ch1 = s.next().charAt(0);
        System.out.println("Enter the 2nd char");
        char ch2 = s.next().charAt(0);
        System.out.println("Sum of two char is " + (ch1 + ch2));
    }
}
```

O/P:

enter the 1st char

a

enter the 2nd char

A

The addition of two char is 162

### Loop Statement

\* Loop statements help the programmer to execute the set of instruction repeatedly.

\* In java we have different types of loop statement, they are:

\* while loop

\* do while loop

\* for loop

\* for each/ advanced loop

\* nested loop

eg: If the programmer wants to print the same statement more than one time or many time we will go for looping.

#### NOTE:

To perform looping we have to follow three steps

They are :

→ declaration & initialization

→ condition

→ updation

## while Loop:

When the programmer do not know the number of iteration. Then he go for while loop.

### Syntax:

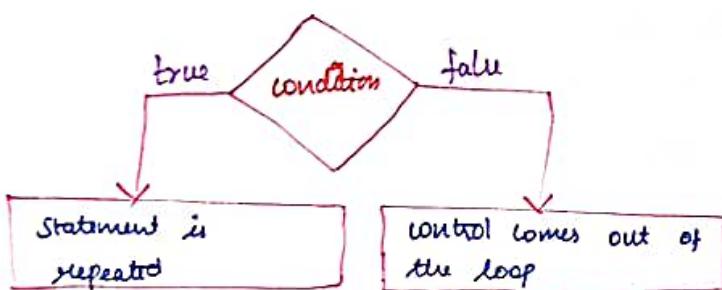
```
while (condition)
```

```
{
```

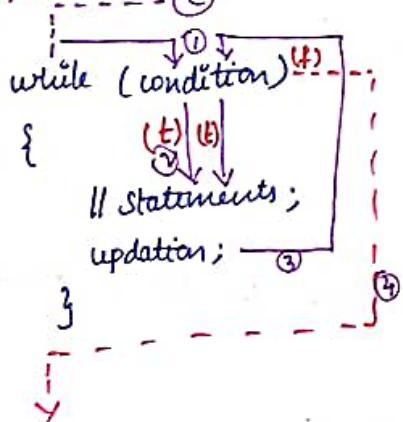
```
  //Statements ;
```

```
}
```

### Flow chart:



### work flow:



### Example:

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

Tracing:

i  
1 2 3 4 5 6

I     $i=1$

$1 <= 5$  (T)

S.o.println("Hi");

$i++$ ;

II     $i=2$

$2 <= 5$  (T)

S.o.println("Hi");

$i++$ ;

III     $i=3$

$3 <= 5$  (T)

S.o.println("Hi");

$i++$ ;

IV     $i=4$

$4 <= 5$  (T)

S.o.println("Hi");

$i++$ ;

V     $i=5$

$5 <= 5$  (T)

S.o.println("Hi");

$i++$ ;

VI     $i=6$

$6 <= 5$  (F)

comes out of the loop

OUTPUT SCREEN

Hi

Hi

Hi

Hi

Hi

WAP to print the first 10 Natural numbers.

```
class NaturalNumber  
{  
    public static void main(String[] args)  
    {  
        int i = 10;  
        while (i <= 10)  
        {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

Tracing:

i.

12345678910

(i)	$i \leq 10$ $1 \leq 10$ (T) $s.o.println(i)$ $i++$	(v) $i \leq 10$ $5 \leq 10$ (T) $s.o.println(i)$ $i++$	(ix) $i \leq 10$ $9 \leq 10$ (T) $s.o.println(i)$ $i++$	<u>OUTPUT</u> <u>SCREEN</u>
(ii)	$i \leq 10$ $2 \leq 10$ (T) $s.o.println(i)$ $i++$	(vi) $i \leq 10$ $6 \leq 10$ (T) $s.o.println(i)$ $i++$	(x) $i \leq 10$ $10 \leq 10$ (T) $s.o.println(i)$ $i++$	1 2 3 4 5
(iii)	$i \leq 10$ $3 \leq 10$ (T) $s.o.println(i)$ $i++$	(vii) $i \leq 10$ $7 \leq 10$ (T) $s.o.println(i)$ $i++$	(xi) $i \leq 10$ $11 \leq 10$ (F) comes out loop	6 7 8 9
(iv)	$i \leq 10$ <del><math>4 \leq 10</math></del> $s.o.println(i)$ $i++$	(viii) $i \leq 10$ <del><math>8 \leq 10</math></del> (T) $s.o.println(i)$ $i++$		10

WAP to print the even numbers in the given range

```
import java.util.Scanner;
class EvenNumber
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("enter the initial number");
        int m = s.nextInt();
        System.out.println("enter the ending number");
        int n = s.nextInt();
        System.out.println("the even numbers are");
        while(m <=n)
        {
            if (m % 2 == 0)
                System.out.println(m);
            m += 2;
        }
    }
}
```

m      n  
2 4 6 8      7

tracing:

(i)  $m \geq n$   
 $2 \leq 7 \text{ (T)}$   
 $2 \times 2 = 0 \text{ (T)}$   
 $s.o.pn(m)$   
 $m += 2$

(iv)  $m \leq n$   
 $8 \leq 7 \text{ (F)}$   
comes out of loop

(ii)  $m \leq n$   
 $4 \leq 7 \text{ (T)}$   
 $4 \times 2 = 0 \text{ (T)}$   
 $s.o.pn(m)$   
 $m += 2$

(iii)  $m \leq n$   
 $6 \leq 7 \text{ (T)}$   
 $6 \times 2 = 0 \text{ (T)}$   
 $s.o.pn(m)$   
 $m += 2$

OUTPUT

SCREEN:

enter the initial number  
2  
enter the ending number  
7

the even numbers are

2  
4  
6

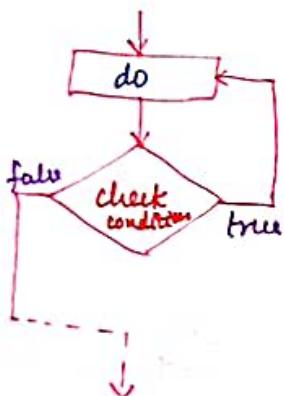
## do-while:

when the programmer wants to execute the instruction at least one time without checking the condition.  
we can go for do-while loop.

## Syntax:

```
do  
{  
    // Statements  
}  
while (condition);
```

## flow chart.

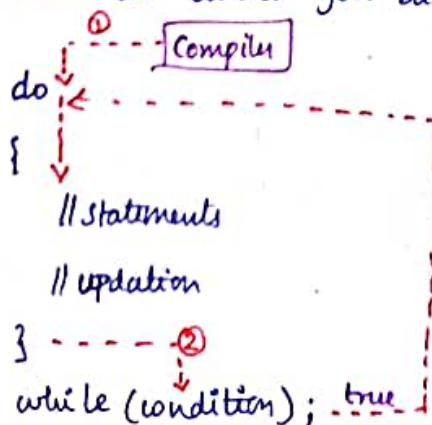


## work flow:

case (i) : when the condition is true .

\* control goes to the loop block directly , execute the instruction

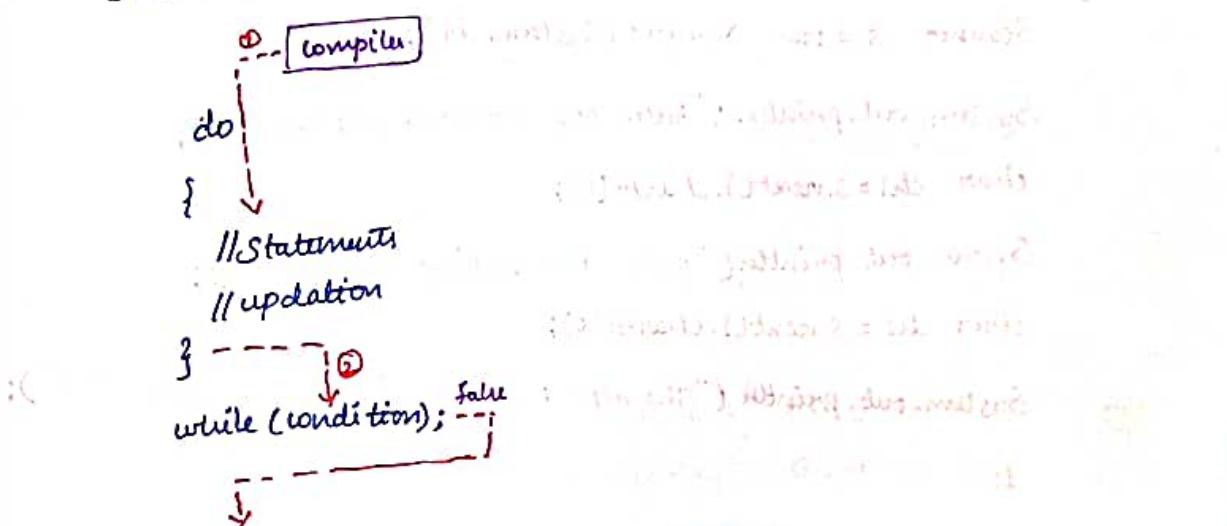
\* Then control goes to the condition , if the condition is true the control goes back to the do block .



case (ii) : when the condition is false

\* control goes to the loop block directly, execute the instruction

\* Then control goes to the condition, if the condition returns false the loop stops and control goes to the next statement.



### Difference between while And do-while Loop

#### WHILE

In while loop first condition being checked. if condition is satisfied then the task will be done

→ Syntax:

```
while (condition)\n{\n    statements;\n}
```

→ eg: class w1

```
{\n    public static void main(String\n        [ ] args)\n    {\n        int count = 1;
```

#### DO-WHILE

In do-while loop first it execute the task and then it will check the condition

→ Syntax:

```
do\n{\n    statement;\n}\nwhile (condition);
```

→ eg: class D1

```
{\n    public static void main(String\n        [ ] args)\n    {\n        int count = 1;
```

```
\n    }
```

**Example:** WAP to print the alphabets in the given range

```
import java.util.Scanner;  
class PrintAlphabets  
{  
    public static void main(String[] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the starting alphabet");  
        char ch1 = s.next().charAt(0);  
        System.out.println("Enter the ending alphabet");  
        char ch2 = s.next().charAt(0);  
        System.out.println("The alphabets in the given range are:");  
        do  
        {  
            System.out.print(ch1 + " ");  
            ch1++;  
        }  
        while (ch1 <= ch2);  
    }  
}
```

Tracing : ch1

ABCDE

ch2

D

OUTPUT SCREEN:

(i) S.o.println("A+ ");  
ch1++;  
ch1 <= ch2  
68 <= 68 (T)

(ii) S.o.println("B+ ");  
ch1++  
ch1 <= ch2  
68 <= 68 (T)

(iii) S.o.println("C+ ");  
ch1++  
ch1 <= ch2  
69 <= 68 (T)

(iv) S.o.println("D+ ");

ch1++

ch1 <= ch2  
69 <= 68 (F)

comes out of  
loop

Enter the starting alphabet

A

Enter the ending alphabet

D

The alphabets in the given range are:

A B C D

WAP to print the sum of natural numbers in the given range

```
import java.util.Scanner;  
class SumOfNaturals  
{  
    public static void main(String [] args)  
    {  
        Scanner s = new Scanner(System.in);
```

System.out.println("Enter the starting Number");

```
int num1 = s.nextInt();
```

System.out.println("Enter the ending number");

```
int num2 = s.nextInt();
```

```
int sum = 0;
```

System.out.println("The sum of natural numbers : ");

```
do
```

```
{
```

```
    sum = sum + num1;
```

```
    num1++;
```

```
}
```

```
while (num1 <= num2);
```

```
System.out.println(sum);
```

```
}
```

```
}
```

Tracing: num1    num2    sum  
1 2 3 4 5 6    5    0 1 3 6 10 15

(i) sum = sum + num1;  
Sum = 0 + 1;  
Sum = 1  
num++;  
2 <= 5 (T)

(iv) sum = 6 + 4  
sum = 10  
num++;  
5 <= 5 (T)

(ii) sum = 1 + 2;  
sum = 3  
num++;  
3 <= 5 (T)

(v) sum = 10 + 5  
sum = 15  
num++;  
6 <= 5 (F)

(iii) sum = 3 + 3  
sum = 6  
num++;

comes of the loop

OUTPUT SCREEN:

Enter the starting Number  
1

Enter the ending Number  
5

The sum of natural numbers:  
15

**for loop:** it is a programming construct to repeat code.

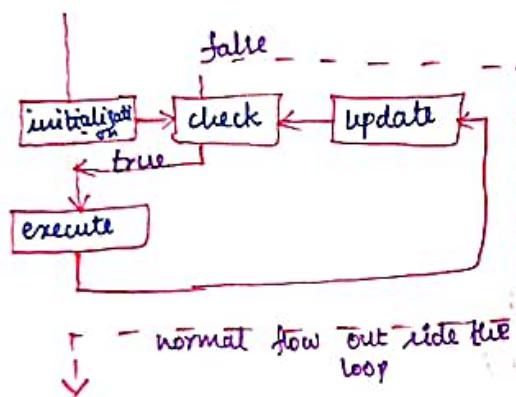
\* When the program knows the number of iteration we can go with for loop.

\* In for loop the basic & important steps three steps (ie) declaration & initialization, condition, update are given in the same line.

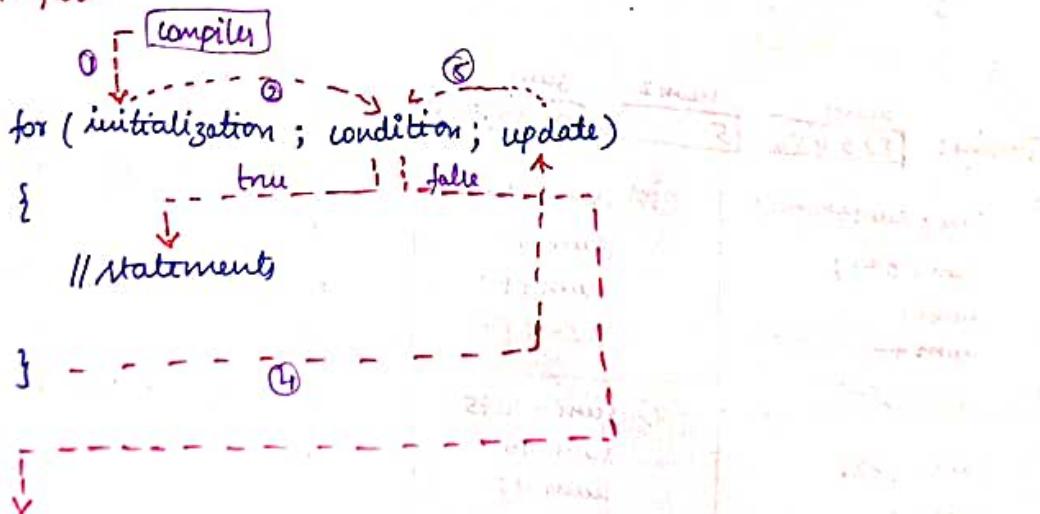
**Syntax:**

```
for (initialization; condition; update)
{
    // statement
}
```

**flow chart:**



**work flow:**



WAP to check the given number is prime or not.

```
import java.util.Scanner;  
class Prime  
{  
    public static void main(String[] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number");  
        int num = s.nextInt();  
        int count = 0; i;  
        for(i=1; i<=num; i++)  
        {  
            if(num % i == 0)  
            {  
                count++;  
            }  
        }  
        if(count == 2)  
        System.out.println("The given number " + num + " is prime");  
        else  
        System.out.println("The given number " + num + " is not prime");  
    }  
}
```

Tracing:

num:	count	i
3	0,1,2	1,2,3,4

(i)       $i=1$   
 $i \leq 3$  (T)  
if ( $3 \% 1 == 0$ ) (T)  
Count++

(iv)       $i=4$   
 $4 \leq 3$  (F)  
comes out of loop

(ii)       $i=2$   
 $2 \leq 3$  (T)  
if ( $3 \% 2 == 0$ ) (F)

if (count == 2)  
 $2 == 2$  (T)  
S.O.PL("The given number " + num + " is prime");

(iii)       $i=3$   
 $3 \leq 3$  (T)  
if ( $3 \% 3 == 0$ ) (T)  
Count++

### OUTPUT SCREEN

Enter the number

3  
The given number 3 is prime

19/3/22

## Programming Session.

2)

- 1) WAP to check whether the given number is prime or not.

```

import java.util.Scanner;
class PrimeOrNot
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter the number.");
        int num = s.nextInt();
        int count = 0;
        for(int i=1; i<=num; i++)
        {
            if(num % i == 0)
            {
                count++;
            }
        }
        if(count == 2)
        {
            System.out.println("The given number " + num + " is a prime");
        }
        else
            System.out.println("The given number " + num + " is not a prime");
    }
}

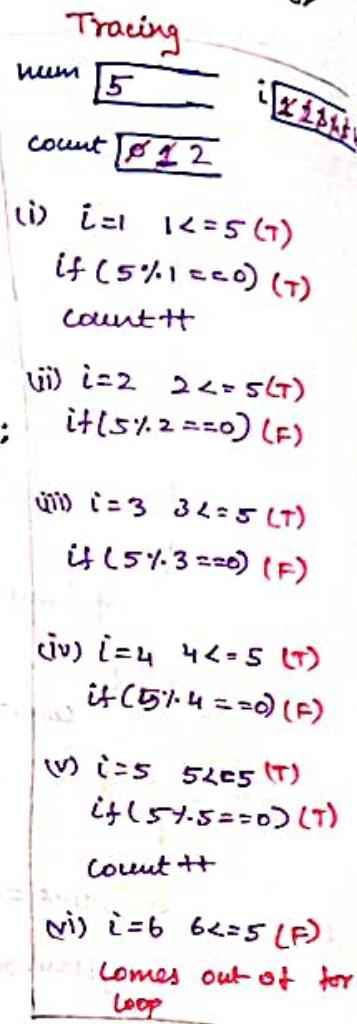
```

## OUTPUT:

Enter the number

5

The given number 5 is a prime.



```

if (count == 2)
    2 == 2 (T)
    S.O..Println("Prime");

```

2) WAP to find the factorial of the given number.

```
import java.util.Scanner;  
class Factorial  
{  
    public static void main(String [] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number");  
        int num = s.nextInt();  
        int fact = 1;  
        for (int i=1; i<=num; i++)  
        {  
            fact = fact * i;  
        }  
        System.out.println("The factorial of the given number " + num +  
                           " is " + fact);  
    }  
}
```

OUTPUT:

Enter the number

5

The factorial of the given number 5 is 120

Tracing:

num	fact	i
5	1 1 2 6 2 4 1 2	1 2 3 4 5 6

- |  |  |   |   |
|--|--|---|---|
| (i) $i=1 \quad 1 \leq 5(T)$<br>$fact = 1 * 1$<br>$fact = 1$    | (ii) $i=2 \quad 2 \leq 5(T)$<br>$fact = 1 * 2$<br>$fact = 2$ | (iii) $i=3 \quad 3 \leq 5(T)$<br>$fact = 2 * 3$<br>$fact = 6$ | (iv) $i=4 \quad 4 \leq 5(T)$<br>$fact = 6 * 4$<br>$fact = 24$ |
| (v) $i=5 \quad 5 \leq 5(T)$<br>$fact = 24 * 5$<br>$fact = 120$ | (vi) $i=6 \quad 6 \leq 5(F)$<br>comes out of<br>for loop     |   |   |

3) WAP to print the factors of the given number

```
import java.util.Scanner;  
class Factors  
{  
    public static void main(String [] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number : ");  
        int num = s.nextInt();  
        System.out.print("The factors of the given number is : ");  
        for(int i=1; i<=num; i++)  
        {  
            if(num % i == 0)  
            {  
                System.out.print(i + " ");  
            }  
        }  
    }  
}
```

OUTPUT :

Enter the number

4

The factors of the given number is: 1 2 4

Tracing

num	i
4	1 / 2 / 3 X 4 5

(i)  $i=1 \quad 1 \leq 4(T)$   
 $\text{if}(4 \% 1 == 0)(T)$   
S.O.P(1)

(ii)  $i=2 \quad 2 \leq 4(T)$   
 $\text{if}(4 \% 2 == 0)(T)$   
S.O.P(2)

(iii)  $i=3 \quad 3 \leq 4(T)$   
 $\text{if}(4 \% 3 == 0)(F)$   
S.O.P(4)

(iv)  $i=4 \quad 4 \leq 4(T)$   
 $\text{if}(4 \% 4 == 0)(T)$   
S.O.P(4)

(v)  $i=5 \quad 5 \leq 4(F)$   
comes out of  
for loop

1) WAP to print the number of digits in the given number.

```
import java.util.Scanner;  
class CountDigits  
{  
    public static void main(String [] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number to find the number of digits");  
        int num = s.nextInt();  
        int count = 0;  
        System.out.print("The number of digits present in the given  
                         number are ");  
        while (num > 0)  
        {  
            num = num / 10;  
            count++;  
        }  
        System.out.println(count);  
    }  
}
```

OUTPUT:

Enter the number to find the number of digits

58127

The number of digits present in the given number are 5

Tracing:

num	count
<u>58127</u>	<u>0</u>
<u>5812</u>	<u>1</u>
<u>581</u>	<u>2</u>
<u>58</u>	<u>3</u>
<u>5</u>	<u>4</u>
<u></u>	<u>5</u>

(i)  $58127 > 0$  (T)  
 $num = 58127 / 10$   
 $num = 5812$   
 $count++$

(ii)  $5812 > 0$  (T)  
 $num = 5812 / 10$   
 $num = 581$   
 $count++$

(iii)  $581 > 0$  (T)  
 $num = 581 / 10$   
 $num = 58$   
 $count++$

(iv)  $58 > 0$  (T)  
 $num = 58 / 10$   
 $num = 5$   
 $count++$

(v)  $5 > 0$  (T)  
 $num = 5 / 10$   
 $num = 0$   
 $count++$

(vi)  $0 > 0$  (F)  
comes out  
of the loop

numbers.

```
import java.util.Scanner;  
class SumOfEvenNumber  
{  
    public static void main(String[] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number");  
        int num = s.nextInt();  
        int store, sum = 0;  
        while (num != 0)  
        {  
            store = num % 10;  
            if (store % 2 == 0)  
            {  
                sum = sum + store;  
            }  
            num /= 10;  
        }  
        System.out.println("The sum of even number is :" + sum);  
    }  
}
```

OUTPUT :

Enter the number

5183414

The sum of even number is 16

(vii)  $5 \neq 0$  (T)

store =  $5 \cdot 10$

store = 5

if ( $5 \cdot 2 == 0$ ) (F)

num =  $5 / 10 \Rightarrow$  num = 0

(viii)  $0 \neq 0$  (F)

comes out of the

loop

(v)  $51 \neq 0$  (T)

store =  $51 \cdot 10$

store = 8

if ( $8 \cdot 2 == 0$ ) (T)

sum =  $8 + 8$

sum = 16

num = num / 10

num = 51

(vi)  $51 \neq 0$  (T)

store =  $51 \cdot 10 \Rightarrow$  store = 1

if ( $1 \cdot 2 == 0$ ) (F)

num =  $51 / 10 \Rightarrow$  num = 5

Tracing

num	5183414	store	4
sum	16		16

(i)  $5183414 \neq 0$  (T) (ii)  $518341 \neq 0$  (T)

store =  $5183414 \cdot 10$

store = 4

if ( $4 \cdot 2 == 0$ ) (T)

sum =  $0 + 4$

sum = 4

num =  $5183414 / 10$

num = 518341

store =  $518341 \cdot 10$

store = 1

if ( $1 \cdot 2 == 0$ ) (F)

num =  $518341 / 10$

num = 51834

(iv)  $5183 \neq 0$  (T)

store =  $5183 \cdot 10$

store = 3

if ( $3 \cdot 2 == 0$ ) (F)

num =  $5183 / 10$

num = 518

(iii)  $51834 \neq 0$  (T)

store =  $51834 \cdot 10$

store = 4

if ( $4 \cdot 2 == 0$ ) (T)

sum =  $4 + 4$

sum = 8

num =  $51834 / 10$

num = 5183

b) WAP to print the reversing order of the given number (without storing)

```
import java.util.Scanner;  
class ReversingNumber  
{  
    public static void main (String [] args)  
    {  
        Scanner s = new Scanner (System.in);  
        System.out.print ("Enter the number to reverse");  
        int num = s.nextInt();  
        int temp = num, rev;  
        while (num != 0)  
        {  
            rev = num % 10;  
            System.out.print (rev);  
            num /= 10;  
        }  
    }  
}
```

OUTPUT:

Enter the number to reverse

1523

3251

Tracing

num	rev
15230	3251

(i) 1523 != 0 (T)

rev = 1523 % 10

s.o.p (rev) = 3

num = 1523 / 10

num = 152

(ii) 152 != 0 (T)

rev = 152 % 10

rev = 2

s.o.p (2)

num = 152 / 10

num = 15

(iii) 15 != 0 (T)

rev = 15 % 10

rev = 5

s.o.p (5)

num = 15 / 10

num = 1

(iv) 1 != 0 (T)

rev = 1 % 10

rev = 1

s.o.p (1)

num = 1 / 10

num = 0

(v) 0 != 0 (F)

comes out  
of loop

i) WJP) to print the reversing order of the given number (with storing)

```
import java.util.Scanner;  
class ReversingNumber  
{  
    public static void main(String [] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number to reverse");  
        int num = s.nextInt();  
        int temp = num, rev = 0, store;  
        while (num != 0)  
        {  
            store = num % 10;  
            rev = rev * 10 + store;  
            num /= 10;  
        }  
        System.out.println("The reversing order of the given number " +  
                           "is " + rev);  
    }  
}
```

#### OUTPUT :

Enter the number to reverse

5237

The reversing order of the given number 5237 is 7325

#### Tracing:

num	temp	rev	store
5 2 3 7	5 2 3 7	0 7 3 2 5	7 3 2 5

(i)  $num \neq 0$  (T)

$$5237 \neq 0$$

$$store = 5237 \% 10$$

$$store = 7$$

$$rev = 0 * 10 + 7$$

$$rev = 7$$

$$num = 5237 / 10$$

$$num = 523$$

(ii)  $523 \neq 0$  (T)

$$store = 523 \% 10$$

$$store = 3$$

$$rev = 7 * 10 + 3$$

$$rev = 73$$

$$num = 523 / 10$$

$$num = 52$$

(iii)  $52 \neq 0$  (T)

$$store = 52 \% 10$$

$$store = 2$$

$$rev = 73 * 10 + 2$$

$$rev = 732$$

$$num = 52 / 10$$

$$num = 5$$

(iv)  $5 \neq 0$  (T)

$$store = 5 \% 10$$

$$store = 5$$

$$rev = 732 * 10 + 5$$

$$rev = 7325$$

$$num = 5 / 10$$

$$num = 0$$

(v)  $0 \neq 0$  (F)  
comes out of the loop

8) WAPP to check the given number is palindrome or not.

```
import java.util.Scanner;  
class Palindrome  
{  
    public static void main(String[] args)  
    {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Enter the number to reverse");  
        int num = s.nextInt();  
        int temp = num, rev = 0, store;  
        while (num != 0)  
        {  
            store = num % 10;  
            rev = rev * 10 + store;  
            num = num / 10;  
        }  
        if (rev == temp)  
            System.out.println("The given number is palindrome");  
        else  
            System.out.println("The given number is not palindrome");  
    }  
}
```

OUTPUT:

Enter the number to reverse

12521

The given number is palindrome.

### Tracing

num	temp
1 2 5 2 1	1 2 5 2 1
rev	s.

0 1 2 1 2 5 1 2 5 2 1	1 2 5 2 1
-----------------------	-----------

(i) 12521 != 0 (T)

$$store = 12521 \% 10$$

$$store = 1$$

$$rev = 0 * 10 + 1$$

$$rev = 1$$

$$num = 12521 / 10$$

$$num = 1252$$

(ii) 1252 != 0 (T)

$$store = 1252 \% 10$$

$$store = 2$$

$$rev = 1 * 10 + 2$$

$$rev = 12$$

$$num = 1252 / 10$$

$$num = 125$$

(iii) 125 != 0 (T)

$$store = 125 \% 10$$

$$store = 5$$

$$rev = 12 * 10 + 5$$

$$rev = 125$$

$$num = 125 / 10$$

$$num = 12$$

(iv) 12 != 0 (T)

$$store = 12 \% 10$$

$$store = 2$$

$$rev = 12 * 10 + 2$$

$$rev = 122$$

$$num = 12 / 10$$

$$num = 1$$

(v) 1 != 0 (T)

$$store = 1 \% 10$$

$$store = 1$$

$$rev = 122 * 10 + 1$$

$$rev = 1221$$

$$num = 1 / 10$$

$$num = 0$$

(vi) 0 != 0 (F)

comes out of  
loop

if (rev == temp)

$$1221 == 1221 \text{ (T)}$$

S.O.PLn ("palindrome");

9. WAP to print the prime numbers in the given range

```
import java.util.Scanner;
```

```
class PalindromeInRange
```

```
{
```

```
    public static void main(String[] args)
```

```
{
```

```
    Scanner s = new Scanner(System.in);
```

```
    System.out.print("Enter the starting number ");
```

```
    int num1 = s.nextInt();
```

```
    System.out.print("Enter the ending number ");
```

```
    int num2 = s.nextInt();
```

```
    System.out.print("The prime numbers in the given range are: ");
```

```
    { int count = 0;
```

```
        for (int j = 1; j <= i; j++)
```

```
        { if (i % j == 0)
```

```
            count++;
```

```
        }
```

```
        if (count == 2)
```

```
            System.out.print(i + " ");
```

```
}
```

```
}
```

OUTPUT:

Enter the starting number

3

Enter the ending number

7

The prime numbers in the given range are : 3 5 7

Tracing

num1	num2
3	7
i	
3 4 5 6 7 8	
(i) 3 <= 7 (T)	(ii) 4 <= 7 (T)
count [0 X 2]	count [0 1 2 3]
j [1 2 3 4	j [1 2 3 4 5
(i) 1 <= 3 (T)	(ii) 1 <= 4 (T)
3 * 1 == 0 (T)	4 * 1 == 0 (T)
count++	count++
(ii) 2 <= 3 (F)	(ii) 2 <= 4 (T)
3 * 2 == 0 (F)	4 * 2 == 0 (T)
count++	count++
(iii) 3 <= 3 (T)	(iii) 3 <= 4 (T)
3 * 3 == 0 (T)	4 * 3 == 0 (F)
count++	count++
if (count == 2) (T)	(iv) 4 <= 4 (T)
2 == 2	4 * 4 == 0 (T)
s.o.p(3)	count++
	5 == 4 (F)
	if (count == 2) (T)
	2 == 2

(3) 5 <= 7 (T)

count [0 X 2]

j [1 2 3 4 5 6

(i) 1 <= 5 (T)

5 \* 1 == 0 (T)

count++

(iii) 2 <= 5 (T)

5 \* 2 == 0 (F)

(iii) 3 <= 5 (T)

5 \* 3 == 0 (F)

(iv) 4 <= 5 (T)

5 \* 4 == 0 (F)

(4) 6 <= 7 (T)

count [0 1 2 3 4

j [1 2 3 4 5 6 7

(i) 1 <= 6 (T)

6 \* 1 == 0 (T)

count++

(ii) 2 <= 6 (T)

6 \* 2 == 0 (T)

(iii) 3 <= 6 (T)

6 \* 3 == 0 (T)

(iv) 4 <= 6 (T)

6 \* 4 == 0 (F)

(5) 7 <= 7 (T)

count [0 1 2

j [1 2 3 4 5 6 7 8

(i) 1 <= 7 (T)

7 \* 1 == 0 (T)

count++

(ii) 2 <= 7 (T)

7 \* 2 == 0 (T)

(iii) 3 <= 7 (T)

7 \* 3 == 0 (F)

(iv) 4 <= 7 (T)

7 \* 4 == 0 (F)

(v) 5 <= 7 (T)

7 \* 5 == 0 (F)

(vi) 6 <= 7 (F)

7 \* 6 == 0 (F)

(vii) 7 <= 7 (T)

7 \* 7 == 0 (T)

(viii) 8 <= 7 (F)

7 \* 8 == 0 (F)

(ix) 9 <= 7 (F)

7 \* 9 == 0 (F)

(x) 10 <= 7 (F)

7 \* 10 == 0 (F)

(xi) 11 <= 7 (F)

7 \* 11 == 0 (F)

(xii) 12 <= 7 (F)

7 \* 12 == 0 (F)

(xiii) 13 <= 7 (F)

7 \* 13 == 0 (F)

(xiv) 14 <= 7 (F)

7 \* 14 == 0 (F)

(xv) 15 <= 7 (F)

7 \* 15 == 0 (F)

(xvi) 16 <= 7 (F)

7 \* 16 == 0 (F)

(xvii) 17 <= 7 (F)

7 \* 17 == 0 (F)

(xviii) 18 <= 7 (F)

7 \* 18 == 0 (F)

(xix) 19 <= 7 (F)

7 \* 19 == 0 (F)

(xx) 20 <= 7 (F)

7 \* 20 == 0 (F)

(xxi) 21 <= 7 (F)

7 \* 21 == 0 (F)

(xxii) 22 <= 7 (F)

7 \* 22 == 0 (F)

(xxiii) 23 <= 7 (F)

7 \* 23 == 0 (F)

(xxiv) 24 <= 7 (F)

7 \* 24 == 0 (F)

(xxv) 25 <= 7 (F)

7 \* 25 == 0 (F)

(xxvi) 26 <= 7 (F)

7 \* 26 == 0 (F)

(xxvii) 27 <= 7 (F)

7 \* 27 == 0 (F)

(xxviii) 28 <= 7 (F)

7 \* 28 == 0 (F)

(xxix) 29 <= 7 (F)

7 \* 29 == 0 (F)

(xxx) 30 <= 7 (F)

7 \* 30 == 0 (F)

(xxxi) 31 <= 7 (F)

7 \* 31 == 0 (F)

(xxxii) 32 <= 7 (F)

7 \* 32 == 0 (F)

(xxxiii) 33 <= 7 (F)

7 \* 33 == 0 (F)

(xxxiv) 34 <= 7 (F)

7 \* 34 == 0 (F)

(xxxv) 35 <= 7 (F)

7 \* 35 == 0 (F)

(xxxvi) 36 <= 7 (F)

7 \* 36 == 0 (F)

(xxxvii) 37 <= 7 (F)

7 \* 37 == 0 (F)

(xxxviii) 38 <= 7 (F)

7 \* 38 == 0 (F)

(xxxix) 39 <= 7 (F)

7 \* 39 == 0 (F)

(xxx) 40 <= 7 (F)

7 \* 40 == 0 (F)

(xxxi) 41 <= 7 (F)

7 \* 41 == 0 (F)

(xxxii) 42 <= 7 (F)

7 \* 42 == 0 (F)

(xxxiii) 43 <= 7 (F)

7 \* 43 == 0 (F)

(xxxiv) 44 <= 7 (F)

7 \* 44 == 0 (F)

(xxxv) 45 <= 7 (F)

7 \* 45 == 0 (F)

(xxxvi) 46 <= 7 (F)

7 \* 46 == 0 (F)

(xxxvii) 47 <= 7 (F)

7 \* 47 == 0 (F)

(xxxviii) 48 <= 7 (F)

7 \* 48 == 0 (F)

(xxxix) 49 <= 7 (F)

7 \* 49 == 0 (F)

(xxx) 50 <= 7 (F)

7 \* 50 == 0 (F)

(xxxi) 51 <= 7 (F)

7 \* 51 == 0 (F)

(xxxii) 52 <= 7 (F)

7 \* 52 == 0 (F)

(xxxiii) 53 <= 7 (F)

7 \* 53 == 0 (F)

(xxxiv) 54 <= 7 (F)

7 \* 54 == 0 (F)

(xxxv) 55 <= 7 (F)

7 \* 55 == 0 (F)

(xxxvi) 56 <= 7 (F)

7 \* 56 == 0 (F)

(xxxvii) 57 <= 7 (F)

7 \* 57 == 0 (F)

(xxxviii) 58 <= 7 (F)

7 \* 58 == 0 (F)

(xxxix) 59 <= 7 (F)

7 \* 59 == 0 (F)

(xxx) 60 <= 7 (F)

7 \* 60 == 0 (F)

(xxxi) 61 <= 7 (F)

7 \* 61 == 0 (F)

(xxxii) 62 <= 7 (F)

7 \* 62 == 0 (F)

(xxxiii) 63 <= 7 (F)

7 \* 63 == 0 (F)

(xxxiv) 64 <= 7 (F)

7 \* 64 == 0 (F)

(xxxv) 65 <= 7 (F)

7 \* 65 == 0 (F)

(xxxvi) 66 <= 7 (F)

7 \* 66 == 0 (F)

(xxxvii) 67 <= 7 (F)

7 \* 67 == 0 (F)

(xxxviii) 68 <= 7 (F)

7 \* 68 == 0 (F)

(xxxix) 69 <= 7 (F)

7 \* 69 == 0 (F)

(xxx) 70 <= 7 (F)

7 \* 70 == 0 (F)

(xxxi) 71 <= 7 (F)

7 \* 71 == 0 (F)

(xxxii) 72 <= 7 (F)

7 \* 72 == 0 (F)

(xxxiii) 73 <= 7 (F)

7 \* 73 == 0 (F)

(xxxiv) 74 <= 7 (F)

7 \* 74 == 0 (F)

(xxxv) 75 <= 7 (F)

7 \* 75 == 0 (F)

(xxxvi) 76 <= 7 (F)

7 \* 76 == 0 (F)

(xxxvii) 77 <= 7 (F)

7 \* 77 == 0 (F)

(xxxviii) 78 <= 7 (F)

7 \* 78 == 0 (F)

(xxxix) 79 <= 7 (F)

7 \* 79 == 0 (F)

(xxx) 80 <= 7 (F)

7 \* 80 == 0 (F)

(xxxi) 81 <= 7 (F)

7 \* 81 == 0 (F)

(xxxii) 82 <= 7 (F)

7 \* 82 == 0 (F)

(xxxiii) 83 <= 7 (F)

7 \* 83 == 0 (F)

(xxxiv) 84 <= 7 (F)

7 \* 84 == 0 (F)

(xxxv) 85 <= 7 (F)

7 \* 85 == 0 (F)

(xxxvi) 86 <= 7 (F)

7 \* 86 == 0 (F)

(xxxvii) 87 <

## Methods :

what is method?

- \* method is a block of instruction which is used to perform a specific task

- \* It is used to transfer a data.

method Syntax:

```
[access modifier] [modifier] return type methodname([datatype vari, ...datatype vari])
{  
}
```

Technical words of method definition:

- \* method signature

- \* method declaration

- \* method definition

method signature:

- \* method name

- \* formal argument

method declaration:

- \* Access modifier

- \* modifier

- \* Return type

- \* Signature

method definition:

- \* method declaration

- \* method body / implementation / block.

Access modifiers:

Access modifiers are the keywords which is responsible for the accessibility of the method. ie, Access modifier is used to modify the accessibility of the method.

We have 4 levels of access modifiers.

- \* private
- \* public
- \* protected
- \* default.

### modifiers:

modifiers are the keywords which are responsible to modify the characteristics of the method.

e.g.:

- \* static
- \* abstract
- \* final
- \* synchronized
- \* volatile
- \* transient.

### Return type:

- \* The method after execution can return a value back to the caller.
- \* Therefore it is mandatory to specify what type of data returned by the method in the method declaration statement, this is done with the help of return type.

### Return type definition:

Return type is a data type which specifies what type of data is returned by the method after execution.

types:

- \* void
- \* primitive datatype
- \* Non-primitive datatype

### void:

- \* void is a data type which is used as a return type when the method returns nothing.
- \* It is a keyword in java.

### NOTE:

- \* A method can't create inside another method.
- \* A class can have any number of methods. But, a class have only one main method.

### NOTE:

we can't print void method. if we try to print we will get CTE.

### Tracing (method calling)

```
class P1{  
    public static void main(String[] args){  
        System.out.println("main Starts");  
        pd();  
        System.out.println("main ends");  
    }  
    public static void pd(){  
        System.out.println("Good Morning");  
        raja();  
        System.out.println("Byee ");  
    }  
    public static void raja(){  
        System.out.println("Good morning sir");  
        manju();  
        System.out.println("Good morning manju");  
    }  
    public static void manju(){  
        System.out.println("Good morning raja");  
        pd();  
        System.out.println("Byee Sir");  
    }  
}
```



class P2 {

    public static void main(String [] args) {

        System.out.println("Main start");

        add();

        add();

        System.out.println("Main end");

}

    public static void add()

    {

        int a=20;

        int b=40;

        int res=a+b;

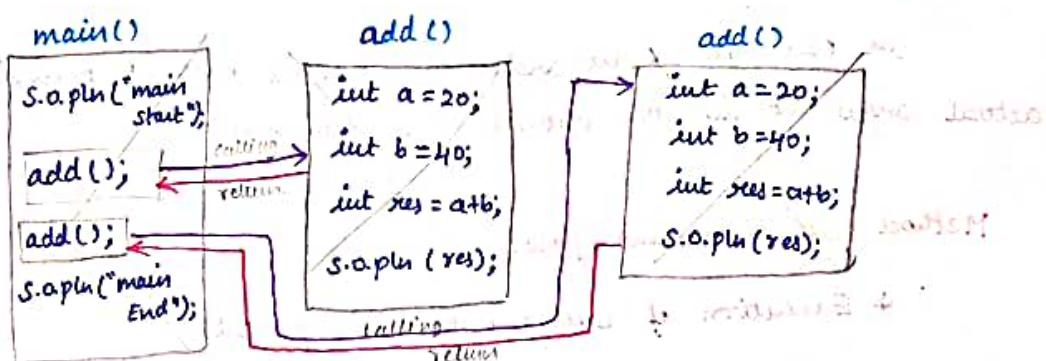
        System.out.println(res);

}

}

**Tracing:**

JVM calling main method.



**OUTPUT:**

Main start

60

60

Main End

## Method call statement:

The statement which is used to call a method is known as method call statement.

### Syntax:

method name([actual argument]);

### NOTE:

A method will get executed only when it is called, we can call a method with the help of method call statement.

## Types of methods Based on arguments:

The methods are classified into two based on arguments.

\* No argument method

\* Parameterized method.

### NOTE:

We can call a no argument method without passing actual argument in the method call statement.

## Method call statement flow:

- \* Execution of calling method is paused
- \* Control is transferred to the called method
- \* Execution of called method begins
- \* Once the execution of called method is completed the control is transferred back to the calling method.
- \* Execution of calling method resumes.

## Calling Method Vs Called Method.

### Calling Method:

The method which is trying to call another method is known as calling method (caller)

### Called Method:

The method which is being called by the caller is known as called method.

## Main Method

### Main Method:

- ⊗ [The execution of the program always starts from main method and ends at main method only] ⊗

```
public static void main(String[] args)
```

```
{
```

```
}
```

### Purpose of main method:

Start the execution

control the flow of the execution

End of execution

### Note:

\* A method can be executed only when it is called, we can call a method any number of times, therefore it is said to be code reusability.

\* Main method is always called by JVM

Example :

WAP to create an application called calculator

```
class Calculator {
```

```
    public static void add(int a, int b) {
```

```
        int res = a+b;
```

System.out.println("The addition of two number is " + res);

```
}
```

```
    public static void sub(int a, int b) {
```

```
        int res = a-b;
```

System.out.println("The difference of two number is " + res);

```
}
```

```
    public static void multiply(int a, int b) {
```

```
        int res = a*b;
```

System.out.println("The product of two number is " + res);

```
}
```

```
    public static void division(int a, int b) {
```

```
        int res = a/b;
```

System.out.println("The division of two number is " + res);

```
}
```

```
}
```

import java.util.Scanner;  
class CalculatorDriver {

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

```
        Scanner s = new Scanner(System.in);
```

System.out.println("1. Addition \n2. Subtraction \n3. Multiplication

\n4. division");

System.out.println("choose the operation which you want to

```
        int choice = s.nextInt();
```

perform");

```
        switch(choice)
```

```
}
```

case 1:

```
{ System.out.println("Enter the first number");
    int a = s.nextInt();
    System.out.println("Enter the second number");
    int b = s.nextInt();
    Calculator.add(a,b);
}
```

break;

case 2:

```
{ System.out.println("Enter the first number");
    int a = s.nextInt();
    System.out.println("Enter the second number");
    int b = s.nextInt();
    Calculator.sub(a,b);
}
```

break;

case 3:

```
{ System.out.println("Enter the first number");
    int a = s.nextInt();
    System.out.println("Enter the second number");
    int b = s.nextInt();
    Calculator.multiply(a,b);
}
```

break;

case 4:

```
{ System.out.println("Enter the first number");
    int a = s.nextInt();
    System.out.println("Enter the second number");
    int b = s.nextInt();
    Calculator.division(a,b);
}
```

}

}

java class calculatorDriver (executing)

OUTPUT:

1. Addition
2. Subtraction
3. Multiplication
4. Division.

2

Enter the first number

20

Enter the second number

9

The difference of two number is

11



## Types of Method :

Based on number of arguments, methods can be classified into two types,

No argument method

Parameterized method

### No argument method:

A method which does not have formal argument is known as no argument method.

### Example:

```
public static void demo() {  
    System.out.println("demo - no argument method");  
}
```

### Parameterized method:

\* The method which has formal argument is known as parameterized method.

\* Parameterized methods are used to accept the data.

### Formal argument:

A variable which is declared in a method declaration is known as formal argument.

### Actual argument:

The values passed in the method call statement is known as actual argument.

### Rules to call the parameterized method:

\* The number of actual argument should be same as the number of formal argument.

\* The type of corresponding actual argument should be same as the type of formal argument, if not compiler tries implicit conversion if it is not possible then we will get CTE

## Session 5 : Activity

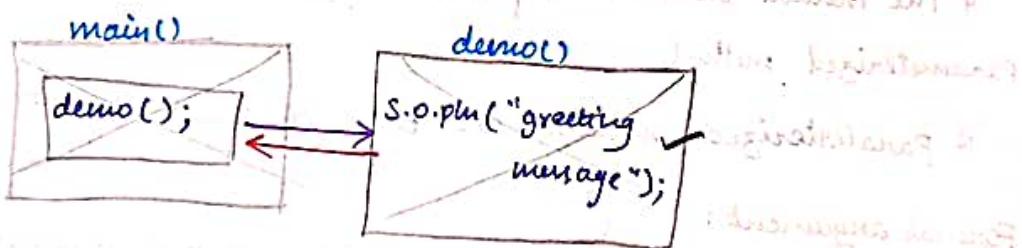
1. Design a method which is used to print "Greeting message" when it is called.

```
class E1 {  
    public static void main(String[] args) {  
        demo();  
    }  
    public static void demo() {  
        System.out.println("Greeting message");  
    }  
}
```

OUTPUT:

Greeting message

Tracing:



2. Do tracing on book for the given code.

```
class Demo {  
    public static void temp() {  
        System.out.println("Hi.....");  
    }  
    public static void main(String[] args) {  
        System.out.println("Start");  
        System.out.println("flow");  
        temp();  
        System.out.println("end");  
    }  
}
```

### OUTPUT :

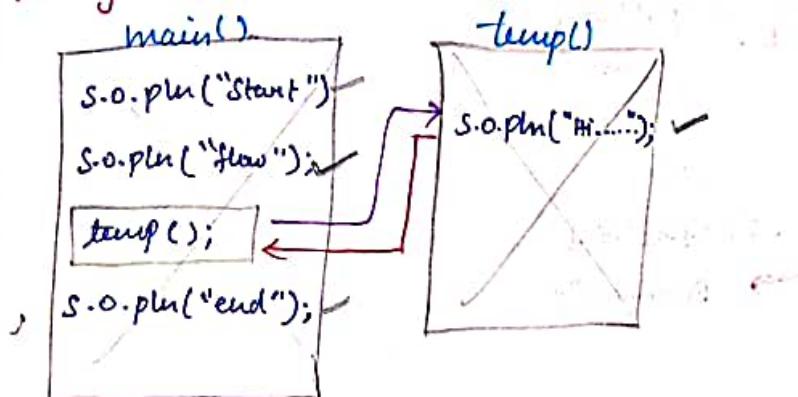
Start

flow

Hi.....

end

### Tracing :



### 3. Do tracing for the code.

```
class Demo
```

```
{
```

```
    public static void sita() {
        System.out.println("Hi..! From sita()");
    }
}
```

```
    public static void ram() {
        System.out.println("Ram Begins");
        sita();
        System.out.println("Ram Ends");
    }
}
```

```
    public static void main(String[] args) {
        System.out.println("Main Begins");
        ram();
        sita();
        System.out.println("Main Ends");
    }
}
```

### OUTPUT:

Main Begins

Ram Begins

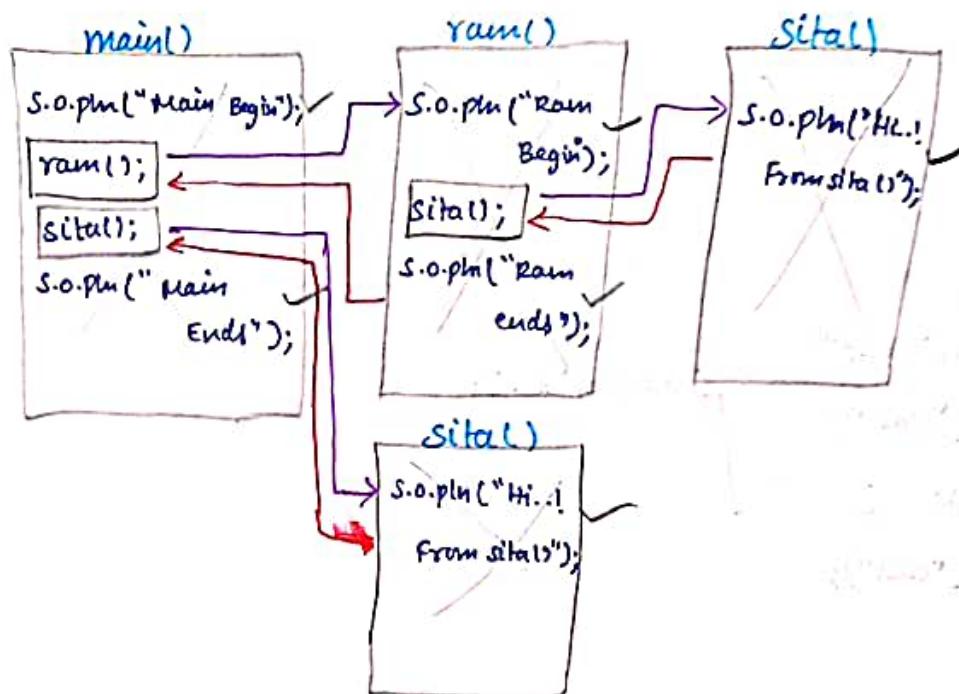
Hi..! From sita()

Ram Ends

Hi..! From sita()

Main Ends

## Tracing:



22-HAR

## Return Statement:

A method after execution will return a data back to the caller with the help of return statement.

### return:

\* return is a keyword

\* It is a control transfer statement.

\* When the return statement is executed, the execution of the method is terminated and control is transferred to the calling method.

### Steps to use return statement:

Step 1: provide a return type for a method (It should not be void)

Step 2: Use the return statement in the value to be returned.

### Rule:

The type specified as return type should be same as the type of value passed in a return statement.

## Method Overloading

If more than one method is created with the same name but different formal arguments in the same class are known as method overloading.

### Example:

java.lang.Math;

abs(double d)  
abs(float f)  
abs(int i)  
abs(long l)

} overloaded method

These are some of the overloaded method (Method with same name different formal arguments) implemented in java.lang.Math class.

### Example:

test(); → void test();  
test(10); → int test(int i);  
test(10.5f); → float test(float f);  
test(10, 10.5f); → void test(int i, float f)  
test(10.5f, 10); → void test(float f, int i)  
test('a');  
test(10, 10); // CTE

## STATIC AND STATIC MEMBERS

**STATIC:** eg: building the house.

- \* Static is a keyword.
- \* It is a modifier
- \* Any member of a class is prefixed with static modifier then it is known as static member of a class.
- \* Static members are also known as class members.

### NOTE:

Static members can be prefixed only for class members (members declared in a class).

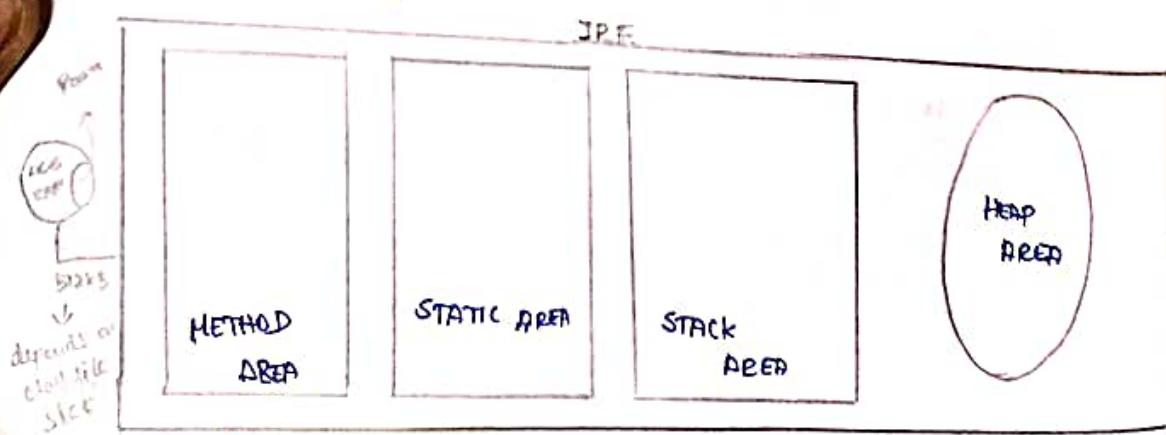
### STATIC MEMBERS:

- \* Static method
- \* static variable
- \* static initializer

### JAVA RUNTIME MEMORY

- \* To execute the java program a portion of memory in RAM is allocated for JRE
- \* In that portion of memory allocated, we have different range of memory, hence they are classified as follows,

- \* Method area
- \* class static area
- \* Stack area
- \* Heap area



### METHOD AREA:

All the methods of a class will be stored in a method area (instruction of the methods).

### CLASS STATIC AREA:

\* For every class there is a dedicated <sup>block</sup> of memory is created in the class static area.

\* The static members of the class will be allocated inside the memory created for the class.

### STACK AREA:

\* Stack area is used for execution of instruction.

\* For every method that is under execution a block of memory is created in this stack area which is known as frame.

\* Once the execution of a method is completed the frame is removed.

### HEAP AREA:

\* In a heap area a block of memory is created for the instance of class.

\* Every block of memory created with the help of reference.

\* All the non static member of a class will be allocated inside this block of memory.

\* Therefore we can access the non-static member with the help of reference.

### STATIC METHOD:

A method prefixed with static modifier is known as static method.

### CHARACTERISTICS:

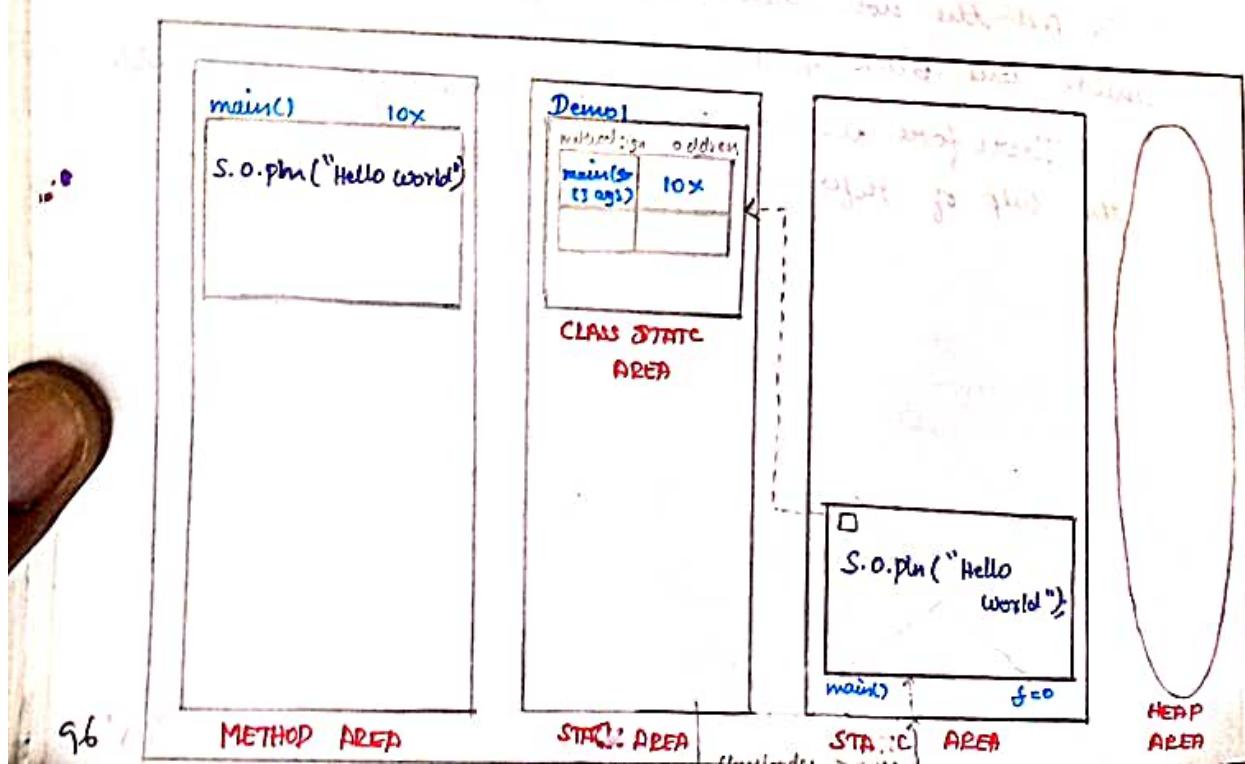
- \* static method block is stored in the method area and reference of the static method is stored inside the class static area
- \* we can use the static method with or without creating object of the class.
- \* we can use the static method with the help of class name
- \* A static method of the class can be used in any class with the help of class name.

### Example:

```
class Demo1{  
    public static void main(String[] args) {
```

```
        System.out.println("Hello world");  
    }
```

```
}
```



executing Demo2

OLP:

Hello world

Example:

class Demo2

{ public static void main (String [] args)

{

System.out.println ("Main start");

mi();

System.out.println ("Main end ");

}

public static void mi()

{

System.out.println ("MI start");

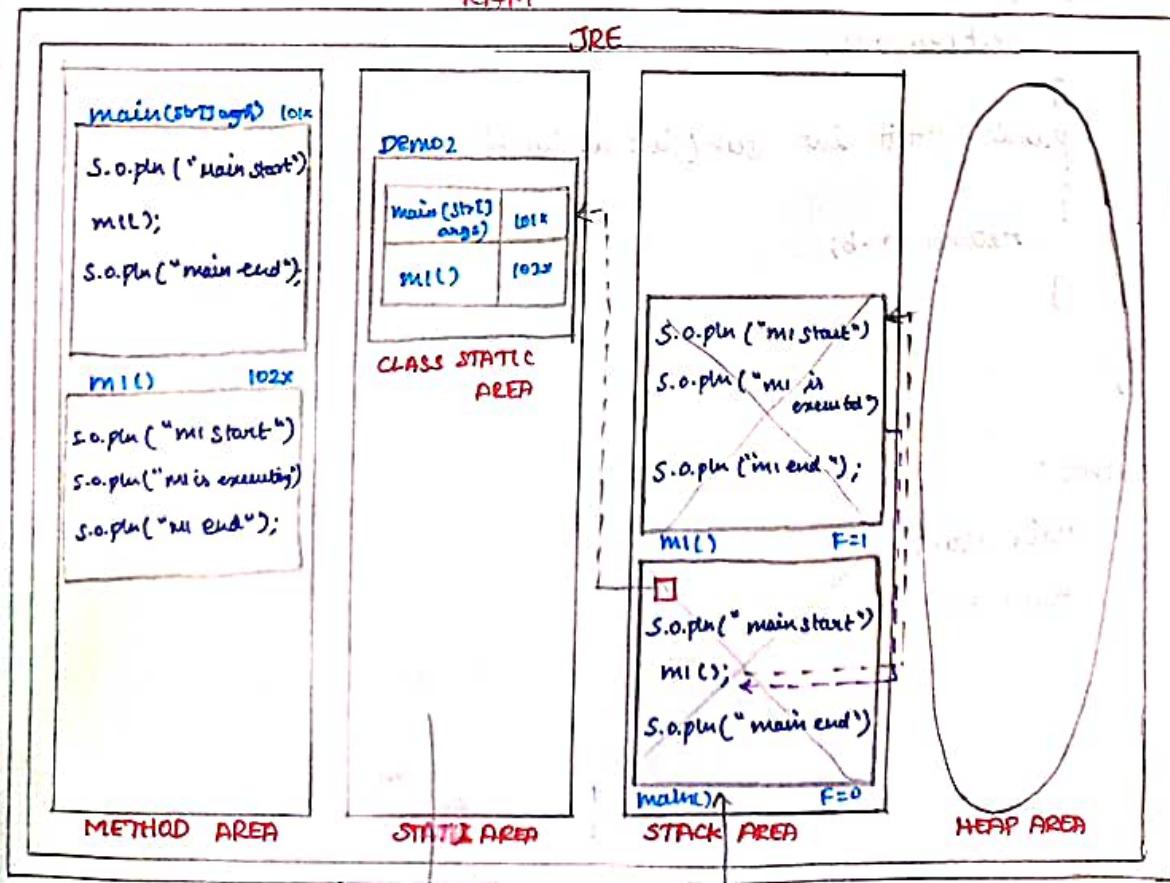
System.out.println ("MI is executing");

System.out.println ("MI end ");

}

}

RAM



### OUTPUT:

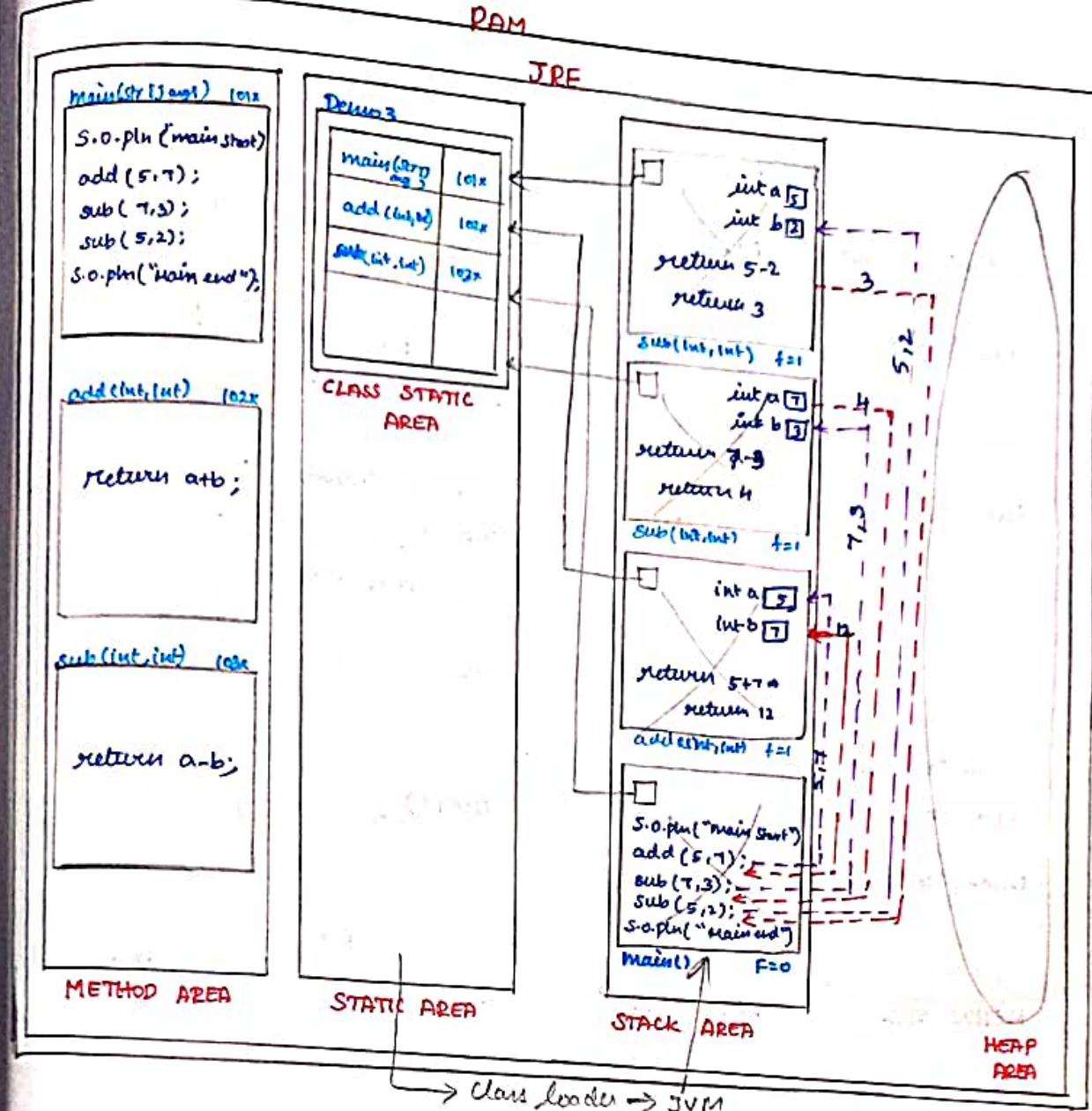
Main start  
M1 start  
M1 is executing  
M1 end  
Main end.

### Example :

```
class Demo3
{
    public static void main(String[] args)
    {
        System.out.println("Main Start");
        add(5,7);
        sub(7,3);
        sub(5,2);
        System.out.println("Main End");
    }
    public static int add(int a, int b)
    {
        return a+b;
    }
    public static int sub(int a, int b)
    {
        return a-b;
    }
}
```

### OUTPUT :

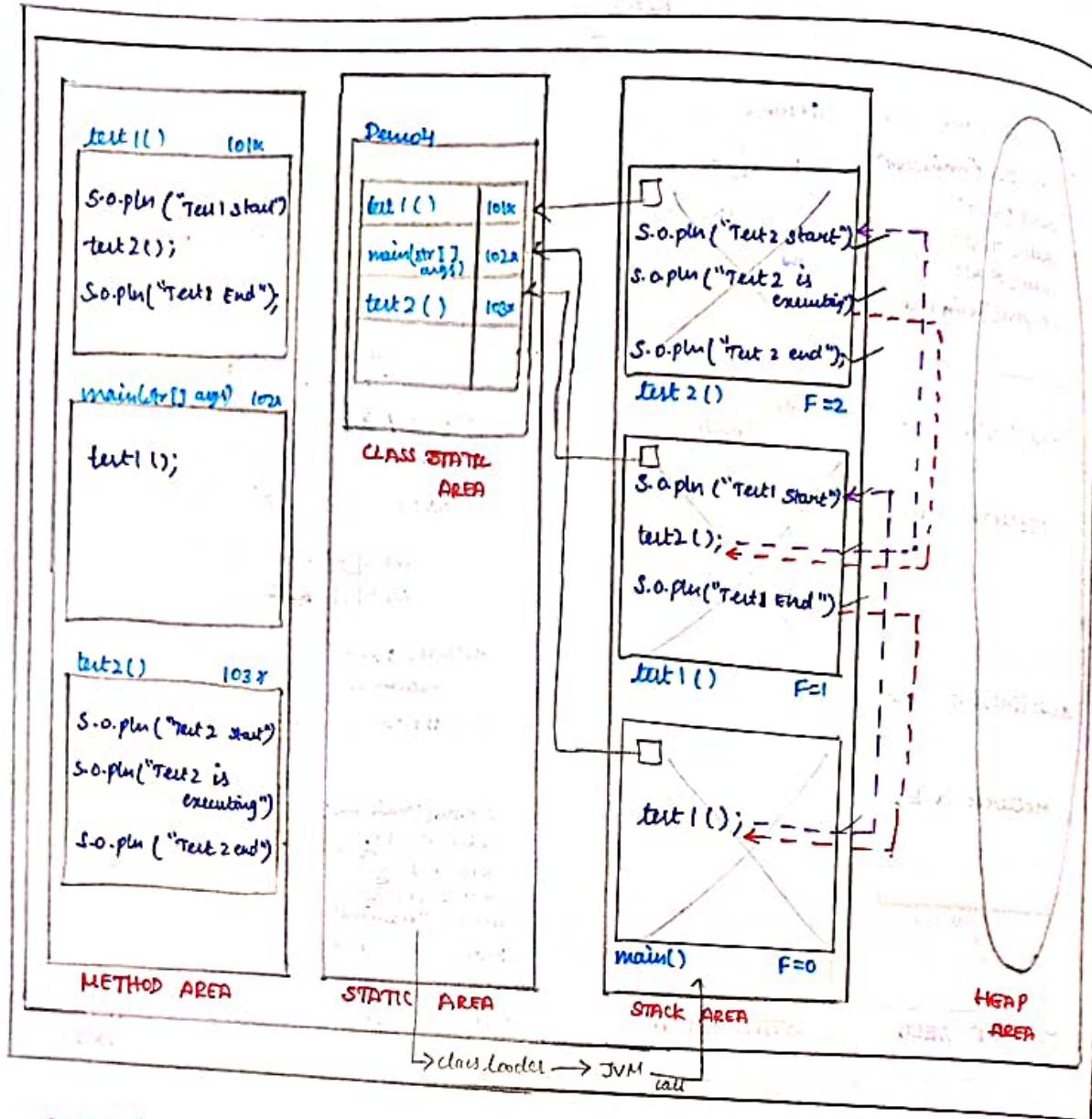
Main Start  
Main End



Example :

```

class Demo2
{
    public static void test1()
    {
        System.out.println ("Test 1 start");
        test();
        System.out.println ("Test 1 End");
    }
    public static void main(String[] args)
    {
        test1();
    }
    public static void test2()
    {
        System.out.println ("Test 2 start");
        System.out.println ("Test 2 is executing");
        System.out.println ("Test 2 End");
    }
}
    
```



OUTPUT:

Test1 start  
 Test2 start  
 Test2 is executing  
 Test2 is end  
 Test1 is end

Example:

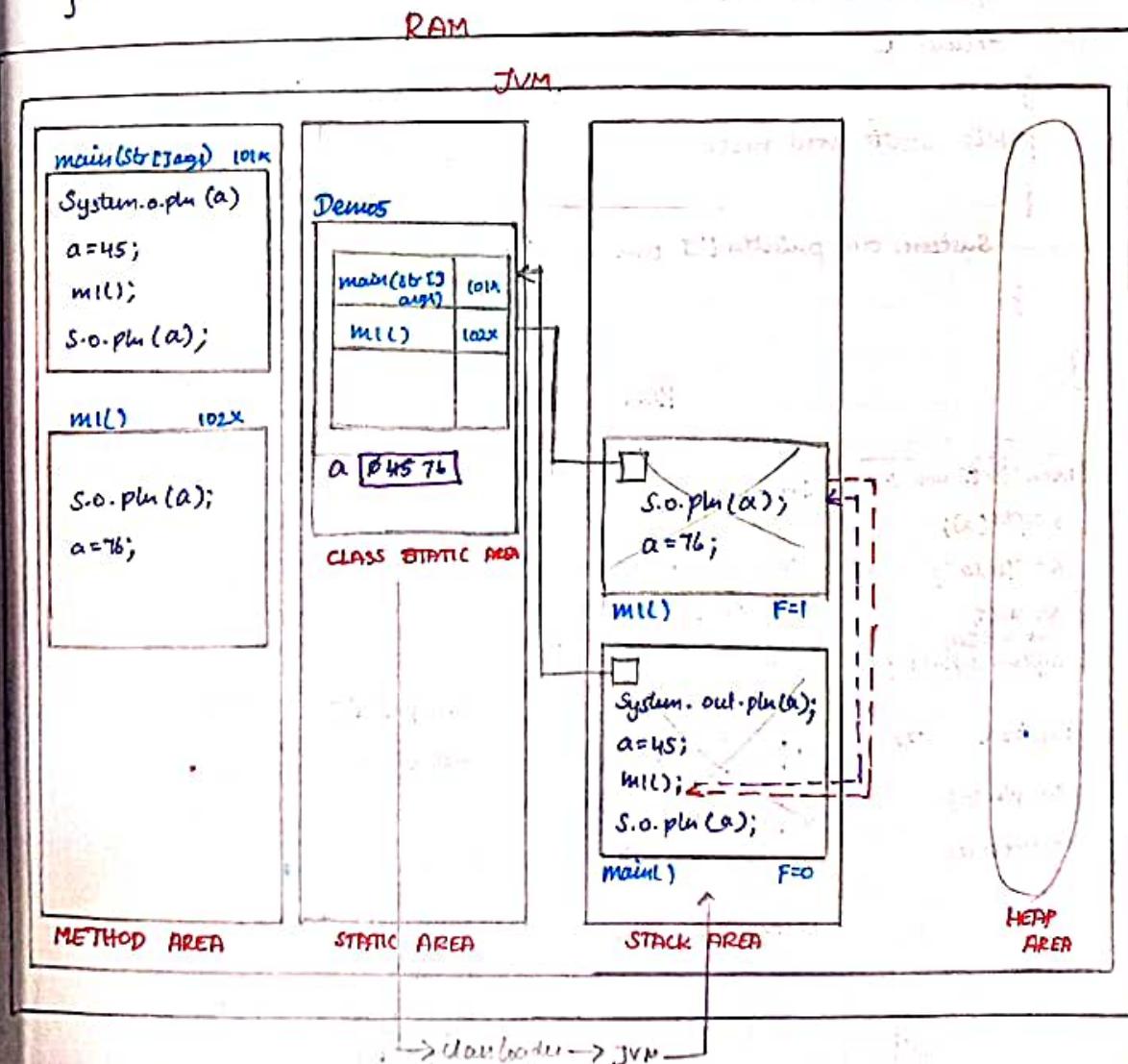
```
class Demo5
{
    static int a;
    public static void main(String[] args)
    {
        System.out.println(a);
        a=45;
        m1();
        System.out.println(a);
    }
    public static void m1()
    {
        System.out.println(a);
        a=76;
    }
}
```

OUTPUT:

0

45

76



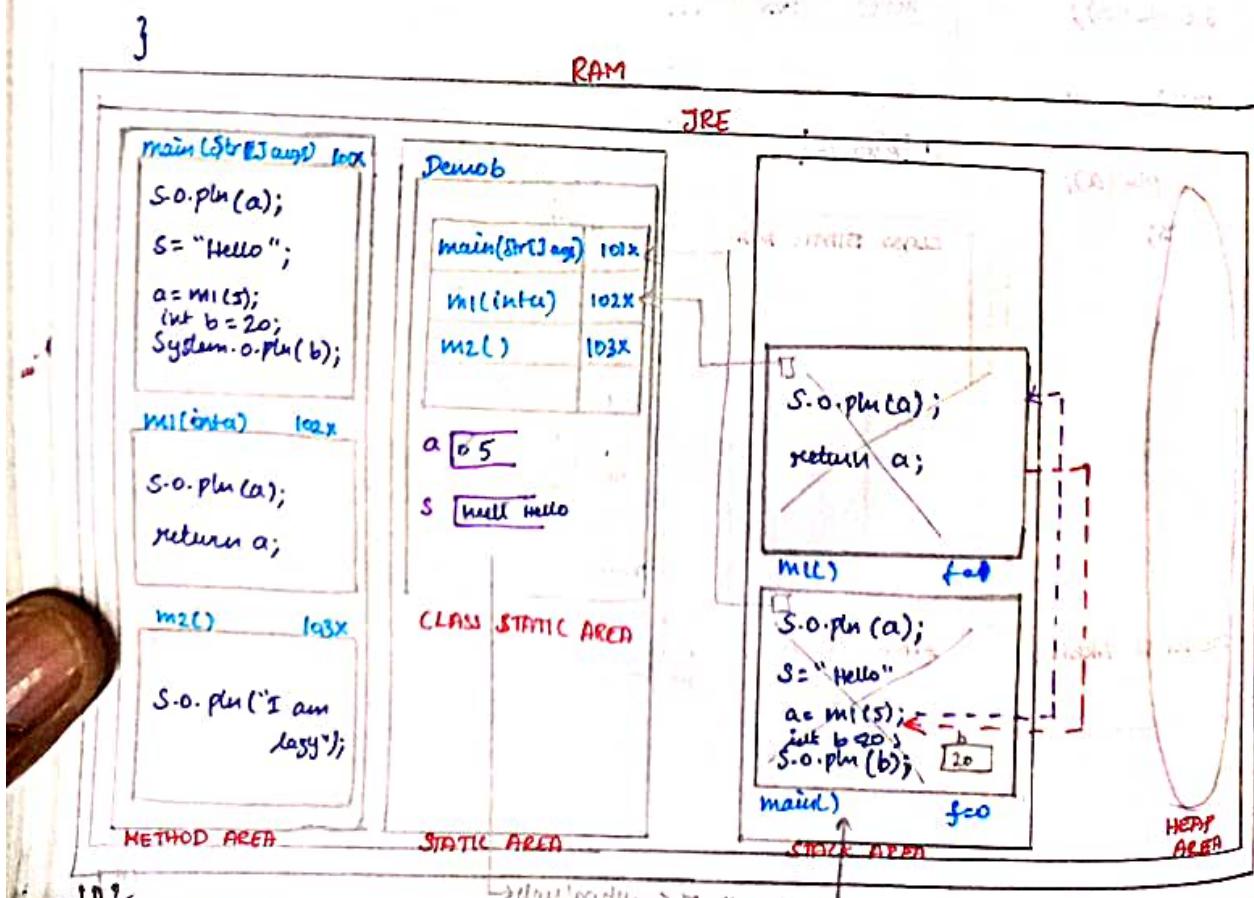
→ User Code → JVM

Example:

```
class Demob  
{  
    static int a;  
    static String s;  
    public static void main(String[] args)  
    {  
        System.out.println(a);  
        s = "Hello";  
        a = m1(5);  
        int b = 20;  
        System.out.println(b);  
    }  
    public static int m1(int a)  
    {  
        System.out.println(a);  
        return a;  
    }  
    public static void m2()  
    {  
        System.out.println("I am lazy");  
    }  
}
```

OUTPUT:

0  
5  
20



**EXAMPLE:**

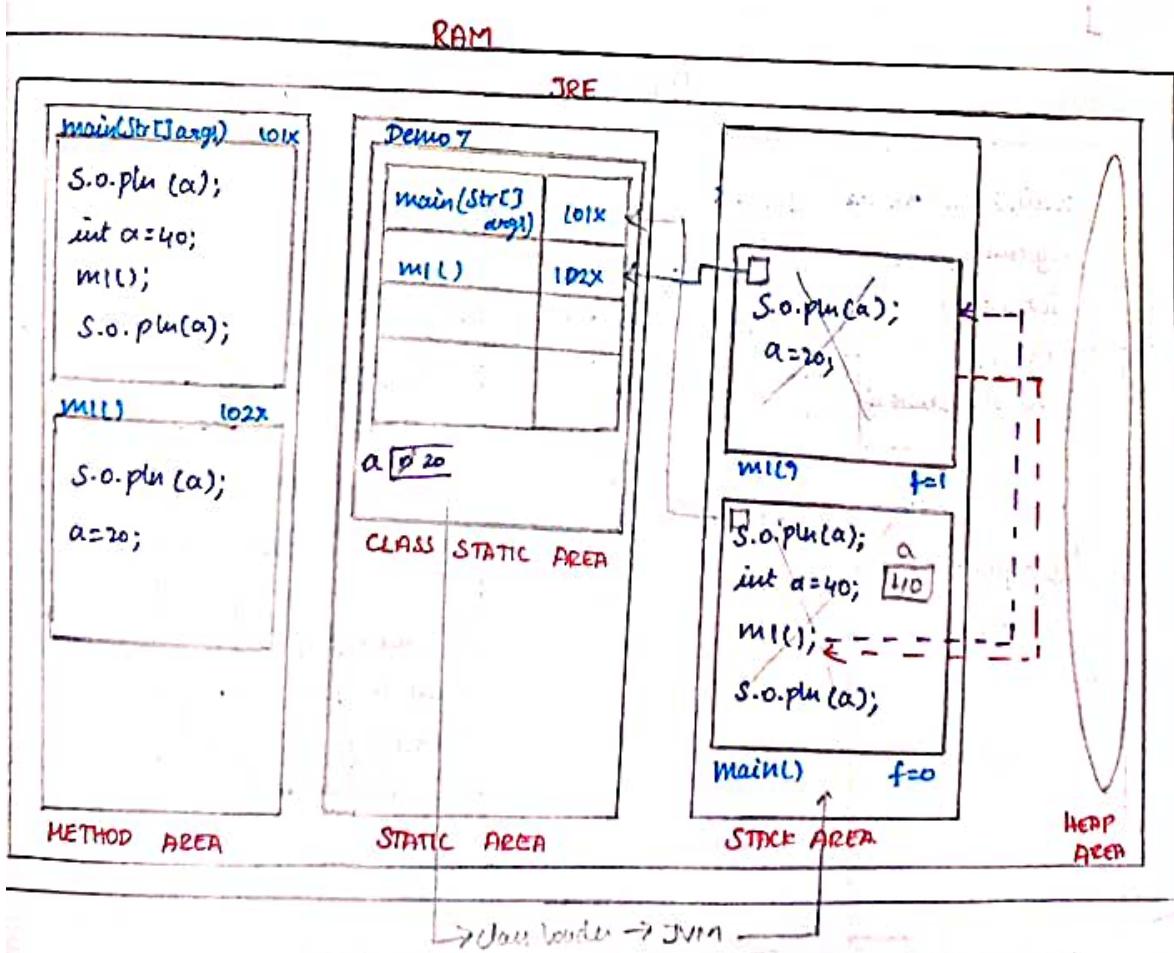
```

class Demo7
{
    static int a;
    public static void main(String [] args)
    {
        System.out.println(a);
        int a=40;
        m1();
        System.out.println(a);
    }
    public static void m1()
    {
        System.out.println(a);
        a=20;
    }
}

```

**OUTPUT:**

0  
0  
40



**EXAMPLE:**

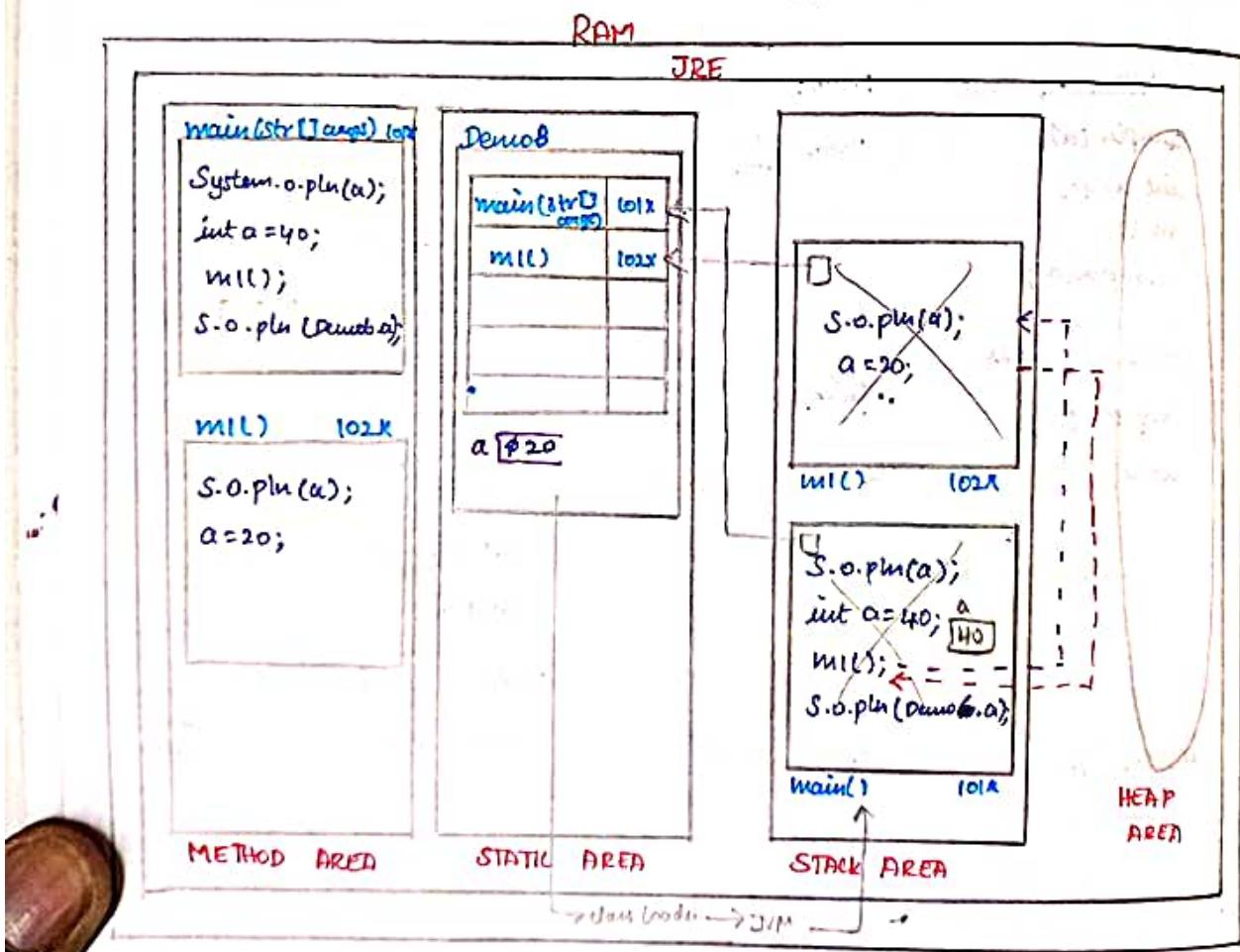
```

class Demo8
{
    static int a;
    public static void main(String [] args)
    {
        System.out.println(a);
        int a=40;
        m1();
        System.out.println(Demo8.a)
    }
    public static void m1()
    {
        System.out.println(a);
        a=20;
    }
}

```

**OUTPUT:**

0  
0  
20



### **STATIC VARIABLE:**

Variable declared in a class block and prefixed with static modifier is known as static variable.

### **CHARACTERISTICS :**

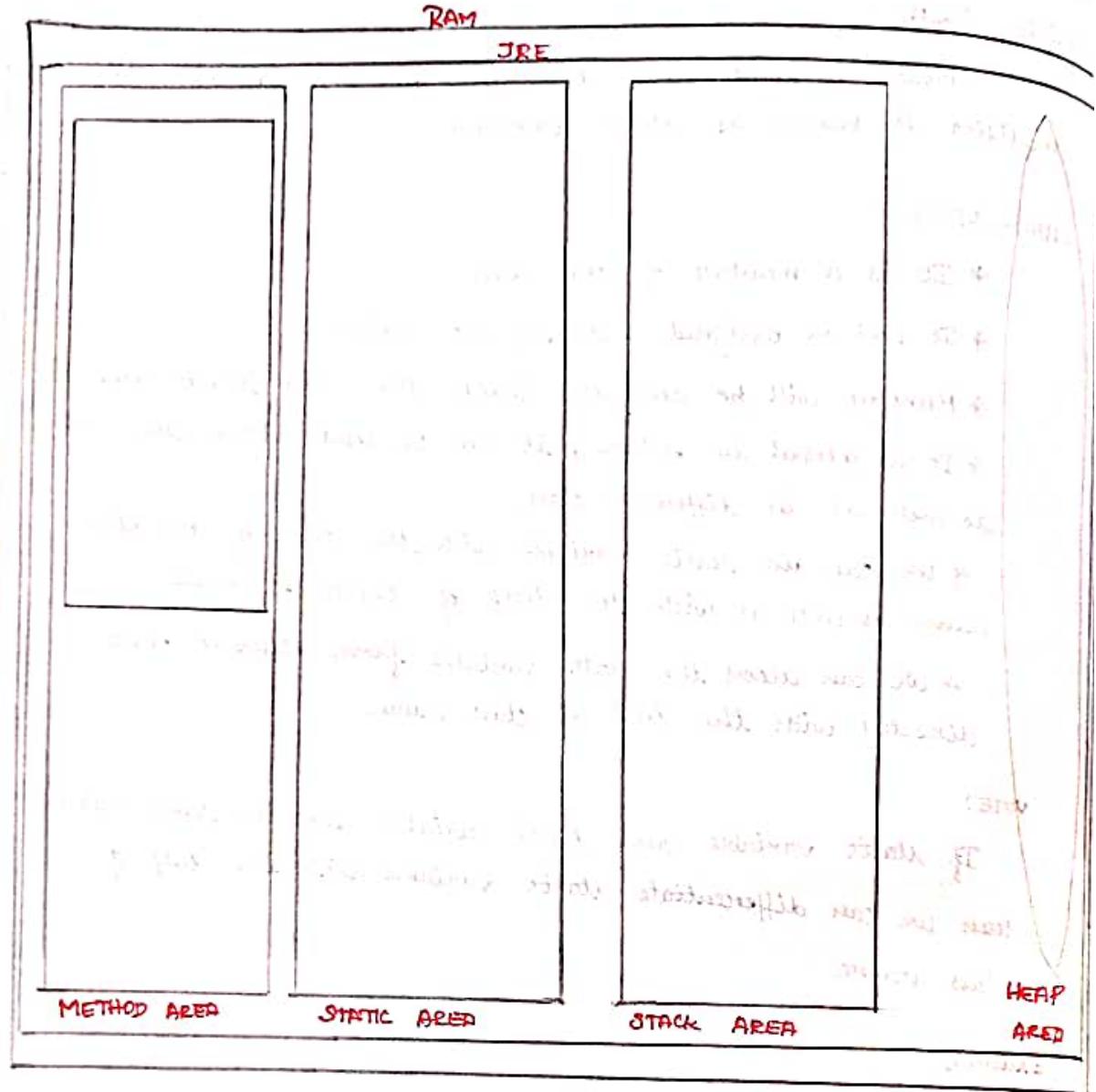
- \* It is a member of the class.
- \* It will be assigned with default value.
- \* Memory will be allocated inside the class static area.
- \* It is global in nature, it can be used within the class as well as in different class.
- \* We can use static variable with the help of the class name as well as with the help of object reference.
- \* We can access the static variable from different class directly with the help of class name.

### **NOTE:**

If static variable and local variable are in same name then we can differentiate static variable with the help of class name.

### **Example:**

```
class Demo7
{
    static int a;
    public static void main(String[] args)
    {
        int a=20;
        System.out.println(a);
        Demo7.a=70;
        m1();
        a=29;
        System.out.println(a++);
        System.out.println(a);
        System.out.println(Demo7.a);
    }
    public static void m1()
    {
        a++;
    }
}
```



13/12/22

## STATIC INITIALIZER:

We have two types of static initializers. They are,

→ Single line static initializer

→ Multi line static initializer

### Single line Initializers:

Syntax:

static datatype variable = value / Expression;

### Multiline static Initializers:

Syntax:

```
static  
{  
    //statements;  
}
```

### Characteristics:

\* Static initializers gets executed implicitly during the loading process of the class.

\* A class can have more than one static initializers they execute top to bottom order.

### Purpose of static Initializers:

\* Static initializers are used to execute the startup instructions

\* As the static blocks get executed before the actual execution

### Loading process of the class:

\* A block is created for a class in the static pool, it can be accessed with the help of class name.

\* All the method definitions are loaded in method area and if the method is static then the reference of that method is stored inside the class block.

\* If class has any static variable they are loaded in the class static area with default value

- \* If the class has any static initializers, they are executed from top to bottom order.
- \* The loading process of the class is completed then JVM will call the main method of initial loading class.

### JVM: NOTE:

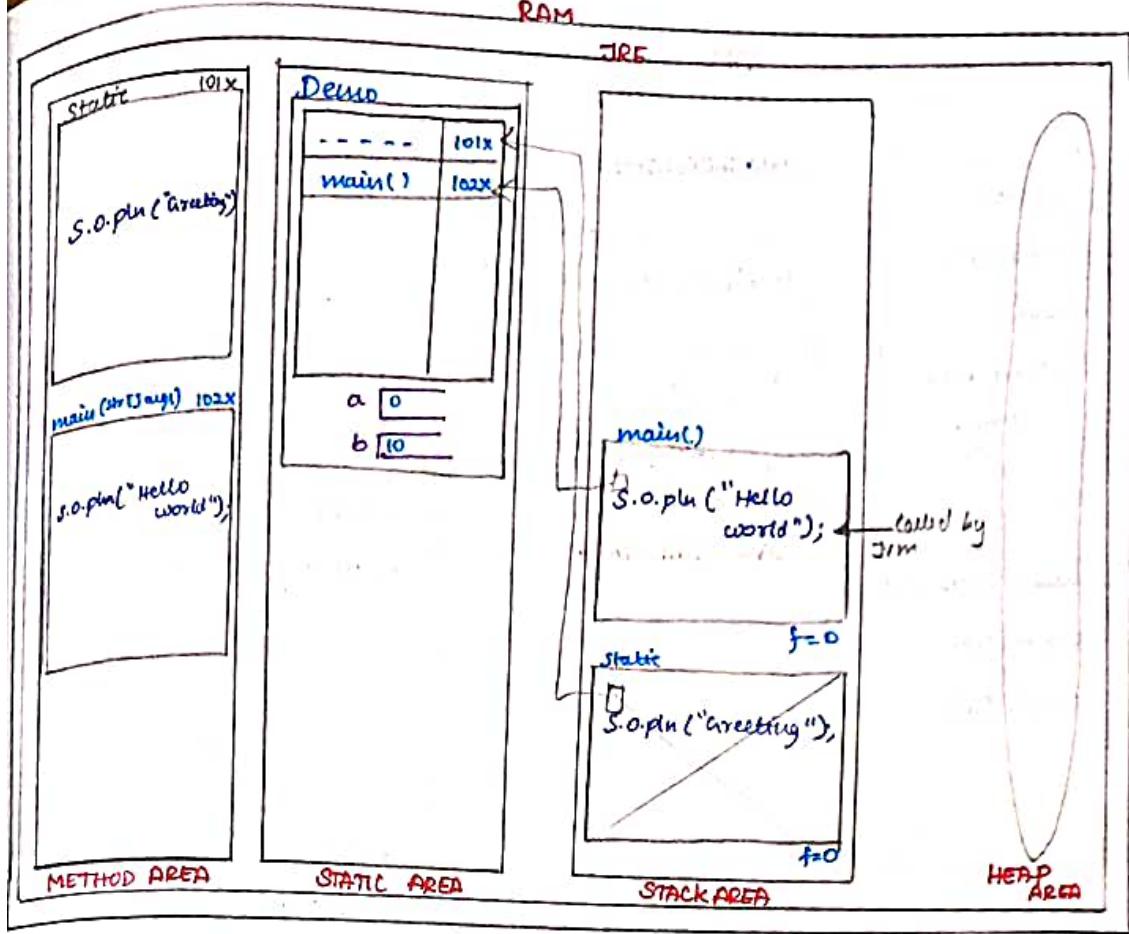
JVM will call only the main method of initial loaded class.

### Example:

```
class Demo {
    static int a;
    static int b=10;
    static {
        System.out.println("Greeting");
    }
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

### OUTPUT:

```
Greeting
Hello world
```



### EXAMPLE:

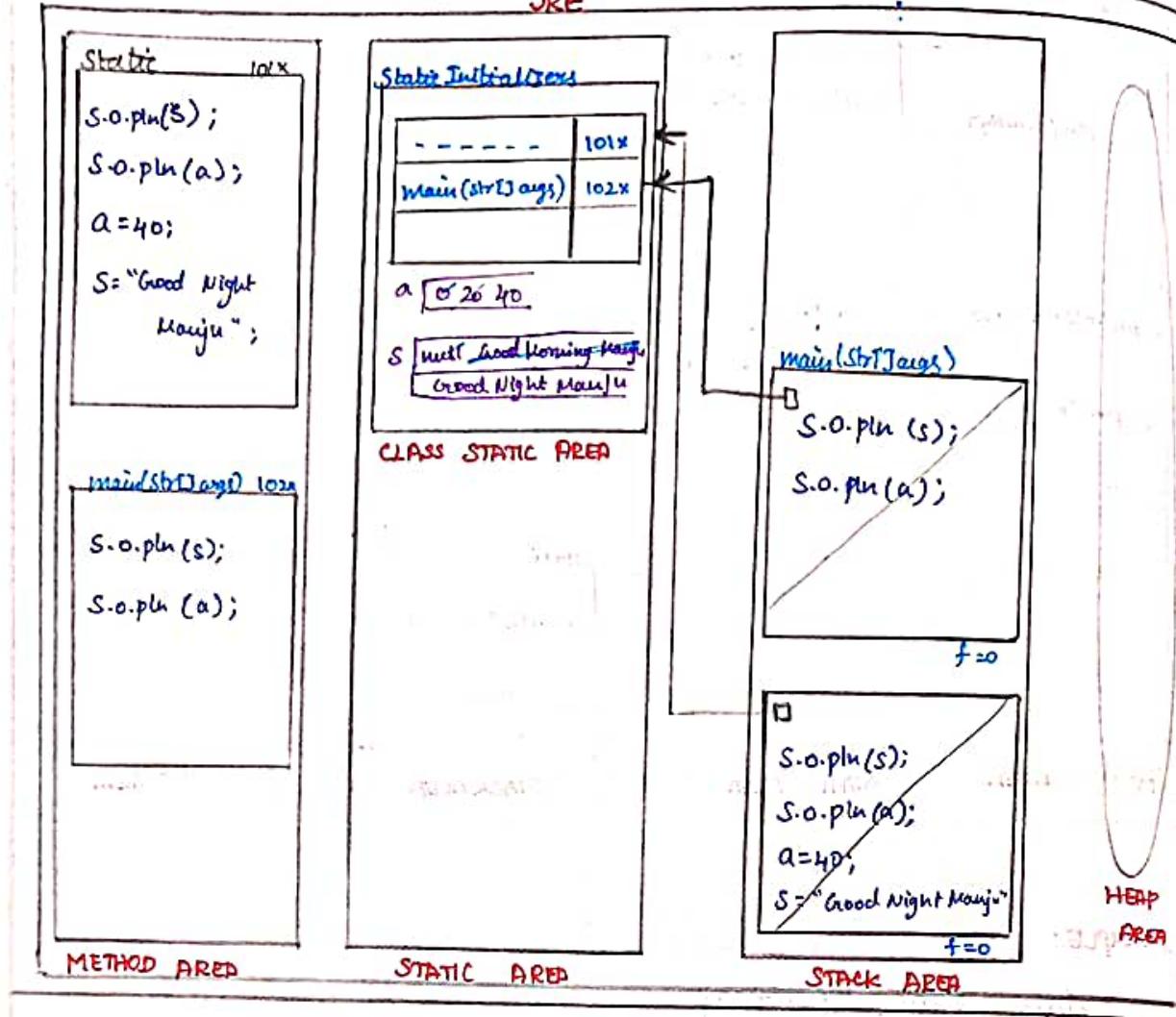
```

class StaticInitializers {
    static int a=20;
    static String s="Good Morning Manju";
    static
    {
        System.out.println(s);
        System.out.println(a);
        a=40;
        s="Good Night Manju";
    }
    public static void main(String[] args)
    {
        System.out.println(s);
        System.out.println(a);
    }
}

```

**OUTPUT:**

Good Morning Manju  
20  
Good Night Manju  
40

**EXAMPLE :**

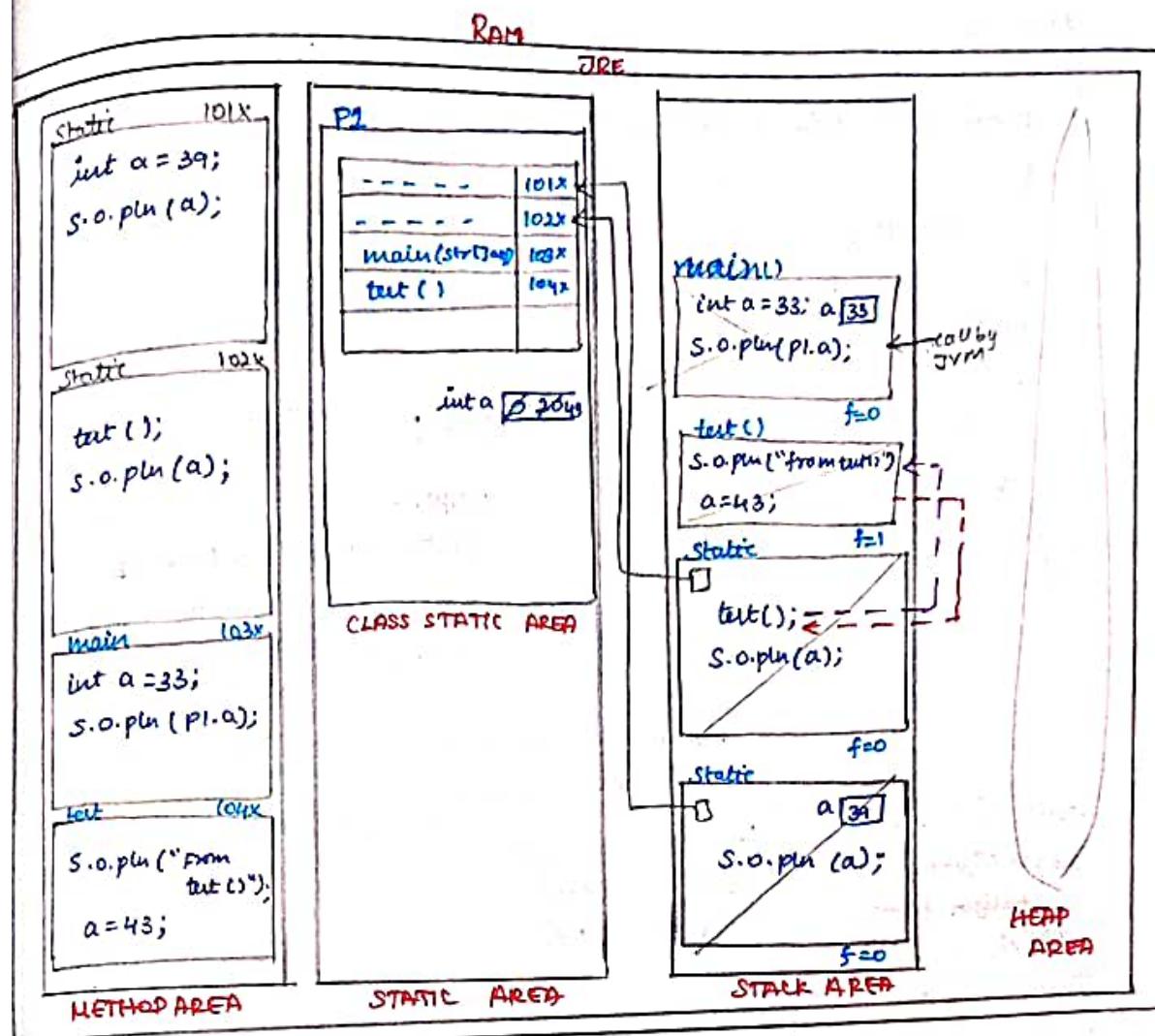
```

class P1
{
    static {
        int a=39;
        System.out.println(a);
    }
    static int a=20;
    public static void main (String [] args)
    {
        int a = 33;
        System.out.println (P1.a);
    }
    public static void test ()
    {
        System.out.println ("From test()");
        a=43;
    }
    static {
        test();
        System.out.println (a);
    }
}

```

**OUTPUT:**

39  
From test()  
43  
43



Example :

```

class C1
{
    static
    {
        System.out.println("static initializer from c1");
    }
    public static void test()
    {
        System.out.println("test () from c1");
    }
    public static void main(String[] args)
    {
        System.out.println("Hello world");
    }
}

```

```

class c2
{
    public static void main(String[] args)
    {
        c1.text();
    }
    static
    {
        System.out.println("Static initializer from c2");
    }
}

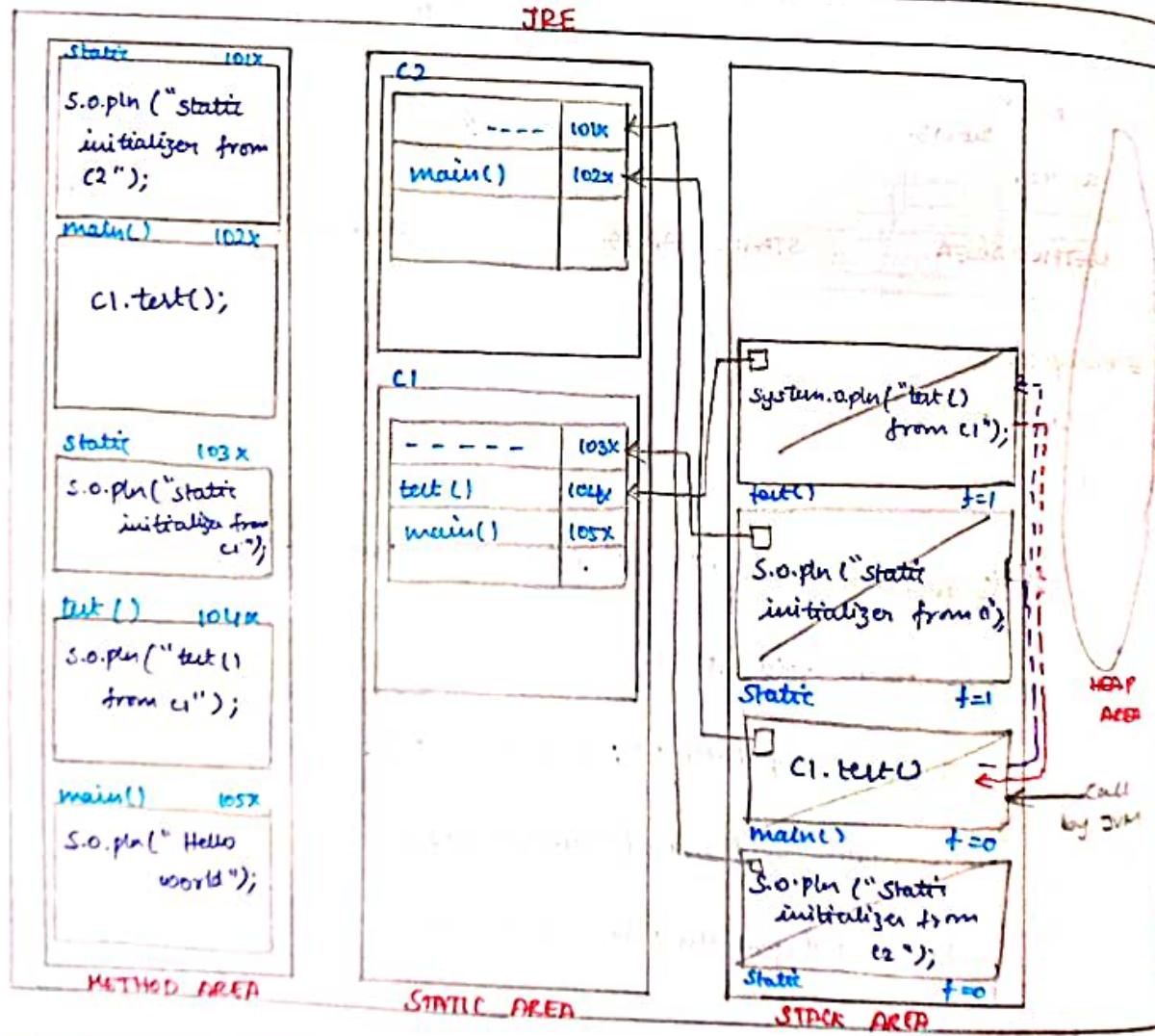
```

**Output :**

Static initializer from c2  
 Static initializer from c1  
 text() from c1

RAM.

JRE



```
class A {  
    static int num = 40;  
    public static void m1 {  
        System.out.println(num);  
        num = 29;  
    }  
    static {  
        System.out.println("Hello from A");  
        m1();  
    }  
    static {  
        System.out.println("Hello from B");  
        System.out.println(num);  
    }  
}
```

```
class B {  
    static int num = 40;  
    public static void main(String[] args) {  
        System.out.println("Main Start");  
        System.out.println(A.num);  
        System.out.println(num);  
    }  
}
```

#### Output:

Main Start.

Hello from A

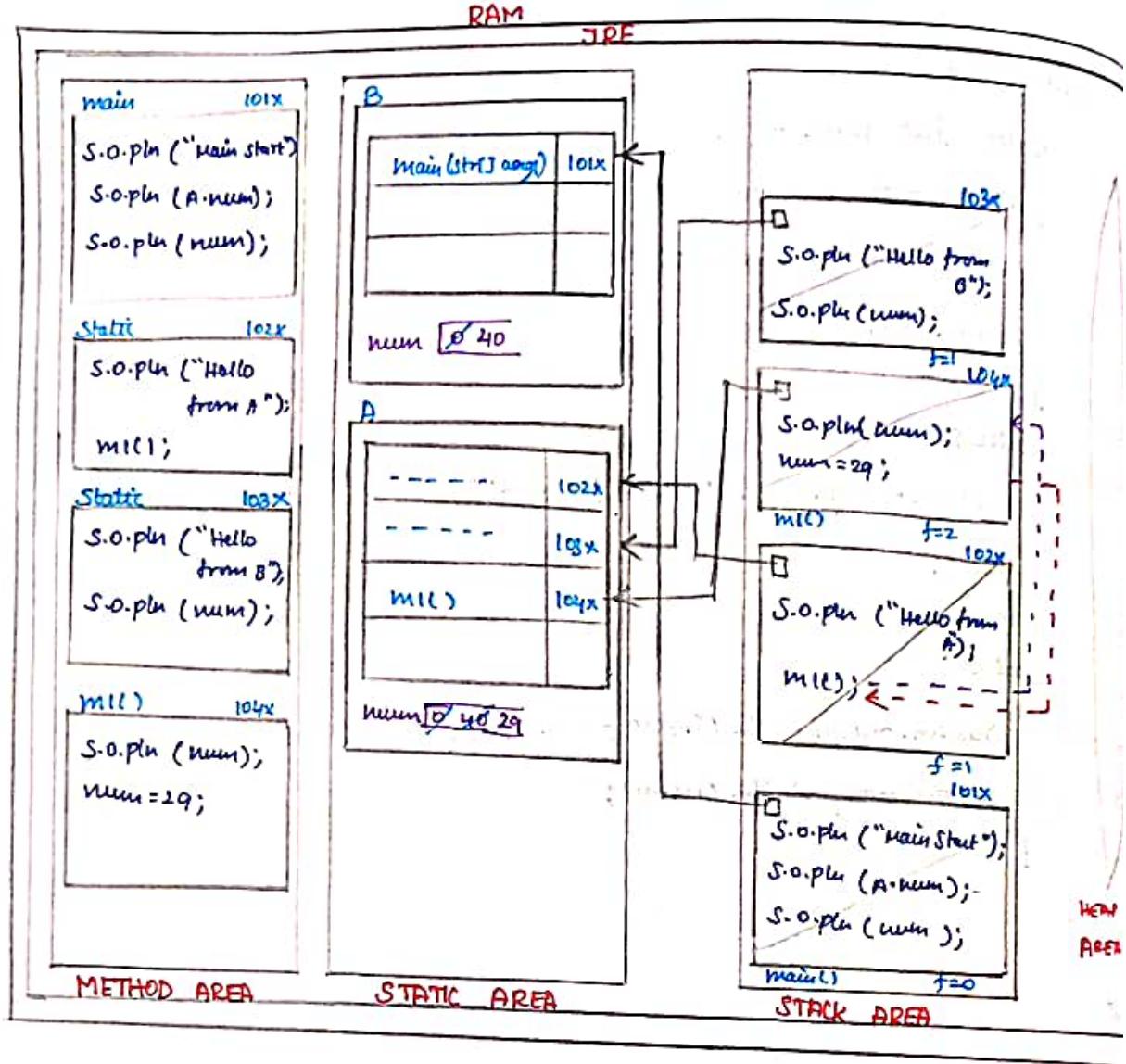
40

Hello from B.

29

29

40



प्रगति में दिया गया

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

जो भी व्यापारिक प्रैग्य के लिए जाना जाता है

**OBJECT:**

\* Any substance which has existence in the real world is known as an object.

\* Every object will have attributes and behaviours.

**OBJECT IN JAVA:**

\* According to object oriented programming object is a block of memory created in the heap area during the runtime, it represents a real world object.

\* A real world object consists of attributes and behaviour.

\* Attributes are represented with the help of non-static variables.

\* Behaviours are represented with the help of non-static methods.

**CLASS STATEMENT****CLASS****CLASS:**

\* According to real world situation before constructing an object blueprint of the object must be designed, it provides specification of the real world object.

\* Similarly in object oriented programming before creating an object the blueprint of the object must be designed which provides the specification of the object, this is done with the help of class.

**DEFINITION OF CLASS:**

\* It is user defined non primitive data type, it represents the blueprint of the real world object.

\* class provides specification of real world object.

**NOTE:**

we can create any number of object. For a class, it is known as instance of a class.

## NON STATIC :

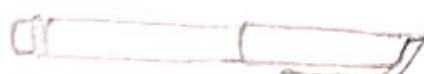
- \* Any member declared in a class and not prefixed with a static modifier is known as a non-static member of a class.
- \* Non-static members belong to an instance of a class. Hence it is also known as an instance member or object member.
- \* The memory for the non-static variable is allocated inside the heap area (instance of a class).
- \* we can create any number of instance for a class
- \* Non-static members will be allocated in every instance of a class.

## NON STATIC MEMBERS :

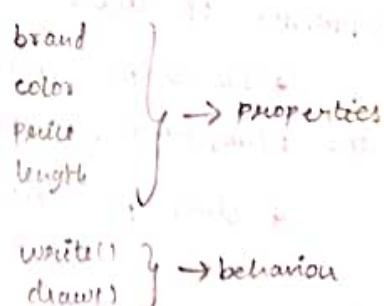
- \* Non static variable
- \* Non static method
- \* Non static initializer
- \* constructors.

## DESIGNING AN OBJECT :

Example :



```
class Pen
{
    String brand;
    String color;
    double price;
    double length;
}
```





```

class Bottle
{
    String brand;
    String color;
    double price;
    int capacity;
    String shape;
}

```

brand  
 color  
 price  
 capacity  
 shape } → properties  
 represented by non static variable  
 drunk()  
 full() } → behaviour  
 represented by non static method

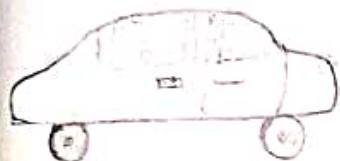


```

class Book
{
    String title;
    int noOfPages;
    String author;
    double price;
}

```

title  
 noOfPages  
 author  
 price } → properties  
 read()  
 return() } → behaviour



```

class Car
{
    String brand;
    String color;
    double price;
    int milage;
    String model;
}

```

brand  
 color  
 price  
 milage  
 model } → properties  
 drive()  
 travel() } → behaviour



```

class Chair
{
    double height;
    String color;
    String typeOfMaterial;
}

```

height  
 color  
 typeOfMaterial } → properties  
 sit()  
 stand() } → behaviour



```

class Bag
{
    int price;
    String brand;
    String color;
    String Material;
    int capacity;
    int warranty;
}

```

price  
 brand  
 color  
 Material  
 capacity  
 warranty } → properties  
 store()  
 carry() } → behaviour

## STEP TO CREATE AN OBJECT:

Step 1: Create a class or use an existing class if already created.

## Step 2: Instantiation.

### INstantiation:

The process of creating an object is known as instantiation.

Syntax to create an object:

```
new className();
```

### NEW:

\* new is a keyword.

\* It is a unary operator.

\* It is used to create a block of memory inside a heap area during execution.

\* Once the object is created it returns the reference of an object.

### CONSTRUCTOR:

constructor is a special member of the class whose name is same as the class name.

constructor is used to load the non static members of a class into the object.

### Example:

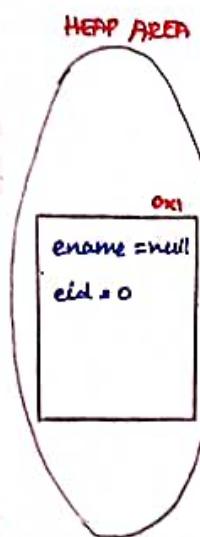
#### Step 1: Designing a class

```
class Employee  
{  
    String ename;  
    int eid;
```

```
}
```

#### Step 2: Instantiation

```
new Employee();
```



## NON-PRIMITIVE DATA TYPE:

Every class name in java is a non primitive type.

Non primitive data type are used.

Example:

```
class Employee
{
    String ename;
    int eid;
}

class EmployeeDriver
{
    public static void main (String [] args)
    {
        Employee e = new Employee ();
        System.out.println (e);
    }
}
```

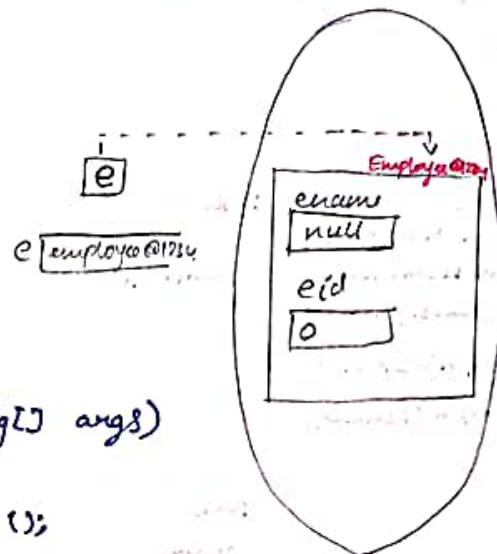
Output :

Employee@ 1234

Example:

```
class Book {
    String title;
    int noOfPage;
    String author;
    double price;
}

class BookDetails {
    public static void main (String [] args) {
        Book b1 = new Book ();
        Book b2 = new Book ();
        b1.title = "Java";
        b1.noOfPages = 800;
        b1.author = "Lava";
        b1.price = 1200;
        b2.title = "code Easy";
        b2.noOfPages = 300;
    }
}
```



```

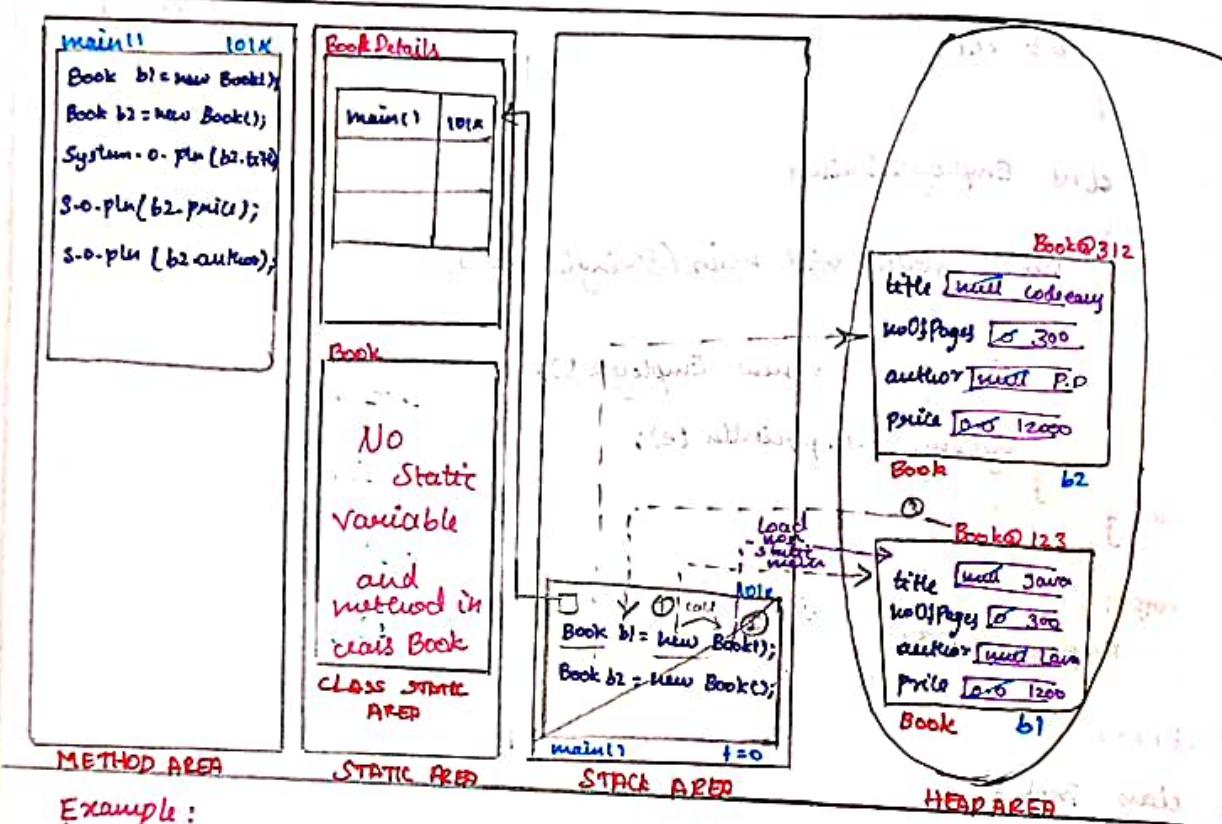
b2.author = "P.D";
b2.price = 12000;
System.out.println(b2.title);
System.out.println(b2.price);
System.out.println(b2.author);

```

3

3

OUTPUT:  
Code easy  
12000  
P.D



Example:

```

class Employee
{
    String ename;
    String eid;
    double sal;
}

```

```

class EmployeeDriver
{

```

```

public static void main(String[] args)
{
    Employee e1 = new Employee();
    Employee e2 = new Employee();
}

```

```
Employee e3 = new Employee();
```

```
    e1.ename = "Chingond";
```

```
    e1.eid = "AJAEBO2";
```

```
    e1.sal = 1500d;
```

```
Employee e2 = new Employee();
```

```
    e2.ename = "Raja";
```

```
    e2.eid = "AJAEBS420";
```

```
    e2.sal = 4800;
```

```
Employee e3 = new Employee();
```

```
    e3.ename = "Sowmya";
```

```
    e3.eid = "AJAEBS05";
```

```
    e3.sal = 3000;
```

```
System.out.println(e1.ename);
```

```
System.out.println(e1.eid);
```

```
System.out.println(e1.sal);
```

```
}
```

```
}
```

OUTPUT:

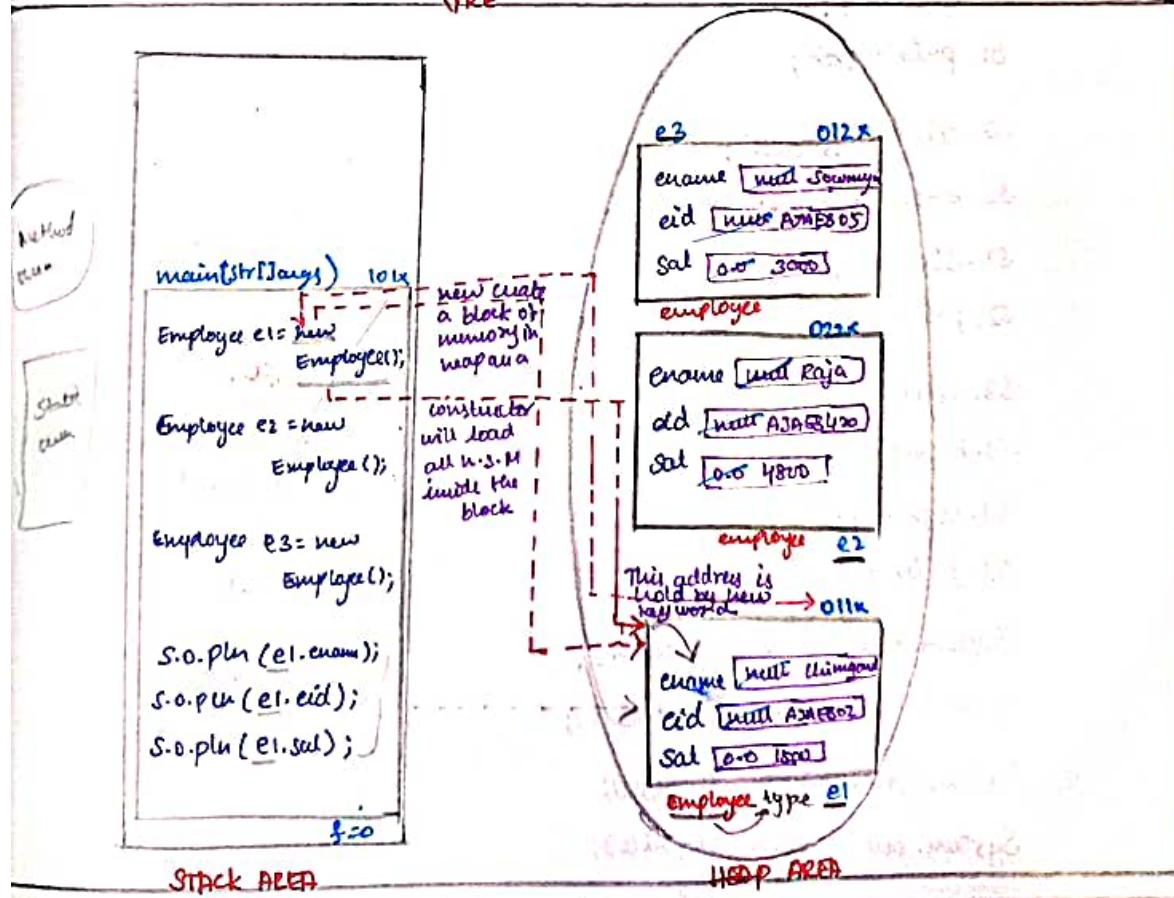
Chingond

AJAEBO2

1500.0

RAM

JRE



Example:

class Shoe

{

String color;

String brand;

int size;

double price;

}



class ShoeDriver

{

public static void main(String[] args)

{

Shoe s1 = new Shoe();

Shoe s2 = new Shoe();

Shoe s3 = new Shoe();

s1.color = "Black";

s1.brand = "Bata";

s1.size = 9;

s1.price = 1800;

s2.color = "Brown";

s2.brand = "woodland";

s2.size = 8;

s2.price = 2500;

s3.color = "white";

s3.brand = "puma";

s3.size = 7;

s3.price = 3000;

System.out.println(s2.color);

System.out.println(s2.brand);

System.out.println(s1.size);

System.out.println(s3.price);

OUTPUT:

Brown

woodland

8

2500.0

white

puma

7

3000.0

1800.0

3

122

## NON STATIC VARIABLE:

A variable declared inside a class block and not prefixed with a static modifier is known as non-static variable.

### Characteristics :

- \* we can't use the non-static variable without creating an object.
- \* we can only use the non-static variable with the help of object reference.
- \* Non-static variable are assigned with default during the object loading process.
- \* Multiple copies of non-static variables will be created (one for every object).

### Example :

```
class Slipper
{
    int size;
    String brand;
    public void slipperDetails()
    {
        System.out.println(size);
        System.out.println(brand);
    }
}
class SlipperDriver
{
    public static void main(String[] args)
    {
        Slipper slipper1 = new Slipper();
        Slipper slipper2 = new Slipper();
        slipper1.size = 8;
        slipper1.brand = "PUMA";
        slipper2.size = 9;
        slipper1.slipperDetails();
        slipper2.slipperDetails();
    }
}
```

### OUTPUT :

8

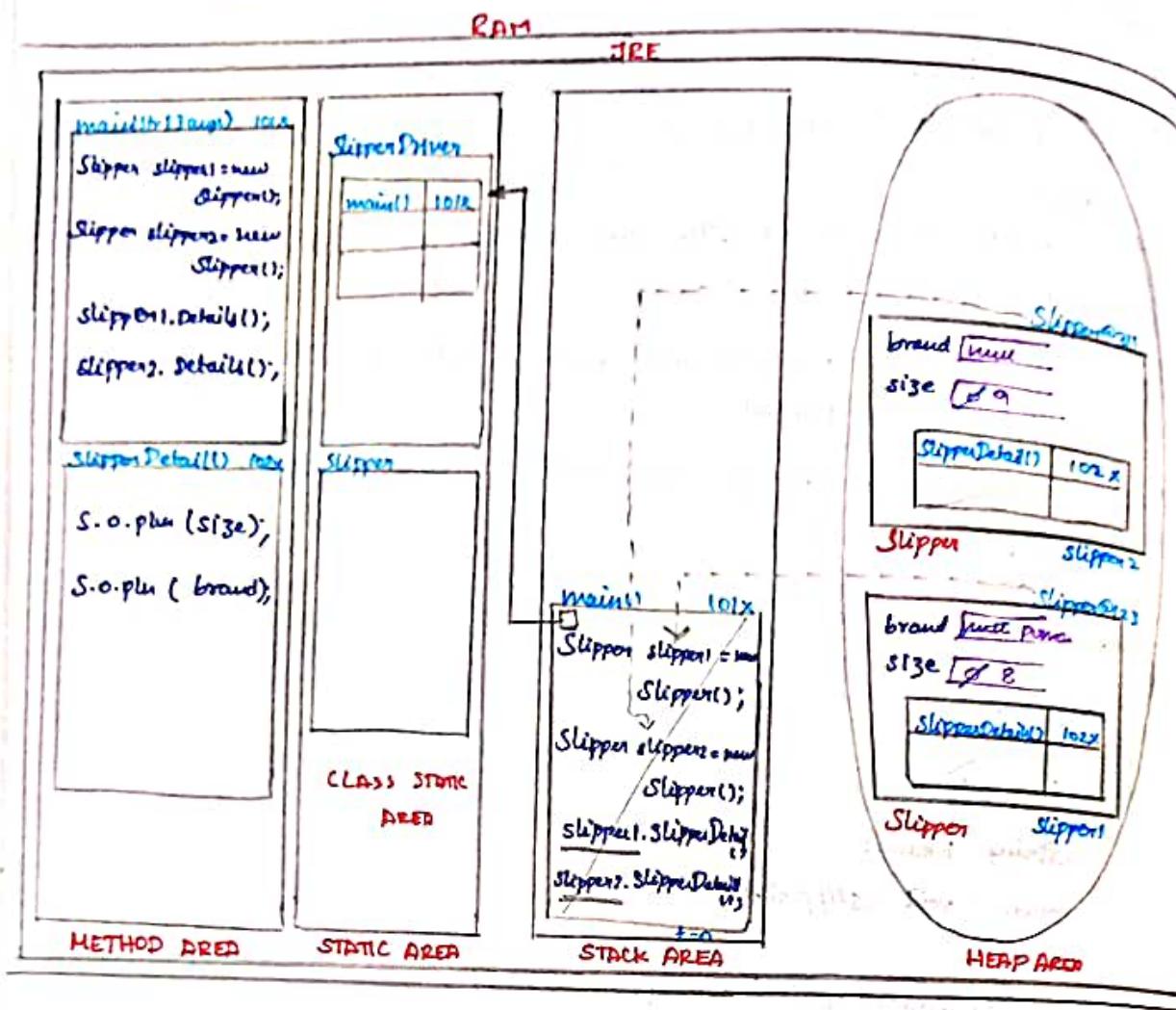
PUMA

9

NULL

**OUTPUT :**

8  
puma  
9  
null



**Non Static Method:**

A method declared in a class block and not prefixed with a static modifier is known as a non-static method.

**Characteristics:**

- \* A method block will get stored inside the method area and a reference of the method is stored inside the instance of a class [object]

- \* We can't call the non-static method of a class without creating an instance of a class [object]

- \* We can't access the non-static method directly with the help of class names.

- \* Non-static method can't be accessed directly with their names inside the static context

Example:

class Shoes

{

    String brand;

    String color;

    int size;

    double price;

    public void setProperty(String brand, String color, int size, double price);

{

    Local variable:

        This.brand = brand;  
        This.color = color;  
        This.size = size;  
        This.price = price;

    Local  
    variable

}

    public void shoesDetails()

{

        System.out.println("The shoe brand is " + brand);

        System.out.println("The shoe color is " + color);

        System.out.println("The shoe size is " + size);

        System.out.println("The shoe price is " + price);

}

}

class ShoesDriver

{

    public static void main(String[] args)

{

        Shoes s1 = new Shoes();

        s1.setProperty("Bata", "Black", 9, 1800);

        s1.shoesDetails();

}

}

OUTPUT:

The shoe brand is Bata

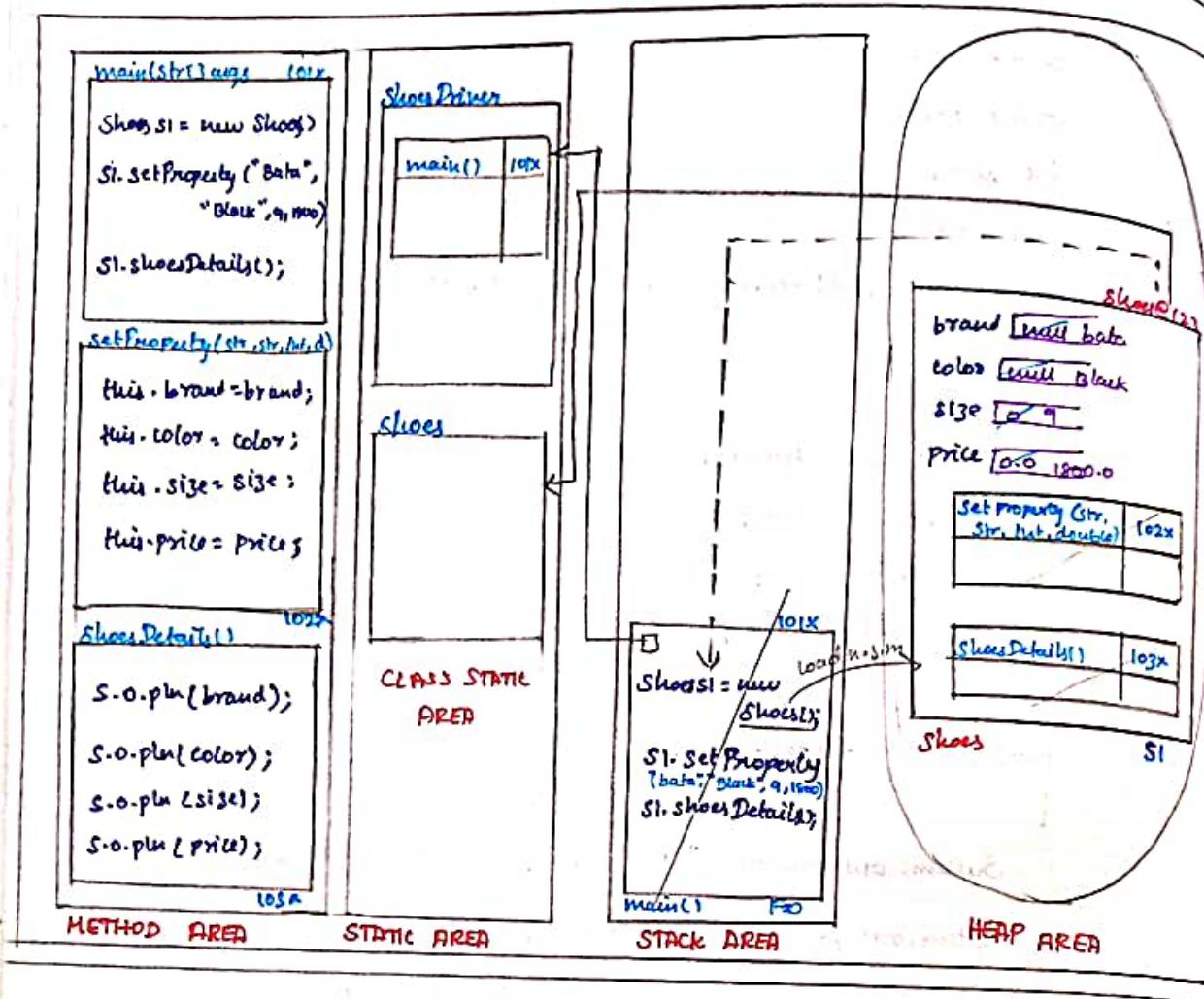
The shoe color is Black

The shoe size is 9

The shoe price is 1800.0

RAM

JRE



### NON STATIC INITIALIZERS :

Non static initializers will execute during the loading process of an object.

Non - static initializers will execute once for every instance of a class created. (Objects created).

### Purpose of non - static initializers :

Non - static initializers are used to execute the startup instructions for an object.

### Types of non - static initializers:

- \* Single line non - static initializers
- \* Multi line non - static initializers

## Single Line Non static initializer:

Syntax:

datatype variable = value/reference

## Multi Line Non static initializer:

Syntax:

```
{  
    //Statement  
}
```

### NOTE :

All the non-static initializers will execute from top to bottom order for every object creation.

## THIS KEYWORD

### This:

It is a keyword.

It is a non static variable it holds the reference of current executing object.

### USES OF this:

- \* Used to access the members of current object.
- \* It is used to give the reference of the current object.
- \* Reference of a current object can be passed from the method using this keyword.
- \* Calling a constructor of the same class is achieved with the help of this called statement.

### NON STATIC CONTEXT :

\* The block which belongs to the non static method and multi line non static initializer is known as non static context.

\* Inside a non static context we can use static and non static members of the same class directly by using its name.

### Example :

```

class NonStaticContext
{
    static int b = 20;
    int a = 40;
}

System.out.println(a); → Multiline Non static
b = 69;           Initializer is here

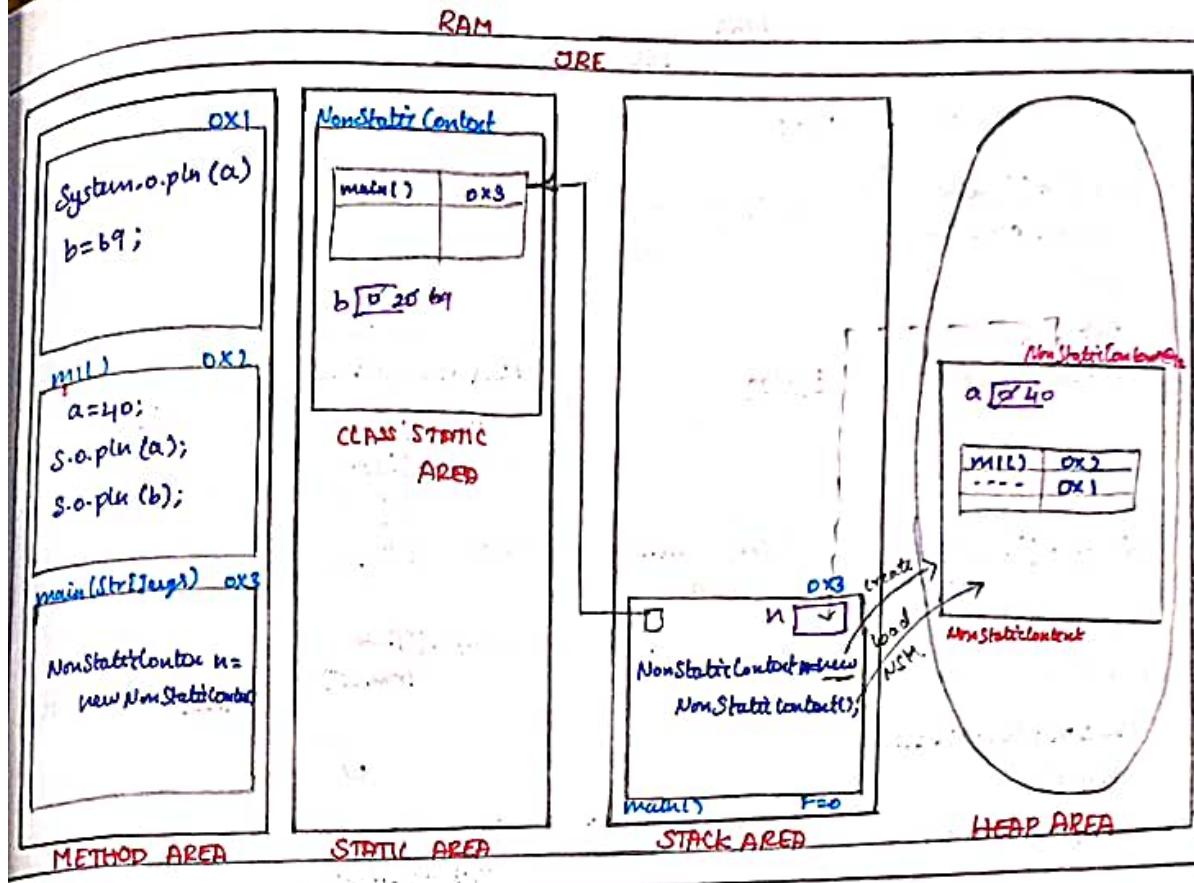
public void m1()
{
    a = 40;
    System.out.println(a);
    System.out.println(b);
}

public static void main(String[] args)
{
    NonStaticContext n = new NonStaticContext();
}

```

### OUTPUT:

40



Example :

```

class Nsc1
{
    {
        System.out.println("Hello from NSI"); } → Multi-line NSI
    }

    static
    {
        int a = 40; } → Multi-line SI
        System.out.println(a+50);

    }

    int b = 367; // Single line SI
}

static
{
    System.out.println("Hello from SI"); } → Multi-line SI
}

static int i=29; // Single line SI
public static void main (String [J args)
{
    Nsc1 n = new Nsc1();
    n.b = 30;
    System.out.println(i+p.b);
}
}

```

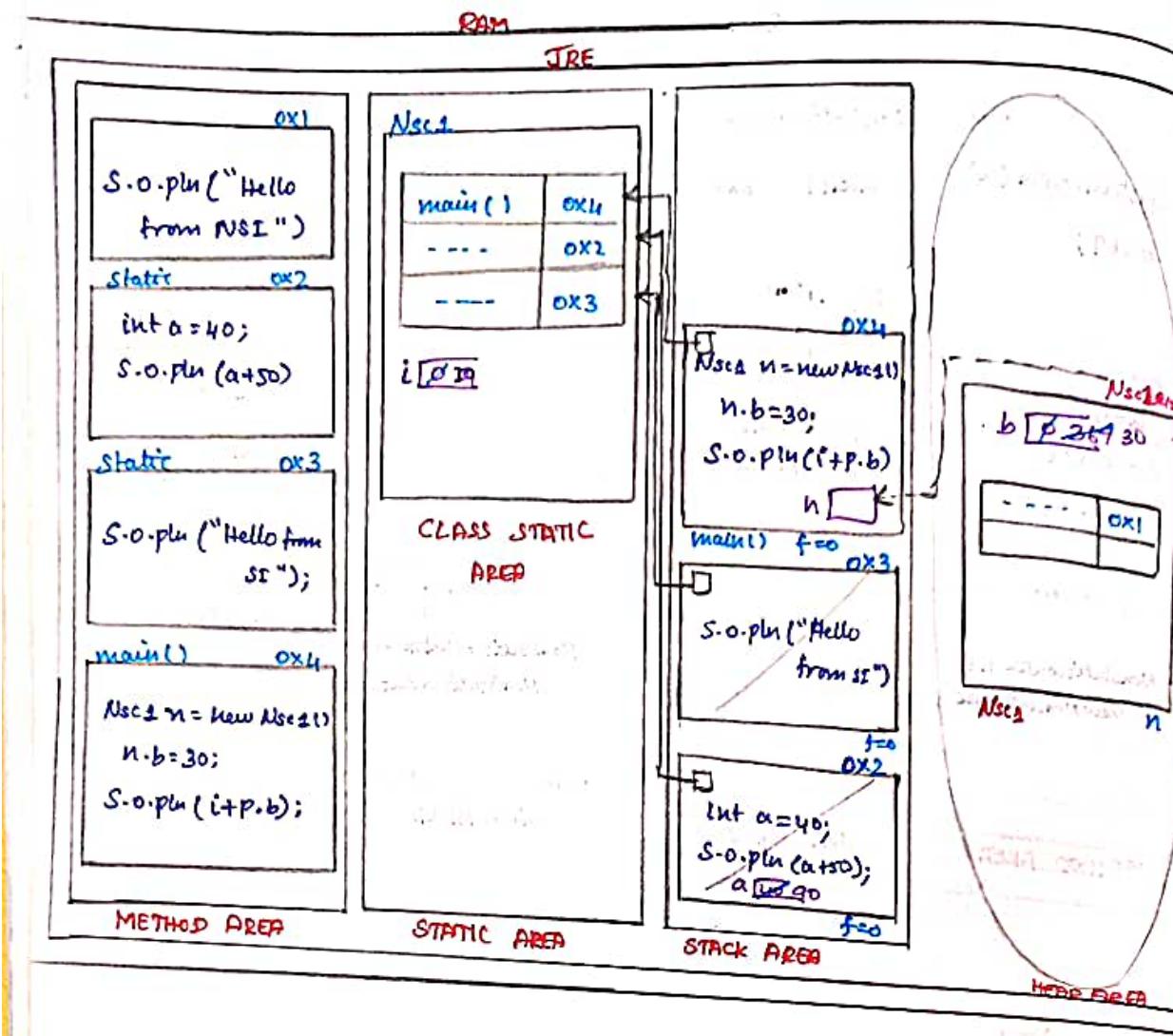
OUTPUT:

90

Hello from SI

Hello from NSI

59



### CONSTRUCTOR:

constructor is a special type of non-static method whose name is the same as the class name but it does not have a return type.

### Syntax:

A programmer can define a constructor by using following syntax.

[A.M J [modifier] className([formal Argument])]

{

Initialization;

}

### CONSTRUCTOR Body:

A constructor body will have the following things:

- \* Load instruction added by the compiler during compile time. i.e., first it will load all the non-static members into the object.
- \* Execute Non-static initializers of a class
- \* Execute programmer written instruction.

### PURPOSE OF THE CONSTRUCTOR:

During the execution of the constructor.

- \* Non-static members of the class will be loaded into the object.
- \* If there is a non-static initializer in the class they start executing from top to bottom order.
- \* Programmer written instruction of the constructor gets executed.

### NOTE :

If the programmer fails to create a constructor then the compiler will add a default constructor.

### Classification of Constructor:

Constructor can be classified into two types based on formal argument.

No argument constructor

Parameterized constructor

#### No argument:

A constructor which doesn't have a formal argument is known as no argument constructor.

#### Parameterized:

A constructor have a formal arguments is known as parameterized constructor.

Syntax: to create no argument constructor

[Access Modifier] [modifier] className ()

```
{  
    //Code;  
}
```

NOTE:

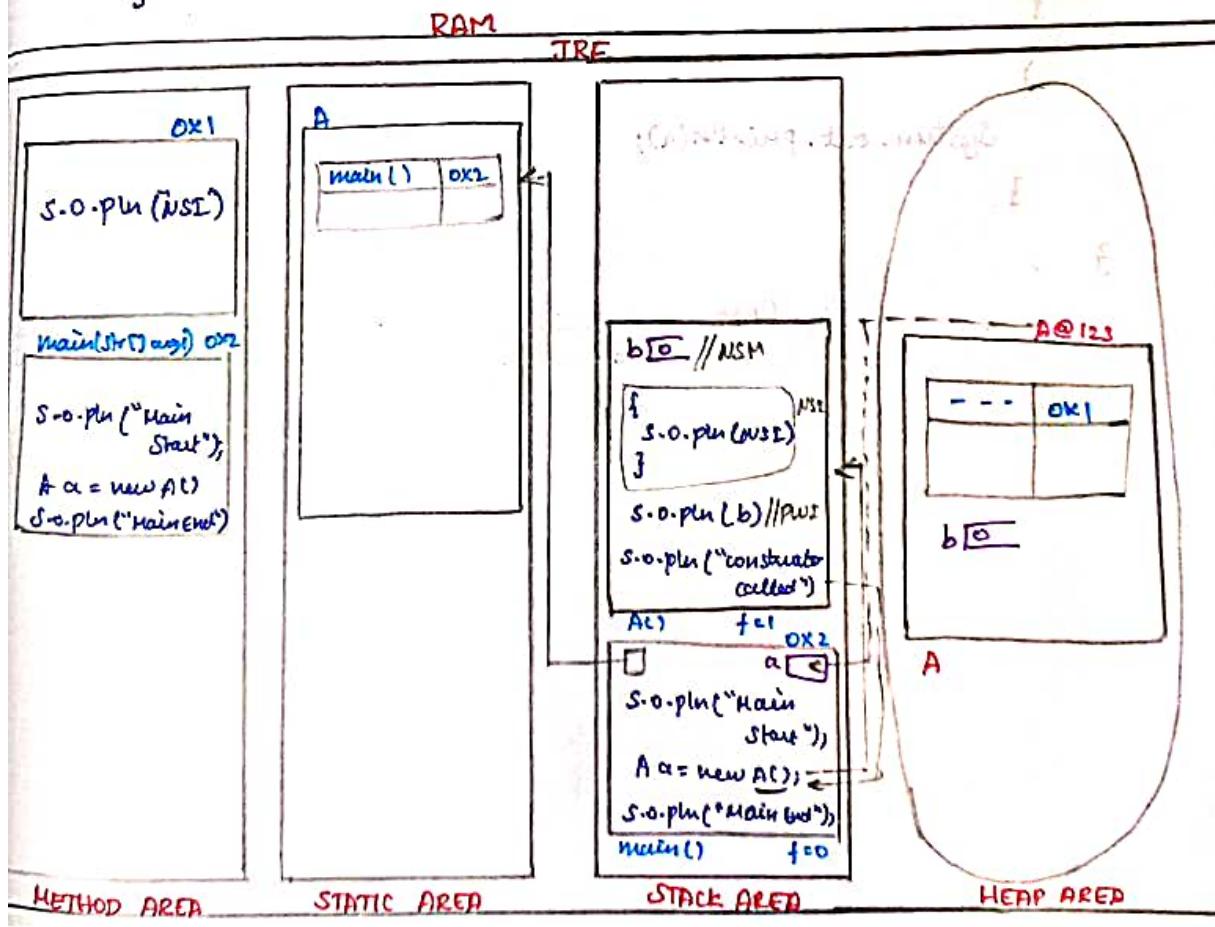
If the programmer fails to create a constructor compiler will add implicitly default constructor.

Working Process of an Object:

- \* A new keyword will create a block of memory in a heap area
- \* Constructor is called.
- \* During execution of the constructor
  - ⇒ All the non static members of the class are loaded into the object
  - ⇒ If there is any non-static initializers they are executed from top. to bottom order.
  - ⇒ Programmer written instruction of the constructor will be executed.
- \* The execution of the constructor is completed
- \* The object is created successfully
- \* The reference is an object is returned by the new keyword
- \* These steps are repeated for every object creation

Example:

```
class A {  
    System.out.println("NSI");  
}  
int b;  
public A()  
{  
    System.out.println(b);  
    System.out.println("constructor called");  
}  
public static void main(String[] args)  
{  
    System.out.println("Main Start");  
    A a = new A();  
    System.out.println("Main End");  
}
```



OUTPUT:

```
Main Start  
NSI  
0  
constructor called
```

### Example :

```
class B
{
    int a = 40;
}

System.out.println(a);

a = 32;

}

B() {
}

int i;

public static void main (String[] args) {
    B b = new B();
    System.out.println(b.i);
}

{
    System.out.println(a);
}

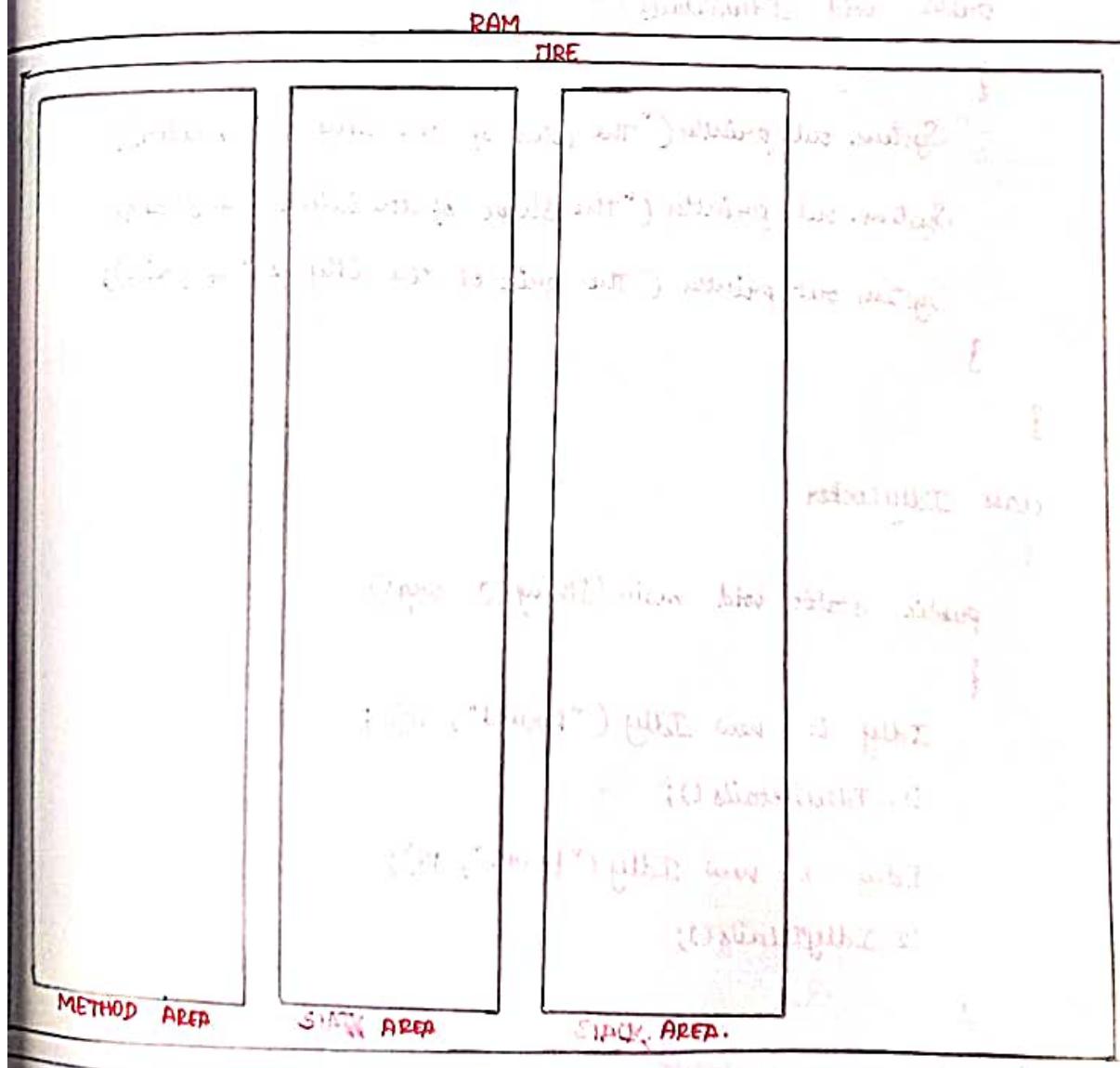
}
```

RAM

JRE

### Example 6:

```
class School  
{  
    String schoolName = "ABC HSC";  
      
    System.out.println("School name is " + schoolName);  
}  
  
public void test()  
{  
    System.out.println("From test");  
}  
  
String sname;  
int std;  
public static void main(String[] args)  
{  
    School s = new School();  
}  
}
```



Example : (for parameterized constructor)

```
class Idly
{
    String color = "white";
    String shape;
    int price;

    Idly (String shape, int price)
    {
        this.shape = shape;
        this.price = price;
    }

    public void IdlyDetails ()
    {
        System.out.println ("The color of the idly is " + color);
        System.out.println ("The shape of the idly is " + shape);
        System.out.println ("The price of the idly is " + price);
    }
}

class IdlyCooker
{
    public static void main (String [] args)
    {
        Idly i1 = new Idly ("round", 10);
        i1.IdlyDetails ();

        Idly i2 = new Idly ("Heart", 15);
        i2.IdlyDetails ();
    }
}
```

**Example:** To withdraw money, go to withdraw menu

**Account**

Acc-holder Name;

Accno;

Password;

Available Balance;

Show Account Details();

(P.T.O)

Example:

class Account

{

String accountHolder;

long accNo;

String password;

double balance;

Account (String accountHolder, long accNo, String password, double bal)

{

this.accountHolder = accountHolder;

this.accNo = accNo;

this.password = password;

this.balance = balance;

}

public void showAccDetails()

{

System.out.println ("The Account holder name is "+accountHolder);

System.out.println ("The Account Number is "+accNo);

System.out.println ("The Account password is "+password);

System.out.println ("The Account balance is "+balance);

}

}

class AccountDriver

{

public static void main (String[] args)

{

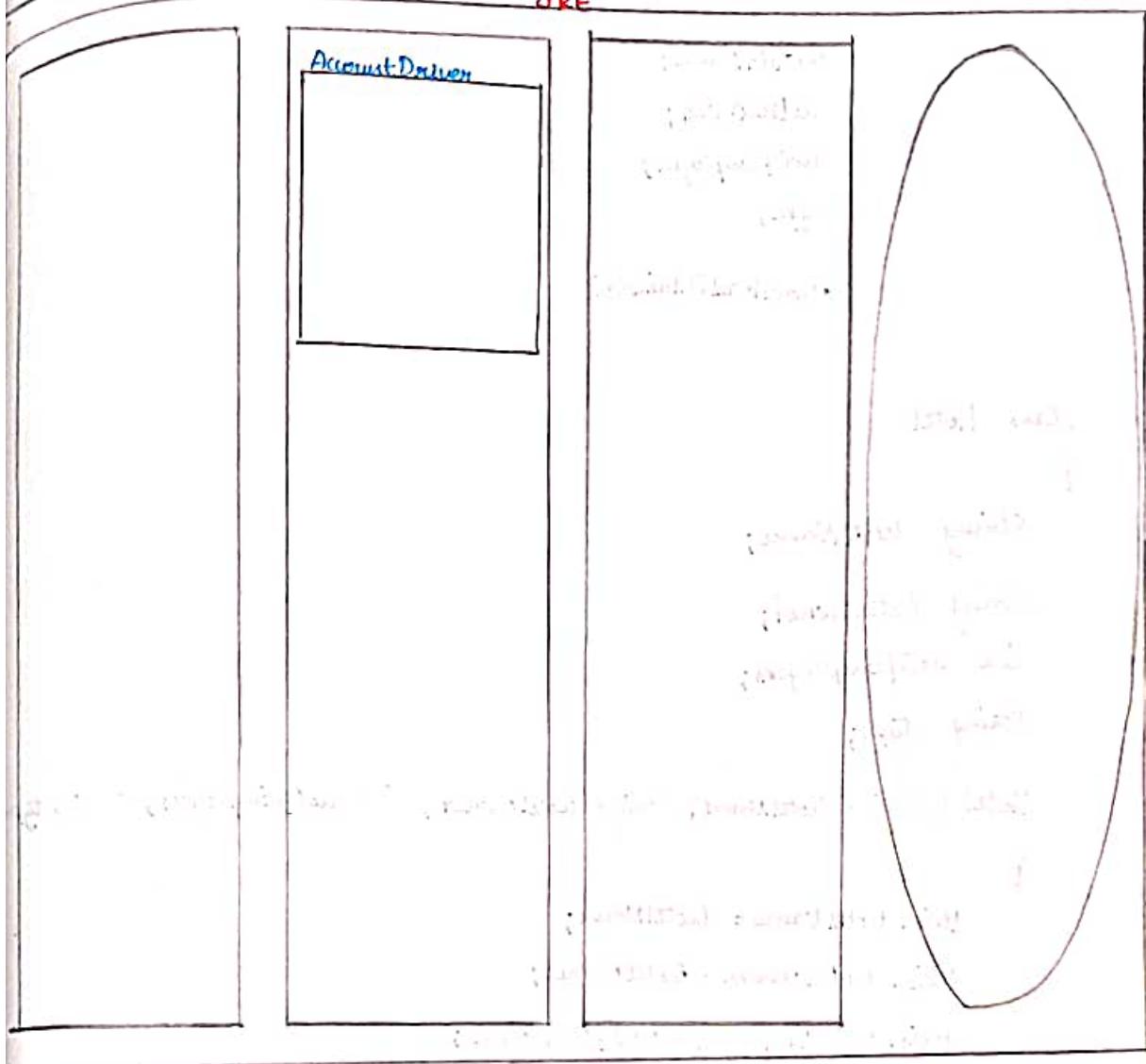
Account a1 = new Account ("Lavanya", 1002315, "lava@123", 2500);

a1.showAccDetails();

Account a2 = new Account ("Manju", 1002316, "manju@312", 2018);

a2.showAccDetails();

}

**OUTPUT :**

The Account holder name is Laranya

The Account Number is 1002315

The Account password is lar@123

The Account balance is 2508.0

The Account holder name is Manju

The Account Number is 1002316

The Account password is manju@512

The Account balance is 20181.0

Example :

```
hotel.Name;  
hotelOwner;  
noOfEmployees;  
type;
```

```
showHotelDetails();
```

```
class Hotel  
{  
    String hotelName;  
    String hotelOwner;  
    int noOfEmployees;  
    String type;  
  
    Hotel (String hotelName, String hotelOwner, int noOfEmployees, String type)  
    {  
        this.hotelName = hotelName;  
        this.hotelOwner = hotelOwner;  
        this.noOfEmployees = noOfEmployees;  
        this.type = type;  
    }  
  
    public void showHotelDetails()  
    {  
        System.out.println("The hotel Name is " + hotelName);  
        System.out.println("The hotel owner name is " + hotelOwner);  
        System.out.println("No. of Employees in hotel " + noOfEmployees);  
        System.out.println("Type of the hotel " + type);  
    }  
}
```

class HotelDriver

{ public static void main (String [] args)

}

    Hotel h1 = new Hotel ("Manju Bhawan", "Manju", 20, "veg");  
    h1.showHotelDetails();

    Hotel h2 = new Hotel ("PD madurai muz", "PD", 10, "Non-veg");

    h2.showHotelDetails();

}

}

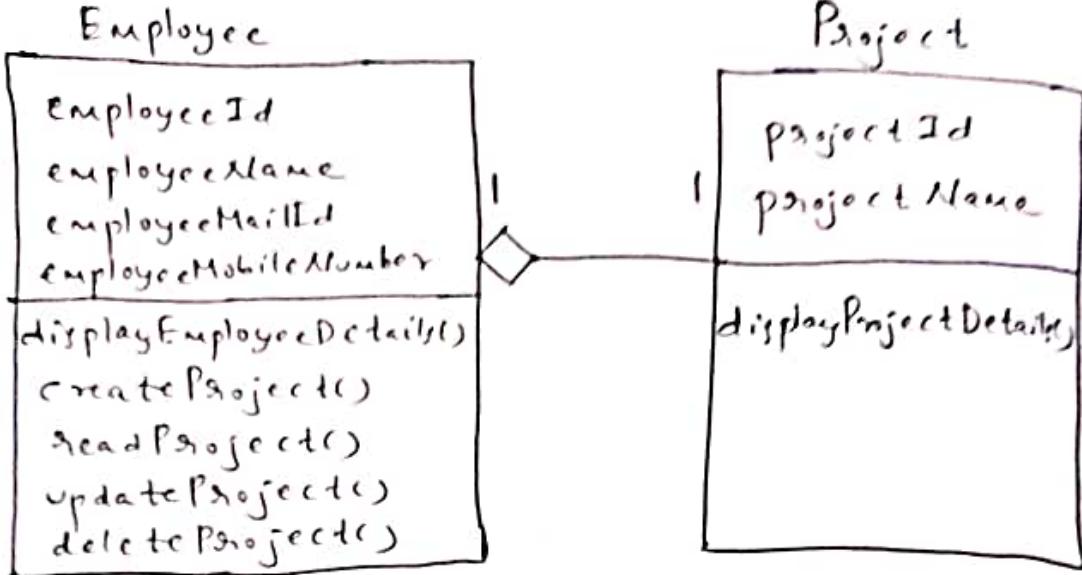
---

parithra → <sup>communication.</sup> 4.

SI.NO	KEYWORDS	FUNCTIONS
33.	<b>private</b>	It is an access modifier. to make the members as private, that is, make the members is only visible within the class. ∴ it is class level modifier.
34.	<b>protected</b>	
35.	<b>public</b>	
36.	<b>return</b>	It is used to return from a method when its execution is complete.
37.	<b>short</b>	It is a primitive datatype to create primitive variable to store short type data.
38.	<b>static</b>	It is a modifier. which describes the characteristics of that method. any class member prefixed with static is called static members. It is global in nature.
39.	<b>strictfp</b> added in 1.2	
40.	<b>Super</b>	It is a keyword used to refer the parent class non-static variables.
41.	<b>switch</b>	It is used for "pattern matching". The switch statement checks the equality of a variable.
42.	<b>Synchronized</b>	
43.	<b>this</b>	It is a non static variable which holds the reference of the current executing object.
44.	<b>throws</b>	
45.	<b>transient</b>	
46.	<b>try</b>	
48.	<b>void</b>	It is a return type which doesn't return anything.
49.	<b>volatile</b>	
50.	<b>while</b>	It is used to start a while loop. This loop iterates a part of

S.NO	KEYWORDS	FUNCTIONS
15.	else	It is an alternate block in an if statement.
16.	enum *** added in 5.0	
17.	extends	It is used to extends a class or between two interfaces. It is used to inherit attributes and methods of one class from another class.
18.	final	
19.	finally	
20.	float	It is primitive data type to create primitive variable to store float type values
21.	for	It is a looping statement. the initialization, condition and updation are given in single line.
22.	goto*	not used
23.	if	it is a decision making statement. If the condition returns true it will execute the code.
24.	implements	It is used to implement an interface to class. That is to give implementation of an method which is in interface we have to get/inherit that method to the class. That can be done by using implements keyword
25.	import	It is a keyword to use one package into another package's class we have to import. That can be done by using import keyword
26.	instanceof	It is a binary operator. It checks whether the object reference having the instance of the given class. It returns boolean type data
27.	int	It is a primitive data type to create primitive variable to store int type value
28.	interface	It is used to declare a special type of class that only contains abstract method.
29.	long	It is a primitive datatype to create primitive variable to store long type value
30.	native	
31.	new	used to create an block of memory in heap area and hold the address of the object.
32.	Package	

SL.NO	KEYWORD	FUNCTIONS
1.	<b>abstract</b>	It indicates that the class have an incomplete nature. It also declared to the method which indicates that method is an incomplete method, i.e., the method doesn't have implementation.
2.	<b>assert ***</b> added in 1.11	
3.	<b>boolean</b>	It is a primitive datatype to create primitive variable to store boolean type values.
4.	<b>break</b>	It is used to break the loop or switch statement. it breaks the current flow and comes out of the block / loop.
5.	<b>byte</b>	It is a primitive datatype to create primitive variable to store byte type values
6.	<b>case</b>	It is used in switch statement. It is a conditional label. in case we can give data / expression
7.	<b>catch</b>	
8.	<b>char</b>	It is a primitive datatype to create primitive variable to store char type value .
9.	<b>class</b>	It is used to declare a class members . It is a blue print of an object.
10.	<b>const *</b>	not used
11.	<b>continue</b>	
12.	<b>default</b>	It is used to specify the default block of code in a switch statement.
13.	<b>do</b>	It is used to the control statement to declare a do-while loop.
14.	<b>double</b>	It is a primitive datatype to create primitive variable to store double type value



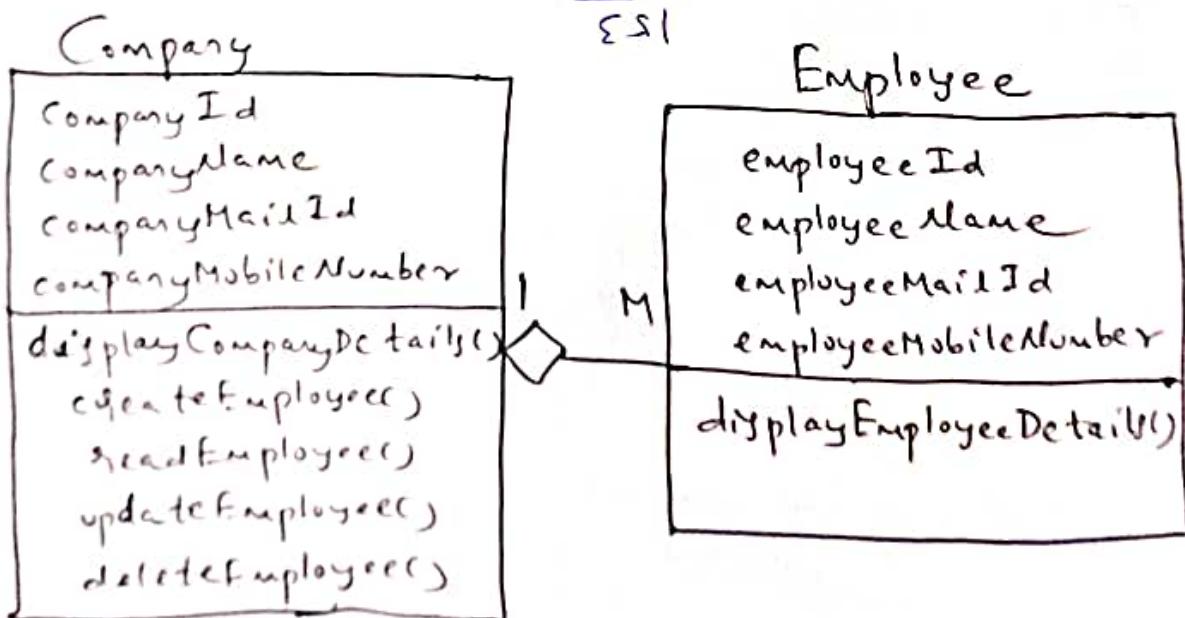
[Designability-01]

= 1 + n + s + h

1351

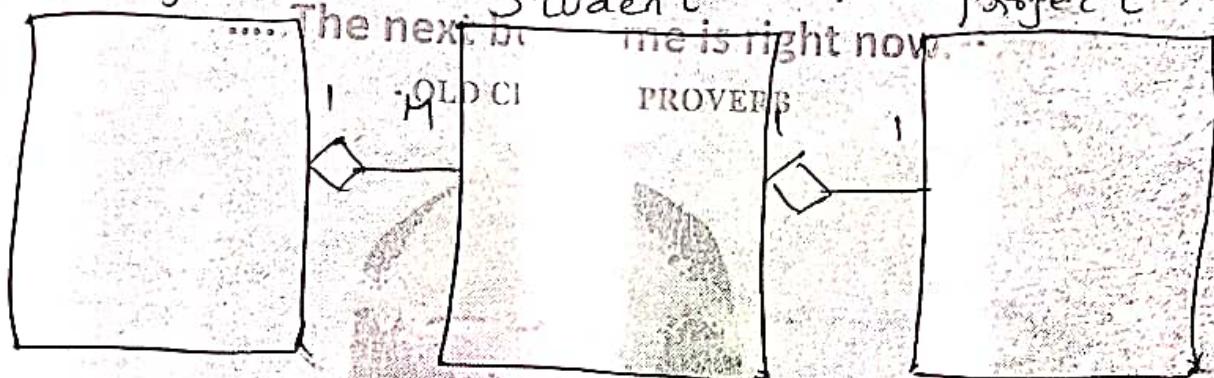
CS1 = E k E S + S!

131



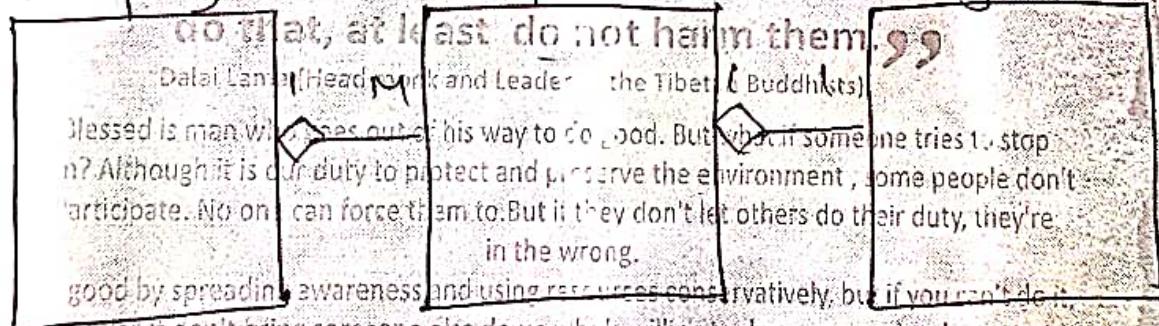
[Designability-02]

The best time to plant a tree is 20 years ago.  
The next best time is now.



[Designability - 0.4]

66 Company you can, help Employee you cannot Project



[Designability - 0.3]

66 The good times of today are the sad thoughts of tomorrow.

Bob Marley (Famous Singer-Songwriter)

human race is one that takes happiness in revelry and festivity. We live for today and think about the future that has taken severely cost our planet. we have been using the natural resources at such a rapid rate that they will be long gone before our future generations can see them.

need to adopt sustained development. Use only what we need and saving the rest for the future. if we only think of today, then there will be no tomorrow.

I) WAP to print the given pattern.

```

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

```

class P1

{

public static void main (String [] args)

{

for (int i=1; i<=5; i++)
 {

for (int j=1; j<=5; j++)
 {

System.out.print ("\* "+ " ");

}
 }

System.out.println ();
 }
}

OUTPUT:

```

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

```

Tracing

① i=1 1 <= 5 (T)	② i=2 2 <= 5 (T)	③ i=3 3 <= 5 (T)	④ i=4 4 <= 5 (T)	⑤ i=5 5 <= 5 (T)	⑥ i=6 6 <= 5 (F)
(i) j=1 1 <= 5 (T)	(ii) j=1 1 <= 5 (T)	(iii) j=1 1 <= 5 (T)	(iv) j=1 1 <= 5 (T)	(v) j=1 1 <= 5 (T)	(vi) j=1 1 <= 5 (T)
(vii) j=2 2 <= 5 (T)	(viii) j=2 2 <= 5 (T)	(ix) j=2 2 <= 5 (T)	(x) j=2 2 <= 5 (T)	(xi) j=2 2 <= 5 (T)	(xii) j=2 2 <= 5 (T)
(viii) j=3 3 <= 5 (T)	(ix) j=3 3 <= 5 (T)	(x) j=3 3 <= 5 (T)	(xi) j=3 3 <= 5 (T)	(xii) j=3 3 <= 5 (T)	(xiii) j=3 3 <= 5 (T)
(ix) j=4 4 <= 5 (T)	(x) j=4 4 <= 5 (T)	(xi) j=4 4 <= 5 (T)	(xii) j=4 4 <= 5 (T)	(xiii) j=4 4 <= 5 (T)	(xiv) j=4 4 <= 5 (T)
(x) j=5 5 <= 5 (T)	(xi) j=5 5 <= 5 (T)	(xii) j=5 5 <= 5 (T)	(xiii) j=5 5 <= 5 (T)	(xiv) j=5 5 <= 5 (T)	(xv) j=5 5 <= 5 (T)
(xi) j=6 6 <= 5 (F)	(xii) j=6 6 <= 5 (F)	(xiii) j=6 6 <= 5 (F)	(xiv) j=6 6 <= 5 (F)	(xv) j=6 6 <= 5 (F)	(xvi) j=6 6 <= 5 (F)



3) WAP to print the given pattern using nested looping in Java.

```
* * * * *
* * * *
* * *
* *
*
```

```
* * * * *
* * * *
* * *
* *
*
```

class P3{

public static void main(String[] args){

int n=5;

for (int i=1; i<=n; i++) {

for (int j=1; j<=n; j++) {

if (i==j)

System.out.print("\*" + " "));

else

System.out.print(" "));

3 System.out.println();

}

Output

```
* * * * *
* * * *
* * *
* *
*
```

### Tracing

① i=1 1<=5 (T) (i) j=1 1<=5 (T) 1<=1 (T) S.o.p(*)	② i=2 2<=5 (T) (i) j=1 1<=5 (T) 2<=1 (F) S.o.p(" ")	③ i=3 3<=5 (T) (i) j=1 1<=5 (T) 3<=1 (F) S.o.p(" ")	④ i=4 4<=5 (T) (i) j=1 1<=5 (T) 4<=1 (F) S.o.p(" ")	⑤ i=5 5<=5 (i) j=1 1<=5 (T) 5<=1 (F) S.o.p(" ")	⑥ i=6 6<=5 (F)
(ii) j=2 2<=5 (T) 2<=2 (T) S.o.p(*)	(ii) j=2 2<=5 (T) 2<=2 (T) S.o.p(*)	(ii) j=2 2<=5 (T) 2<=2 (F) S.o.p(" ")	(ii) j=2 2<=5 (T) 4<=2 (F) S.o.p(" ")	(ii) j=2 2<=5 (T) 5<=2 (F) S.o.p(" ")	
(iii) j=3 3<=5 (T) 3<=3 (F) S.o.p(*)	(iii) j=3 3<=5 (T) 3<=3 (T) S.o.p(*)	(iii) j=3 3<=5 (T) 3<=3 (T) S.o.p(*)	(iii) j=3 3<=5 (T) 4<=3 (F) S.o.p(" ")	(iii) j=3 3<=5 (T) 5<=3 (F) S.o.p(" ")	
(iv) j=4 4<=5 (T) 4<=4 (F) S.o.p(*)	(iv) j=4 4<=5 (T) 4<=4 (T) S.o.p(*)	(iv) j=4 4<=5 (T) 4<=4 (T) S.o.p(*)	(iv) j=4 4<=5 (T) 4<=4 (T) S.o.p(*)	(iv) j=4 4<=5 (T) 5<=4 (F) S.o.p(" ")	
(v) j=5 5<=5 (T) 5<=5 (T) S.o.p(*)	(v) j=5 5<=5 (T) 5<=5 (T) S.o.p(*)	(v) j=5 5<=5 (T) 5<=5 (T) S.o.p(*)	(v) j=5 5<=5 (T) 4<=5 (F) S.o.p(*)	(v) j=5 5<=5 (T) 5<=5 (T) S.o.p(*)	
(vi) j=6 6<=5 (F)	(vi) j=6 6<=5 (F)	(vi) j=6 6<=5 (F)	(vi) j=6 6<=5 (F)	(vi) j=6 6<=5 (F)	(vi) j=6 6<=5 (T)

3

4) WAP to print the given pattern

```
* * * * *
*
*
*
*
*
* * * * *
```

class P4{

public static void main (String[] args) {

int n=5;

for (int i=1; i<=n; i++) {

for (int j=1; j<=n; j++) {

if (i==1 || j==1 || i==n || j==n)

System.out.print ("\*");

else

System.out.print (" "));

}

System.out.println();

}

}

Tracing

OUTPUT:

```
* * * * *
*
*
*
*
*
* * * * *
```

① i=1 1 <= 5 (T) (i) j=1 1 <= 5 (T) [i==1    j==1    i==n    j==n] (T) S.O.P(*)	② i=2 2 <= 5 (T) (ii) j=1 1 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	③ i=3 3 <= 5 (T) (iii) j=1 1 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	④ i=4 4 <= 5 (T) (iv) j=1 1 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑤ i=5 5 <= 5 (T) (v) j=1 1 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)
⑥ j=2 2 <= 5 (T) [i==1    j==1    i==n    j==n] (T) S.O.P(*)	⑦ j=2 2 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑧ j=2 2 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑨ j=2 2 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑩ j=2 2 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)
⑪ j=3 3 <= 5 (T) [i==1    j==1    i==n    j==n] (T) S.O.P(*)	⑫ j=3 3 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑬ j=3 3 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑭ j=3 3 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑮ j=3 3 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)
⑯ j=4 4 <= 5 (T) [i==1    j==1    i==n    j==n] (T) S.O.P(*)	⑰ j=4 4 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑱ j=4 4 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑲ j=4 4 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	⑳ j=4 4 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)
㉑ j=5 5 <= 5 (T) [i==1    j==1    i==n    j==n] (T) S.O.P(*)	㉒ j=5 5 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	㉓ j=5 5 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	㉔ j=5 5 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)	㉕ j=5 5 <= 5 (T) [i==1    j==1    i==n    j==n] (F) S.O.P(*)
㉖ j=6 6 <= 5 (F)	㉗ j=6 6 <= 5 (F)	㉘ j=6 6 <= 5 (F)	㉙ j=6 6 <= 5 (F)	㉚ j=6 6 <= 5 (F)

(W)

⑥ i=6  
6 <= 5 (F)

## CONSTRUCTOR OVERLOADING

Definition :

If a class is having more than one constructor it is known as constructor overloading.

Rule :

The signature of the constructor must be different.

Example :

class Eraser

{

String color;

String shape;

double price;

int length;

Eraser()

{ }

Eraser(String color, String shape)

{

this.color = color; this.shape = shape

this.shape = shape;

}

Eraser(String color)

{

this.color = color;

}

Eraser(int length)

{

this.length = length;

}

Eraser(double price)

{

this.price = price;

}

Eraser(String color, String shape, double price, int length)

{

this.color = color;

this.shape = shape;

this.price = price;

```
    this.length = length;  
}  
public void eraserDetails()  
{  
    System.out.println(color);  
    System.out.println(shape);  
    System.out.println(price);  
    System.out.println(length);  
}  
  
}  
  
3  
class EraserDriver  
{  
    public static void main(String[] args)  
    {  
        Eraser e1 = new Eraser();  
        Eraser e2 = new Eraser("white", "round", 3, 5);  
        Eraser e3 = new Eraser(5);  
        Eraser e4 = new Eraser("red");  
        Eraser e5 = new Eraser(10.0);  
        Eraser e6 = new Eraser("Blue", "Rectangle");  
        e2.eraserDetails();  
        e6.eraserDetails();  
        e1.eraserDetails();  
    }  
}
```

(Hence this will print white, round, 3, 5, white, round, 5)

⑥

## OUTPUT:

white

grand

3.0

5

blue

Alphab.

Grand

3.0

5

Blue

## Rectangle

0.0

0

null

null

0.0

0

length

width

Alphab. 1st

Grand 1st

Point point

Value width

2 length

2 width

Length, width, start, end (Alphab. 1st) stored

(Alphab. = Alphab. 1st)

(Grand = Grand. 1st)

(Point = Point. 1st)

(Value = Value. 1st)

(Length 1st, Grand 1st) stored

(Alphab. = Alphab. 1st)

(Grand = Grand. 1st)

(Point point, Grand 1st) stored

(Alphab. = Alphab. 1st)

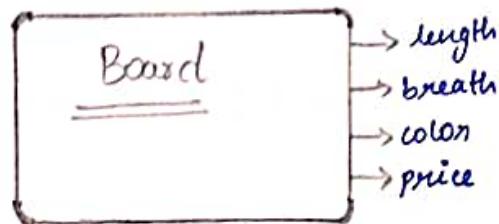
(Point = Point. 1st)

(Value value, Point point) stored

(Point = Point. 1st)

(Value = Value. 1st)

Example:



```
class Board
{
    int length;
    int breath;
    String color;
    double price;

    Board()
    {}

    Board (int length, int breath, String color, double price)
    {
        this.length = length;
        this.breadth = breadth;
        this.color = color;
        this.price = price;
    }

    Board (int length, int breath)
    {
        this.length = length;
        this.breadth = breadth;
    }

    Board (int length, String color)
    {
        this.length = length;
        this.color = color;
    }

    Board (String color, double price)
    {
        this.color = color;
        this.price = price;
    }
}
```

Board (int length, int breadth, double price) : properties

{  
this.length = length;  
this.breadth = breadth;  
this.price = price;

}

Board (int length, int breadth, String color) : properties

{  
this.length = length;  
this.breadth = breadth;  
this.color = color;

}

public void boardDetails() {  
overrided method of Board class

{

System.out.println("The length of the board is " + length);

System.out.println("The breadth of the board is " + breadth);

System.out.println("The color of the board is " + color);

System.out.println("The price of the board is " + price);

}

}

class BoardDriver

{ public static void main(String[] args)

{

Board b1 = new Board();

Board b2 = new Board("white", 500);

Board b3 = new Board(35, 45, "green", 700);

Board b4 = new Board(35, 45, 800);

b2.boardDetails();

b3.boardDetails();

}

}

**OUTPUT:**

Buying Board, Length 35, Width 45, Height 10, Price 500.0

The length of the board is 0

The breath of the board is 0

The color of the board is white

The price of the board is 500.0

The length of the board is 35

The breath of the board is 45

The color of the board is green

The price of the board is 200.0

**Example:**

DUSTER → color;

DUSTER → price;

DUSTER → brand;

class Duster

{

String color;

double price;

String brand;

Duster()

{ }

Duster(String color, double price)

{

this.color = color;

this.price = price;

}

Duster (String color, String brand)

{

This. color = color;

This. brand = brand;

}

Duster (String color)

{

This. color = color;

}

Duster (double price, String brand)

{

This. price = price;

This. brand = brand;

}

Duster (String color, double price, String brand)

{

This. color = color;

This. price = price;

This. brand = brand;

}

public void dusterDetails ()

{

System.out.println ("The color of the duster is " + color);

System.out.println ("The price of the duster is " + price);

System.out.println ("The duster brand is " + brand);

}

}

```
class DusterDriver
```

```
{
```

```
    public static void main(String [] args)
```

```
{
```

```
        Duster d1 = new Duster();
```

(color = color . null)

(price = price . null)

```
        Duster d2 = new ("black", 50);
```

(color = color . value)

```
        Duster d3 = new ("red", "camlin");
```

(model = model . value)

```
        Duster d4 = new (50, "camlin");
```

(price = price . value)

```
        Duster d5 = new ("black", 80, "camlin");
```

(color = color . value)

```
        d3.dusterDetails();
```

(brand = brand . null)

```
        d5.dusterDetails();
```

(brand = brand . null)

```
        d2.dusterDetails();
```

(color = color . value)

```
}
```

(model = model . value)

(color = color . value)

(brand = brand . value)

OUTPUT:

The color of the duster is red

(color = color . value)

The price of the duster is 0.0

(price = price . value)

The duster brand is camlin

(brand = brand . value)

The color of the duster is black

(color = color . value)

The price of the duster is 80.0

(price = price . value)

The duster brand is camlin

(brand = brand . value)

The color of the duster is black

(color = color . value)

The price of the duster is 50.0

(price = price . value)

The duster brand is null

(brand = brand . null)

## constructor chaining:

\* A constructor calling another constructor is known as constructor chaining

\* In java, we can achieve constructor chaining by using two ways.

⇒ `this()` (this call statement)

⇒ `super()` (super call statement)

### `this()`:

It is used to call the constructor of the same class from another constructor.

#### RULE:

`this()` can be used only inside the constructor

it should always be the first statement in the constructor

The recursive call to the constructor is not allowed  
(calling by itself)

If a class has n number of constructors, we can use `this()` statement for n-1 constructor only (at least a constructor should be without `this()`)

#### NOTE:

If the constructor has `this()` statement then the compiler doesn't add load instruction & non-static initializers into the constructor body.

### Example :

```
class student {  
    String sname;  
    int sid;  
    long cno;  
  
    Student()  
    {}  
  
    Student(String sname)  
    {  
        this.sname = sname;  
        System.out.println("single parameterized const");  
    }  
  
    Student(String sname, int sid)  
    {  
        this(sname);  
        this.sid = sid;  
        System.out.println("two parameterized construct");  
    }  
  
    Student(String sname, int sid, long cno)  
    {  
        this(sname, sid);  
        this.cno = cno;  
        System.out.println("Three argument constructor");  
    }  
}
```

class StudentDriver

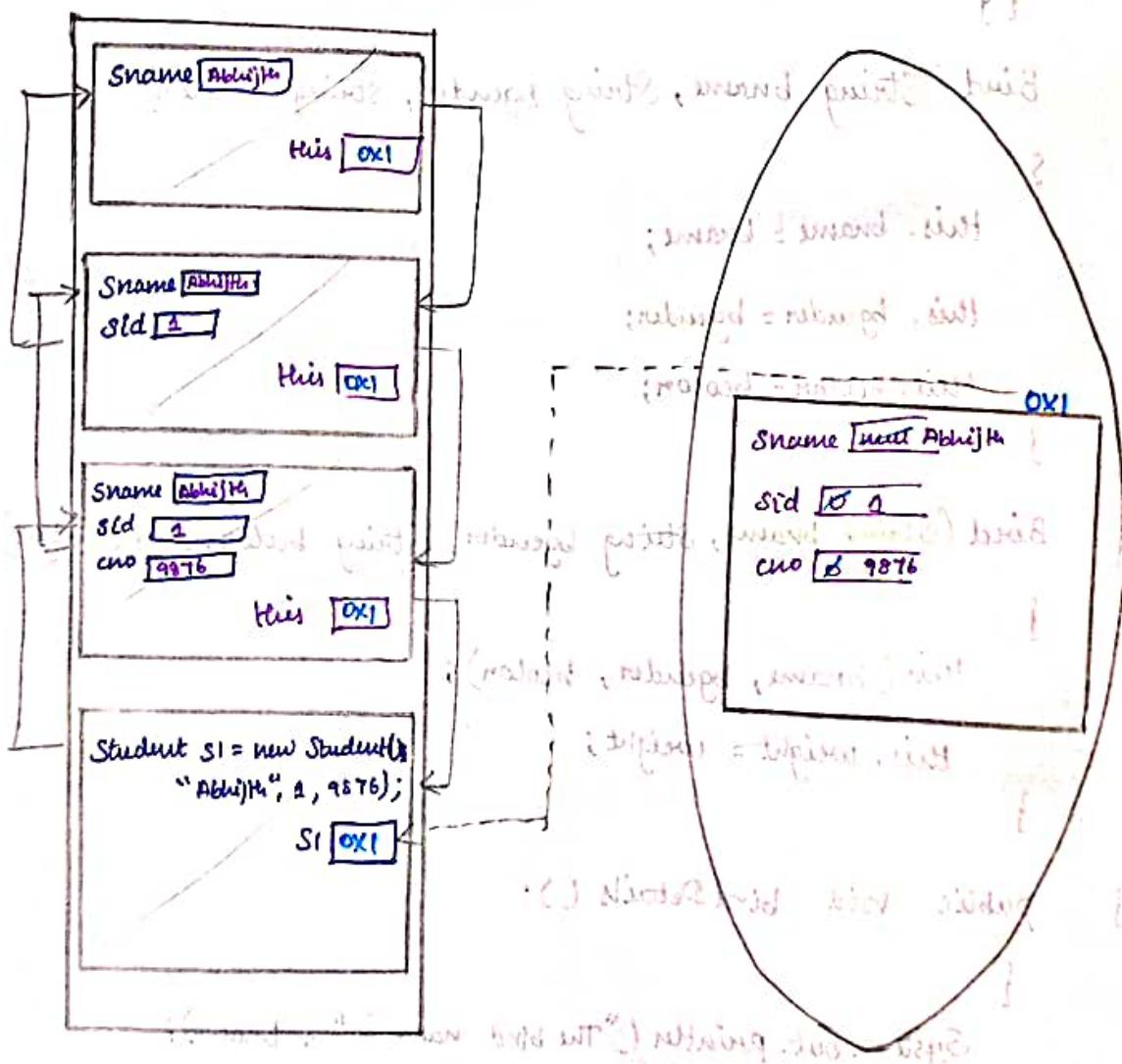
{ public static void main(String[] args)

}

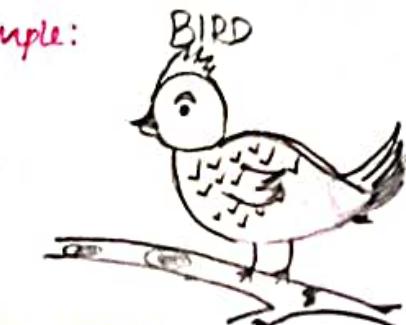
Student s1 = new Student("Abhijit", 1, 9876);

}

Tracing:



Example:



class Bird  
{

modified  
by  
Raman / uday

String bname;

String bgender;

String bcolor;

double weight;

Bird ()

{ }

Bird (String bname, String bgender, String bcolor)

{

this. bname = bname;

this. bgender = bgender;

this. bcolor = bcolor;

}

Bird (String bname, String bgender, String bcolor, double weight)

{

this (bname, bgender, bcolor);

this. weight = weight;

}

public void birdDetails ()

{

System.out.println ("The bird name is " + bname);

System.out.println ("The gender is " + bgender);

System.out.println ("The color is " + bcolor);

System.out.println ("The weight is " + weight);

}

3

## class BirdDriver

```
{  
    public static void main(String[] args)  
    {  
        Bird b1 = new Bird("Parrot", "male", "green", 300);  
        Bird b2 = new Bird("chicken", "female", "white", 1.5);  
  
        b1.birdDetails();  
        b2.birdDetails();  
    }  
}
```

### OUTPUT:

The bird name is Parrot

The gender is male

The color is green

The weight is 300.0

The bird name is chicken

The gender is female

The color is white

The weight is 1.5

31/3/22

## Example :

WATCH



- color
- price
- Brand
- type

class Watch

{

String color;

double price;

String brand;

String type;

Watch()

{ }

Watch(String color)

{

this.color = color;

}

Watch(String color, double price)

{

this(color);

this.price = price;

}

Watch(String color, String brand, double price)

{

this(color, price);

this.brand = brand;

}

Watch(String color, double price, String brand, String type)

{

this(color, price, brand);

this.type = type;

}

public void watchDetails()

{

System.out.println(color);

System.out.println(price);

System.out.println(brand);

System.out.println(type);

}

}

Statement of return type

```

class WatchDriver {
    public static void main(String[] args) {
        Watch w1 = new Watch("Blue");
        Watch w2 = new Watch("black", 1800, "fast track", "analog");
        w1.watchDetails();
        w2.watchDetails();
    }
}

```

### OUTPUT:

Blue (null, null) date, with pointer to memory address  
 0.0 means no month in milliseconds we're initialized.  
 null  
 null and after every bit of initializing variables, we'll get  
 black 1800.0 to initialized blue date, that next needs , make with  
 fast track  
 analog

which all pointers and the initializing process will be.

Now, basically we're initializing to memory in object  
 so called reference from there we go to initialize which is  
 made up of initializing big memory date of those variables  
 public static no variable is created  
 all attributes have option of initial, initialized date is  
 all initialized have private, protected, and public

## PRINCIPLES OF OOPS

Principle Of Ooops:

Object oriented programming has the following Principles:

\* Encapsulation

\* Inheritance

\* Polymorphism

\* Abstraction

Encapsulation:

\* The process of binding the state (attribute/fields) and behaviour of an object together is known as encapsulation.

\* We can achieve encapsulation in java with the help of the class, class has both state and behaviour of an object.

ADVANTAGE OF ENCAPSULATION:

By using encapsulation we can achieve the data hiding.

DATA HIDING:

\* It is a process of restricting the direct access of data members of an object and provides indirect secured access of data members via methods of the same object is known as data hiding.

\* Data hiding helps to verify and validate the data before storing and modifying it.

Dingi

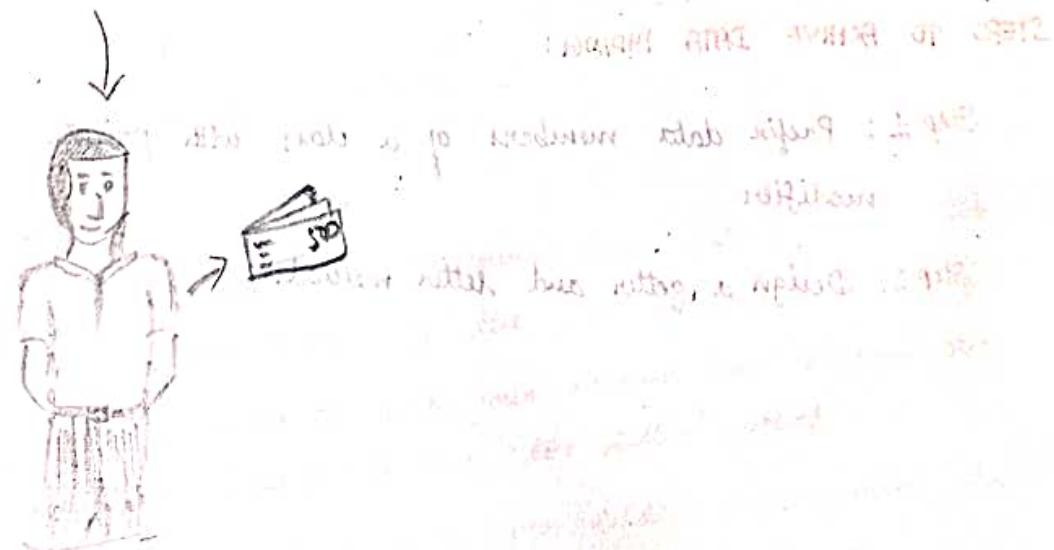
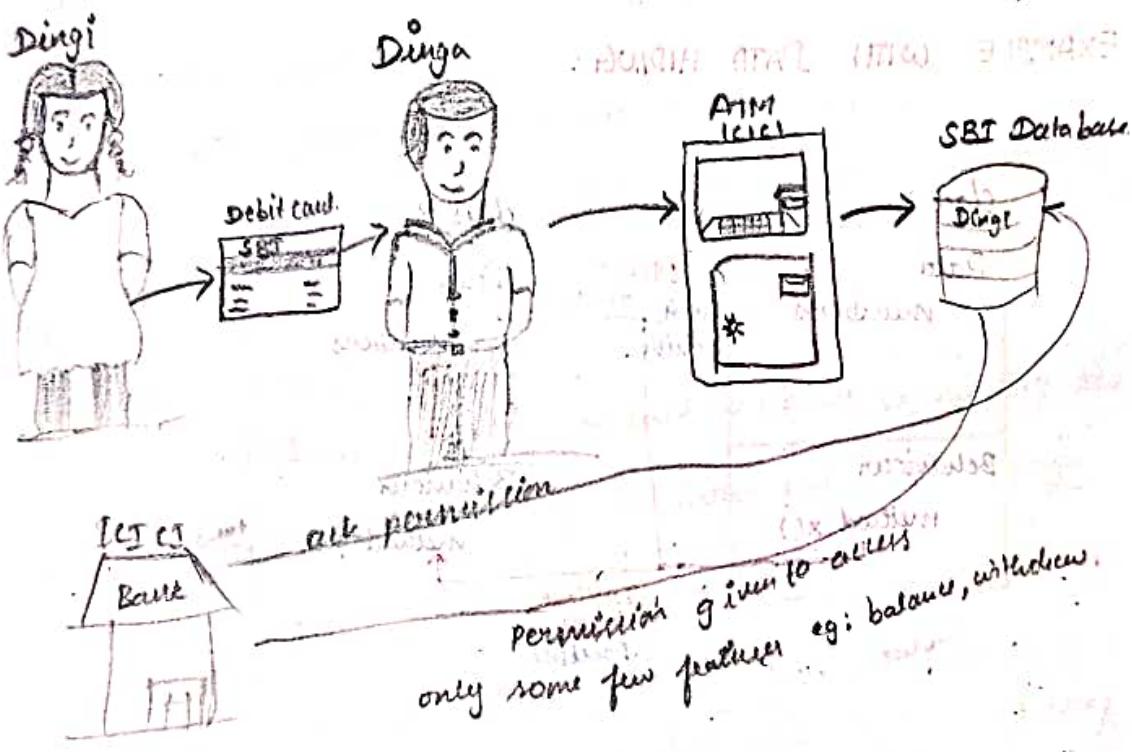


→ he is having a debit card of his friend, now he is going to withdraw the money in ICICI ATM, but his friend account is in SBI bank.

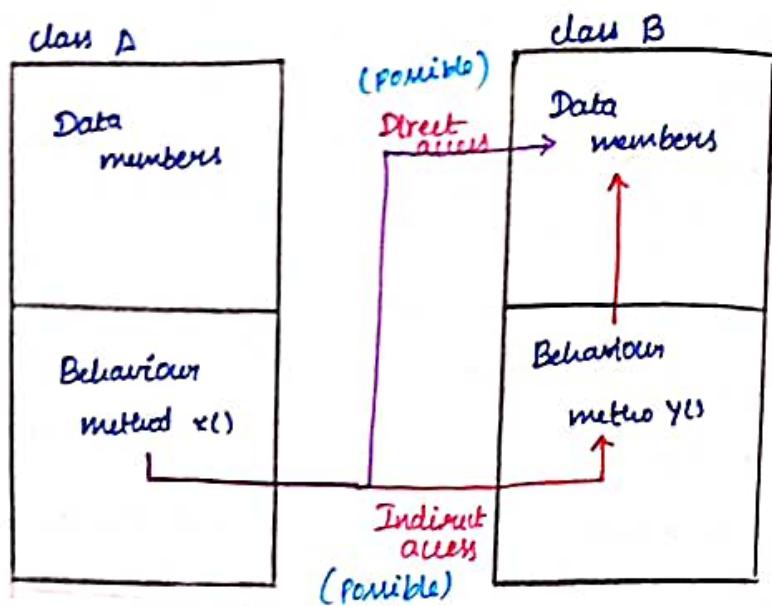
Dingi

Now can Dingi able to withdraw the money from ICICI ATM? Yes. How?

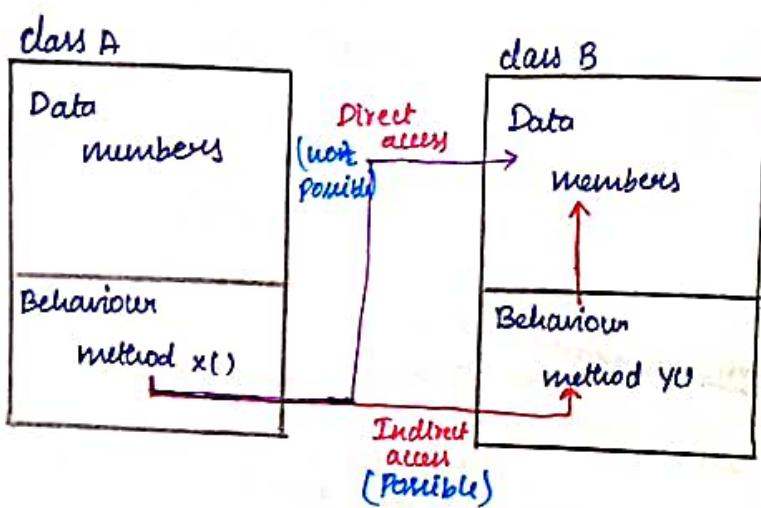
The details of Dingi Account will be present in which server? SBI bank server, but who is accessing the data? ICICI bank due you think the details are open for everyone? No then still how it is accessing? ICICI bank ask the permission to SBI bank.



## EXAMPLES WITHOUT DATA HIDING:



## EXAMPLE WITH DATA HIDING:



## STEPS TO ACHIEVE DATA HIDING:

Step 1: Prefix data members of a class with private modifier

Step 2: Design a getter and setter method.

## PRIVATE MODIFIER:

- \* Private is an access modifier
- \* private is a class level modifier
- \* If the members of the class are prefixed with a private modifier then we can access that member only within the class

NOTE: Data hiding can be achieved with the help of a private modifier

## GETTER AND SETTER METHOD:

### Getter Method:

- \* The getter method is used to fetch the data.
- \* The return type of the getter method is the type of the hidden value.

### Setter method:

- \* The setter method is used to update or modify the data.
- \* The return type of the setter method is always void.

### NOTE:

The validation and verification can be done in this method before storing the data and before reading private data member.

### NOTE:

- \* If you want to make your hidden data member - only readable then create only the getter method.
- \* If you want to make your hidden data member - only modifiable then create only the setter method.
- \* If you want to make your hidden data member both readable and modifiable then create both getter and setter methods.

\* If you want to make your hidden data member neither readable and nor modifiable then don't create a getter and setter method.

### ADVANTAGES OF DATA HIDING:

- \* Provides security to the data members
- \* We can verify and validate the data before modifying it
- \* We can make the data member of the class to
  - ⇒ only readable
  - ⇒ only modifiable
  - ⇒ Both readable and modifiable
  - ⇒ Neither readable nor modifiable.

Example:

class Sim

{

    private String serviceProvider;

    private long simNo;

    private double balance;

    String colon;

}

    public String getServiceProvider()

    {

        return serviceProvider;

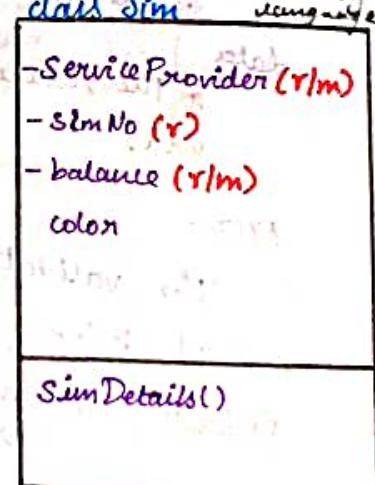
    }

    public void setServiceProvider(String serviceProvider)

    {

        this.serviceProvider = serviceProvider;

    }



```
public long getSimNo()  
{  
    return simNo; } // Wiederrufe hier statisch
```

```
public double getBalance()  
{  
    return balance; } // (Sim) returning 300.00000000000002  
// (Sim) returning 300.00000000000002  
// (Sim) returning 300.00000000000002
```

```
public void setBalance(double balance)  
{  
    this.balance = balance; } //
```

### // constructor

```
Sim()  
{ } // new Upcast (new hier statisch)
```

```
(Sim(String serviceProvider, long simNo, double balance,  
String color)  
{ } // (String serviceProvider, long simNo, double balance, String color)
```

```
{  
    this.serviceProvider = serviceProvider;  
    this.simNo = simNo;  
    this.balance = balance; } // (String serviceProvider, long simNo, double balance, String color)  
    this.color = color; } // (String serviceProvider, long simNo, double balance, String color)
```

```
}
```

Java

JAVA

class class

method

constructor

use

24

//behavior:

```
public void simDetails()
{
    System.out.println(serviceProvider);
    System.out.println(simNo);
    System.out.println(balance);
    System.out.println(color);
}

class SimDriver
{
    public static void main(String[] args)
    {
        Sim s1 = new Sim("Airtel", 7825251213, 250, "Red");
        System.out.println(s1.getServiceProvider());
        System.out.println(s1.getSimNo());
        s1.setServiceProvider("vodafone");
        s1.simDetails();
    }
}
```

OUTPUT

Airtel  
Airtel  
7825251213  
vodafone  
7825251213  
250  
Red

Example:

class School

-sname (r)	(beginning, middle, end)
-principal (r/w)	middle, middle
-noOfStaff (r/w)	(beginning, middle, end)
-HeadMaster (r/w)	(beginning, middle, end)
SchoolDetails()	

{

private String sname;

private String principal;

private int noOfStaff;

private String headMaster;

School()

{}

School (String sname, String principal)

{

this.sname = sname;

this.principal = principal;

}

School (String sname, String principal, int noOfStaff)

{

this(sname, principal);

this.noOfStaff = noOfStaff;

}

School (String sname, String principal, int noOfStaff, String headMaster)

{

this(sname, principal, noOfStaff);

this.headMaster = headMaster;

}

```

public String getName()
{
    return name;
}

public String getPrincipal()
{
    return principal;
}

public void setPrincipal(String principal)
{
    this.principal = principal;
}

public int getNoOfStaff()
{
    return noOfStaff;
}

public void setNoOfStaff(int noOfStaff)
{
    this.noOfStaff = noOfStaff;
}

public String getHeadMaster()
{
    return headMaster;
}

public void setHeadMaster(String headMaster)
{
    this.headMaster = headMaster;
}

```

```
public void SchoolDetails()
{
    System.out.println("The school name is " + name);
    System.out.println("The principal name is " + principal);
    System.out.println("Total no of Staff in a school " + noOfStaff);
    System.out.println("Head Master name is " + headMaster);
}
```

{

class SchoolDriver

{

public static void main (String [ ] args)

{

School s = new School ("P.D Mr.Sec.school", "Lavanya", 45,
 "Kauju");

System.out.println (s.getName());

System.out.println (s.getPrincipal());

System.out.println (s.getNoOfStaff());

System.out.println (s.getHeadMaster());

s.setPrincipal ("Chethna");

s.setNoOfStaff (10);

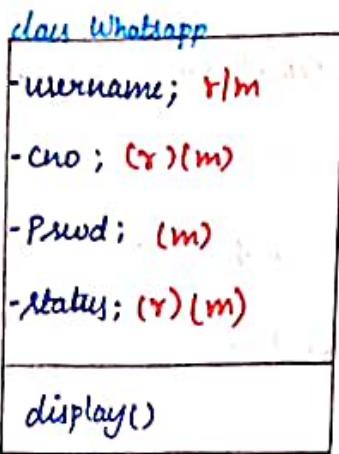
s.setHeadMaster ("Eshwar");

s.schoolDetails();

}

{

Example:



```
class Whatsapp
{
    private String username;
    private long cno;
    private String pwd;
    private String status;

    Whatsapp()
    {}

    Whatsapp(String username)
    {
        this.username = username;
    }

    Whatsapp(String username, long cno)
    {
        this.username = username;
        this.cno = cno;
    }

    Whatsapp(String username, long cno, String pwd, String status)
    {
        this(username, cno);
        this.pwd = pwd;
        this.status = status;
    }
}
```

```
public void getUsername()
{
    return username;
}

public String setUsername(String username)
{
    this.username = username;
}

public void setCno()
{
    return cno;
}

public long setCno(long cno)
{
    this.cno = cno;
}

public void setPwrd()
{
    this.pwrd = pwrd;
}

public void getStatus()
{
    return status;
}

public String setStatus(String status)
{
    this.status = status;
}

public void display()
{
    System.out.println("The username is " + username);
    System.out.println("status");
}
```

```

class WhatsappDriver
{
    public static void main(String [] args)
    {
        Whatsapp w = new Whatsapp("Manju", 9872536173, "manju@91",
                                    "I am busy");
        w.display();
        w.setStatus("call me later");
        w.setUsername("Manju ES");
        w.display();
    }
}

```

#### OUTPUT :

The username is Manju

I am busy

The username is Manju ES

Call me later

bean class:

The class where is having ~~private members~~ with ~~getters~~  
and ~~setters~~.

The no argument constructor ~~should be~~ ~~be~~ ~~new~~

It should implement ~~initialization~~ ~~interface~~

~~(initiate \*)~~ ~~attaching classmate~~

# Relationship

## Relationship:

The connection (Association) between two objects is known as the relationship.

## Types of Relationship:

- \* has-a- Relationship
- \* Is-a- Relationship

has-a- Relationship: one object having part of another object is known as has-a.

\* If one object is dependent on another object it is known as a has-a-relationship.

\* Based on the level of dependency has-a relationship is classified into two types.

- \* aggregation (weakly dependent)
- \* composition (strongly dependent)

## Aggregation:

The dependency between two objects such that one object can exist without the other is known as aggregation.

eg:

cab-ola, Train - Online Ticket Booking, Bus - Passenger, etc., coffee - cup, bat - ball, mobile - sim.

## Composition:

The dependency between two objects can't exist without the other is known as composition.

eg:

car - engine, Human - oxygen, etc., laptop - battery, remote - battery, mobile - battery,

## Advantage:

one object uses the features of another object.

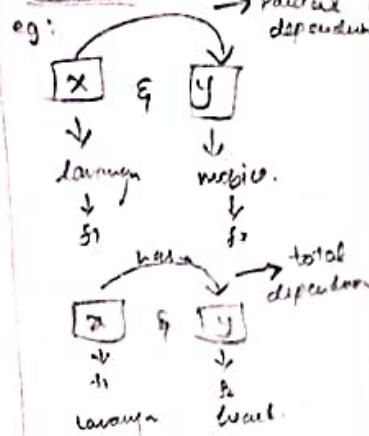
eg: Relationship

ola - application

↓  
cab

ola can't exist without cab  
cab can exist without ola

has-a



(i) diff composition & inheritance

Incomposition one object uses another object as a part

Inheritance one object owns another object as a part.

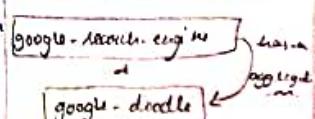
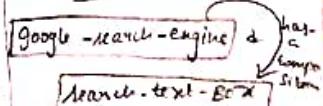
## Has-a

one object dependent on another object to use the features of that object.

## Is-a

one object belongs to another object to use the features of the object

## examples of has-a



## cardinality relation

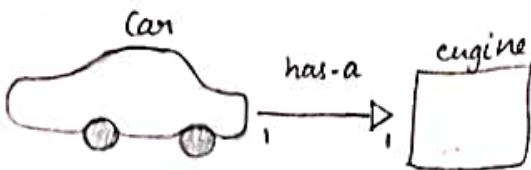
one object dependent on one object i.e., 1:1

eg: car - engine

one object dependent on many objects i.e., 1:m

eg: note - page

## Explanation for composition



- 1) Car can't exist without engine so it is composition
  - 2) one car have one engine so it is 1:1
  - 3) In real life engine is present inside the car  
In Java we have to design engine inside the car
  - 4) first we have to create engine object
- ```
class Engine { }
```
- 5) Second we have to create car object.

```
class Car { }
```

- 6) Now both doesn't have relationship. So, how have to built the relationship between these two objects. how?

by creating the reference variable of Engine type inside the car class.

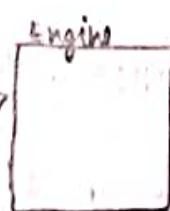
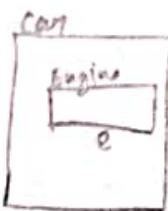
```
class Car {
```

```
    Engine e;
```

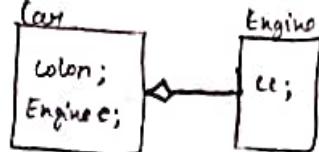
```
}
```

```
class Engine { }
```

```
}
```



7) Let us consider as per the UML diagram.



design the class

class Car

}

String color;

Engine e = new Engine(1000); // early instantiation.

// declaration and initialization are done.

Car (String color)

}

this.color = color;

}

}

class Engine

{

int cc;

Engine (int cc)

{

this.cc = cc;

}

}

class CarDriver

{

public static void main (String [] args)

{

Car c1 = new Car("red");

c1.show();

}

}

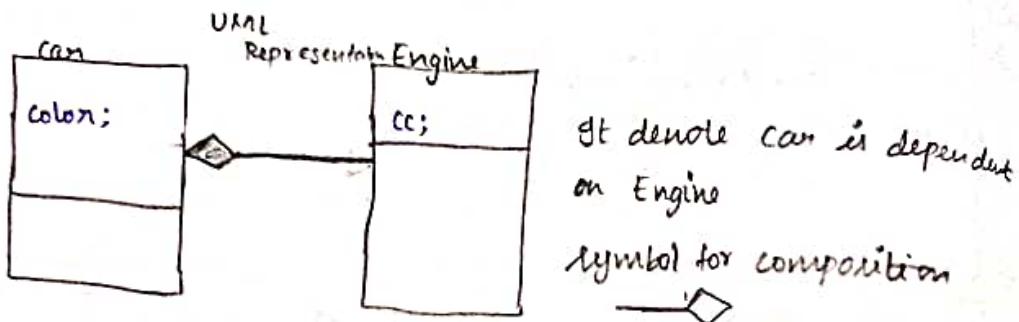
In the above CarDriver class we are creating object for Car but internally the Engine object also created. i.e., when the Car class parameterized constructor is called the constructor will load all NSM. therefore it load non static members (color and e) then it execute NSI. In Car class we have one non-static initializer (single line) so that will execute. which means it create an object for Engine class.

Finally we can say that whenever Car object is created Engine object is also created. This technique is called early instantiation.

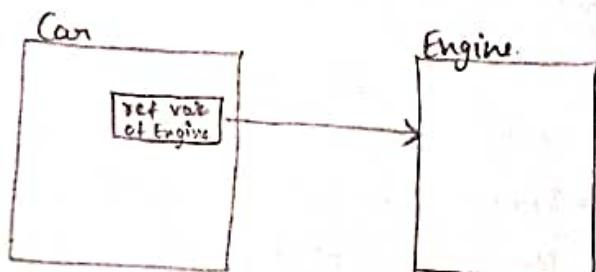
early instantiation technique is used for composite relationship.

\* In java we can achieve has a relationship by creating the reference variable of one object inside another object.

Eg: It has consider two objects car and engine, where the car object is dependent on engine object



\* In java this can be achieved such that side the car object we should have the reference variable of Engine type



\* The above design can be achieved by creating a non-static variable of Engine type inside the Car class

\* The instance of Engine object can be created in two different ways. by using different design techniques

- \* Early instantiation
- \* lazy instantiation

## Early Initialization

In this design technique the instance (object) of the dependent object is implicitly created.

eg:

when a car's user creates the instance of car, the instance of Engine is implicitly created.

NOTE :

we can achieve this design with the help of initializers

example:

```
class Engine
{
    private double cc; → 1000
    // getter
    public double getCC()
    {
        return cc;
    }
    // setter
    public void setCC(double cc)
    {
        this.cc = cc;
    }
    // no argument constructor
    Engine()
    {
    }
    // parameterized constructor
    Engine(double cc)
    {
        this.cc = cc;
    }
}
```

```

class Car
{
    // creating object by using a method because
    // creating object is also one task.
    // It is not mandatory but it is good way
    // design.

    public static Car createCar()
    {
        return new Car();
    }

    public static Car createCar(String color)
    {
        return new Car(color);
    }

    private String color; // Private non-static variable color of
    // String type
    // getter
    public String getColor()
    {
        return color;
    }

    // setter
    public void setColor(String color)
    {
        this.color = color;
    }

    // has-a -> Early instantiation
    private Engine e = new Engine(1000); // Private
    // non-static variable e of Engine
    // type
    // getter
    public Engine getEngine()
    {
        return e;
    }
}

```

(car1) // default constructor

{ }

→ Car (String color) // parameterized constructor.

```
{  
    this.color = color; //  
}
```

}

class CarDriver

{

public static void main (String [] args)

{

Car c1 = Car.createCar ("Black");

System.out.println ("color :" + c1.getColor());

System.out.println ("cc :" + c1.getEngine ().getCC());

parameterized  
method in car class.

or

or 2

}

}

Black

1000

another primary constructor does this records (data)

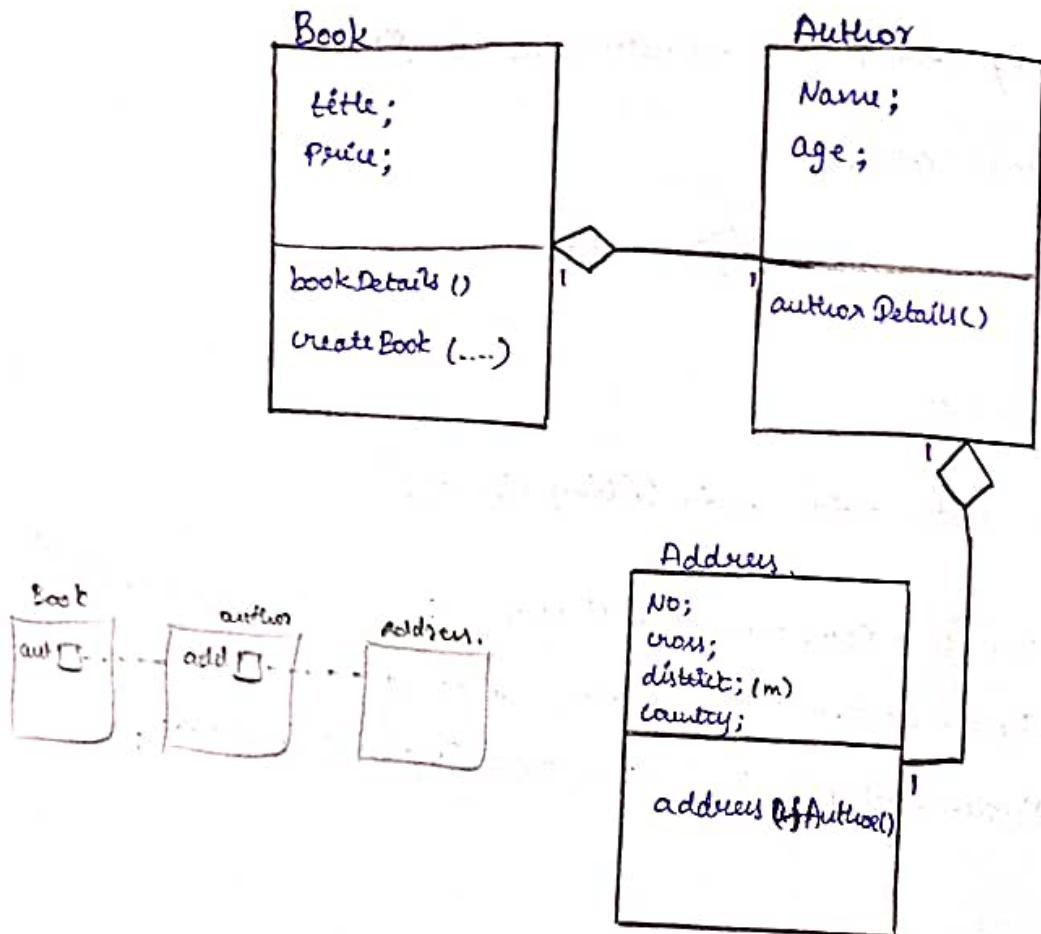
OUTPUT :

color : Black  
cc : 1000

Conclusion :

when the car object is created the engine object is also created implicitly.  
Hence this process is known as early instantiation.

## Example 2:



Create driver class and perform the following actions

1. Create a book object
2. display the address of the author
3. display the author details
4. display the book details
5. change the district of the author to bangalore
6. display the updated district.

```
public class Address
```

```
{ // states
```

```
private int no;
```

```
private String cross;
```

```
private String district;
```

```
private String country;
```

```
// getters
```

```
public int getNo() {
```

```
    return no;
```

```
}
```

```
public String getCross() {
```

```
    return cross;
```

```
}
```

```
public String getDistrict() {
```

```
    return district;
```

```
}
```

```
public String getCountry() {
```

```
    return country;
```

```
}
```

```
// setters
```

```
public void setNo(int no) {
```

```
{
```

```
    this.no = no;
```

```
}
```

```
public void setCross(String cross) {
```

```
{
```

```
    this.cross = cross;
```

```
}
```

```
public void setDistrict(String district) {
```

```
{
```

```
    this.district = district;
```

```
}
```

```
public void setCountry(String country) {
```

```
{
```

```
    this.country = country;
```

```
}
```

//constructor:

Address ()

{ }

Address ( int no, String cross, String district, String country )  
{ }

this.no = no;

this.cross = cross;

this.district = district;

this.country = country;

}

//behaviour

public void addressOfAuthor()

{

System.out.println("Number :" + no);

System.out.println("cross :" + cross);

System.out.println("District :" + district);

System.out.println("Country :" + country);

}

}

---

class Author {

private String name;

private int age;

private Address address;

//getters

public String getName()

{

return name;

}

```
public int getAge()
{
    return age;
}

public Address getAddress()
{
    return address;
}

// setters

public void setName(String name)
{
    this.name = name;
}

public void setAge(int age)
{
    this.age = age;
}

public void setAddress(Address address)
{
    this.address = address;
}

// constructor

Author()
{
}

Author(String name, int age)
{
    this.name = name;
    this.age = age;
}
```

## // behaviour

```
public void authorDetails() {  
    System.out.println("name: " + name);  
    System.out.println("age: " + age);  
    if (address != null)  
    {  
        address.address();  
    }  
    else  
        System.out.println("address not yet initialised...");  
}  
}  
}
```

---

## class Book

```
{  
    // State  
    private String title; java code  
    private double price; java code  
    private Author author; author @ mm
```

## // getters

```
public String getTitle()  
{  
    return title;  
}  
public double getPrice()  
{  
    return price;  
}  
public Author getAuthor()  
{  
    return author;  
}
```

### // setter

```
public void setTitle (String title)
```

```
{  
    this.title = title;  
}
```

```
public void setPrice (double price)
```

```
{  
    this.price = price;  
}
```

```
public void setAuthor (Author author) <----- 2nd line.
```

```
{  
    this.author = author;  
}
```

### // constructor

```
Book ()
```

```
{ }
```

```
Book (String title, double price) <----- 1st line.
```

```
{  
    this.title = title;  
}
```

```
    this.price = price;  
}
```

### // behaviour

```
public void bookDetails ()
```

```
{  
    System.out.println ("***** Book Details *****");
```

```
    System.out.println ("title : " + title);
```

```
    System.out.println ("price : " + price);
```

```
    if (author != null)
```

```
    {  
        author.authorDetails ();
```

```
    }
```

```
    else
```

```
        System.out.println ("author not yet initialized");
```

```
}
```

```

class BookDriven {
    public static void main (String [] args) {
        Book b1 = new Book("Java", 1500);
        b1.setAuthor(new Author("James Gosling", 65));
        System.out.println(b1.getAuthor().getAddress());
        System.out.println(b1.getBookDetails());
    }
}

```

### OUTPUT :

\* \* \* \* \* Book Details \* \* \* \* \*

title : Java

price : 1500

name : James Gosling

age : 65

Number : 1

Cross : 15<sup>th</sup> Cross

District : TN

Country : India

### Conclusion:

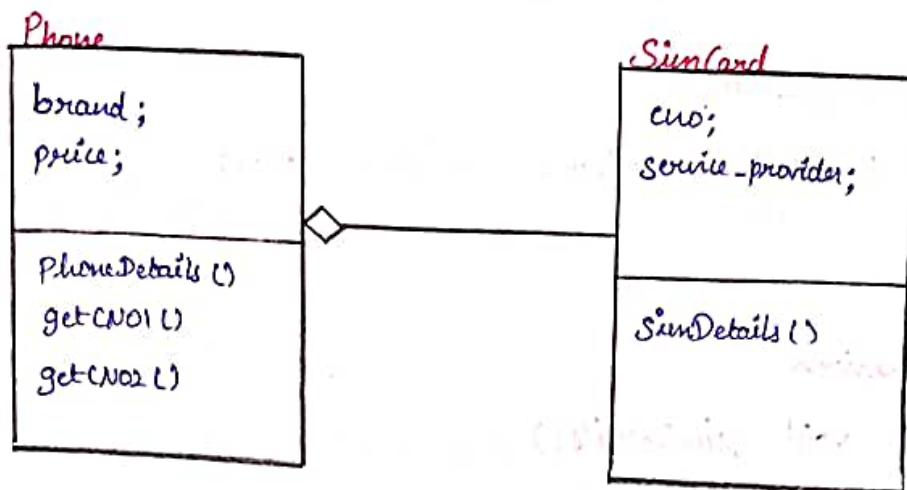
when the book object is created only one object is created.

if the user want to create object for author or address they can explicitly create an object by calling the helper method.

This process is known as lazy instantiation.

Example:

### Aggregation



```
class SimCard
{
    private long cno;
    private String service-provider;

    // getters
    public long getCNO()
    {
        return cno;
    }
    public String getServiceProvider()
    {
        return service-provider;
    }

    // setters
    public void setServiceProvider(String service-provider)
    {
        this.service-provider = service-provider;
    }

    // constructor
    SimCard()
    {
    }
```

SimCard (long cno, String service-provider)

{

    this.cno = cno;

    this.service-provider = service-provider;

}

// behavior

    public void simDetails()

    {

        System.out.println("cno : " + cno);

        System.out.println("Service Provider : " + service-provider);

    }

}

class Phone

{

// state

    private String brand;

    private double price;

    private SimCard sim1;

    private SimCard sim2;

// getters

    public String getBrand()

    {

        return brand;

    }

    public double getPrice()

    {

        return price;

    }

IN

```
public SimCard getSim1()
{
    return sim1;
}

public SimCard getSim2()
{
    return sim2;
}
```

### // setters

```
public void setBrand(String brand)
```

```
{
    this.brand = brand;
}
```

```
public void setPrice(double price)
```

```
{
    this.price = price;
}
```

```
public void setSim1(SimCard sim1)
```

```
{
    this.sim1 = sim1;
}
```

```
public void setSim2(SimCard sim2)
```

```
{
    this.sim2 = sim2;
}
```

### // constructor

```
Phone()
```

```
{ }
```

```
Phone(String brand, double price)
```

```
{
    this.brand = brand;
    this.price = price;
}
```

## //behaviour

```
public long getENO1()
{
    return sim1.getENO();
}
public long getENO2()
{
    return sim2.getENO();
}
public void phoneDetails()
{
    System.out.println("Brand : " + brand);
    System.out.println("Price : " + price);
    if (sim1 != null)
    {
        sim1.simDetails();
    }
    else
        System.out.println("The sim1 is not inserted");
    if (sim2 != null)
    {
        sim2.simDetails();
    }
    else
        System.out.println("The sim2 is not inserted");
}
```

class PhoneDriver

{

public static void main (String [] args)

{

Phone p = new Phone ("vivo", 26000);

p.setSim1(new simCard (9876540321, "Airtel"));

p.setSim2 (new simCard(012345-7896, "Airtel"));

p.getSim2 ().setServiceProvider ("JIO");

p.phoneDetails ();

p.getSim1 ().simDetails ();

p.getSim2 ().simDetails ();

}

}

OUTPUT:

Brand : vivo

Price : 26000

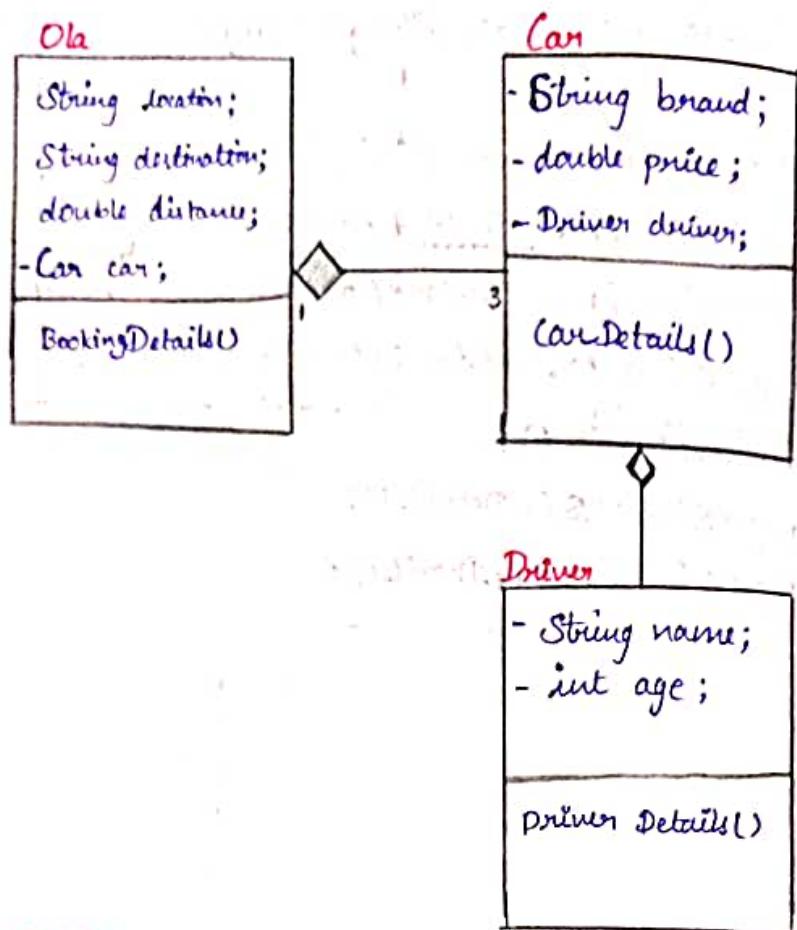
CNO : 9876540321

Service provider : Airtel

CNO : 0123457896

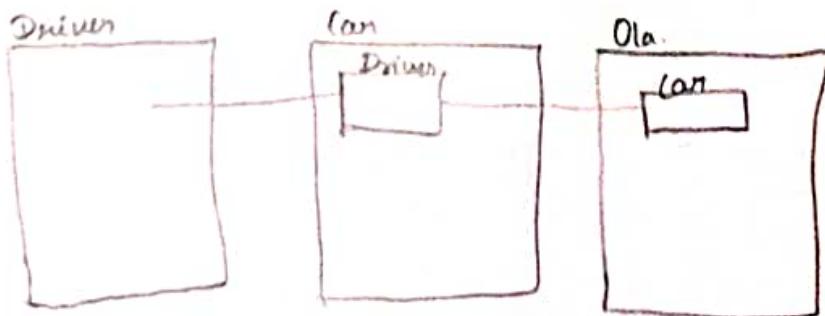
Service provider : JIO

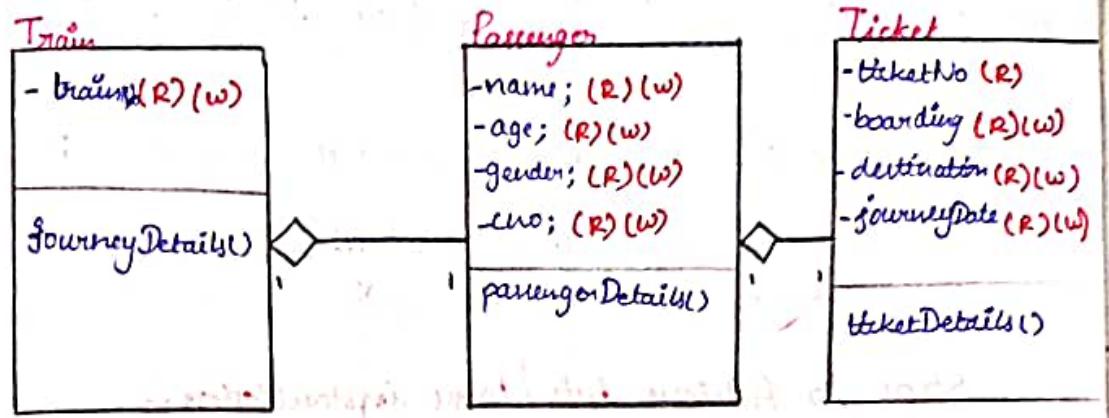
9-4-22



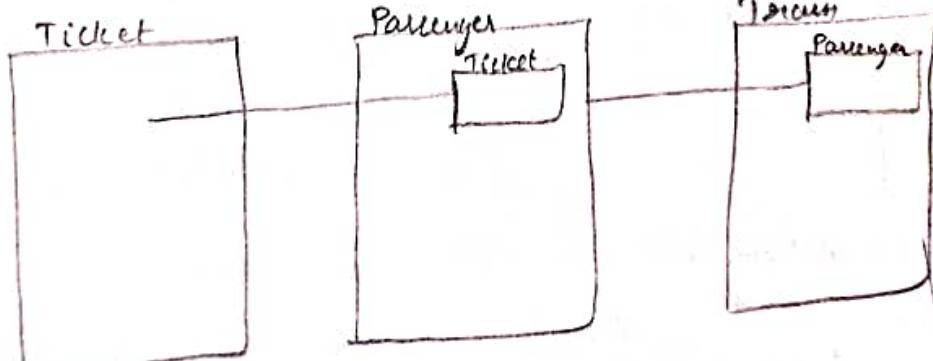
### OlaDriver

Add the car  
show the Driver detail  
display the car detail  
display the Booking detail  
change the destination  
display the updated details



**TrainDriver**

- addPassenger
- bookTicket
- cancelTicket
- change Destination
- changeBoarding
- Exit.



## Lazy / Late Initialization:

In this design the instance of dependent object is created only when it is required (it is not implicitly created)

We can achieve this design with the help of method, it can be called as an helper method.

## Steps to Achieve late / Lazy instantiation:-

Step 1: Create a dependent class

Step 2: Create another class and define a parameterized method that will accept the reference of the dependent object and inside that method initialize the dependent object

Step 3: Create the object for a class and call a method by passing dependent type object reference so that, we can achieve a late / lazy instantiation

11/4/22

## Is - a - Relationship :

\* The association between two objects similar to parent and child is known as is - a - relationship.

### NOTE:

Both the objects belongs to the same type. hence can we called as a family (hierarchy)

\* In is-a Relationship the child object will have all the properties of the parent object and extra properties of child object

\* Therefore, the parent can be a generalized type and child can be a specialized type.

### Example 1:

Doctor - Cardiologist

→ Cardiologist is also an doctor. therefore, Both are belongs to same family.

→ Cardiologist have all the features and properties of doctor along with some extra features.

### Example 2:

Web application - FaceBook

→ FaceBook is a type of Web Application. Therefore Both are belongs to same family

→ FaceBook have all the features and properties of web application along with some extra features.

### Example 3:

Fruit - Apple

→ Apple is a type of fruit. therefore both are belongs to same family.

→ Apple have all the features of fruit along with some extra features.

54

**NOTE :**

IS-a Relationship can be achieved with the help of inheritance

**Inheritance :**

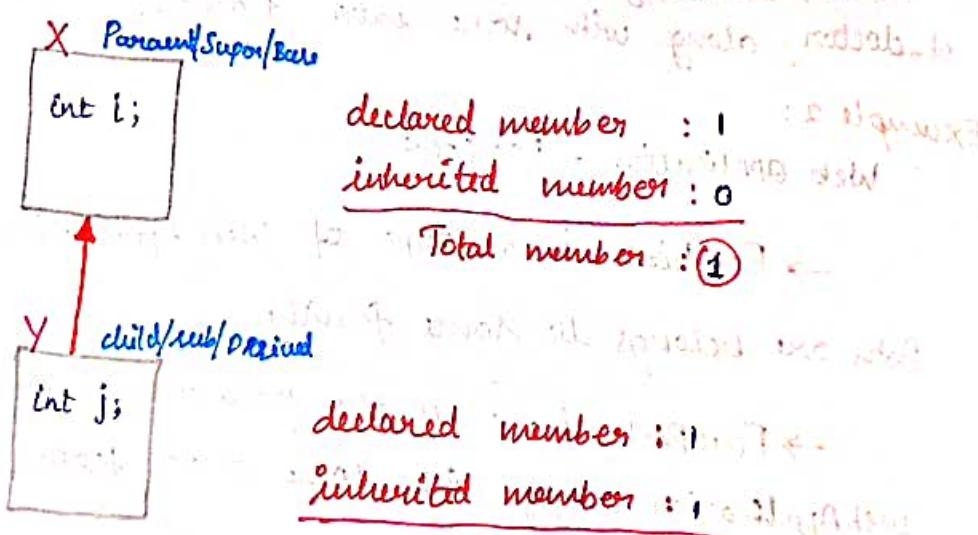
\* It is the process of obtaining / acquiring all the properties of one object into another object

**NOTE :**

The object which is receiving the properties is termed as child / sub / derived class

\* The object which is providing the properties is known as parent / super / base class

**Example :**



## How to achieve inheritance in java?

We can achieve inheritance in Java by using the keyword called **extends** or implements.

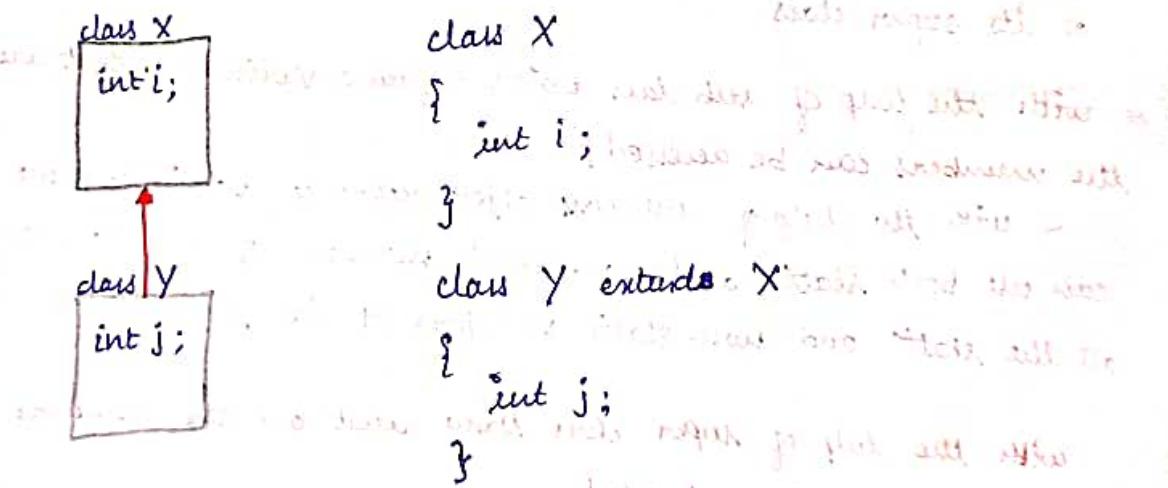
### **extends :**

- \* It is a keyword in Java.
- \* It is used to achieve inheritance between two classes or between two interfaces.

### **Using extends for Two classes:**

The **extends** keyword can be used only with the child class.

**Syntax:** To achieve inheritance between two classes.



**Generalized Syntax:** for inheritance between two classes

```
class ChildClass-Name extends ParentClass-Name
{
}
}
```

What are the members are inherited between two classes?

\* Except private members and initializers of the class all the other members of the class are inherited.

\* Initializers includes static initializer and non-static initializer and constructors.

#### NOTE:

With the help of subclass Name what are the members can be accessed?

With the help of sub class name we can use all the static members of sub class and all the static members of its super class.

With the help of sub class object reference variable what are the members can be accessed?

With the help of sub class object reference variable, we can use both static and non-static members of sub class and all the static and non-static members of its super class.

With the help of super class Name what are the members can not be accessed?

With the help of super class Name we can not use the static members of its sub class.

With the help of super class object reference what are the members can not be accessed?

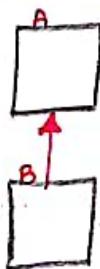
With the help of super class object reference we can not use the members of its sub class.

## Types Of Inheritance

- \* Single Level Inheritance
- \* Multi level Inheritance
- \* Hierarchical Inheritance
- \* Multiple Inheritance
- \* Hybrid Inheritance

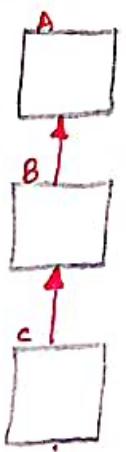
### Single Level Inheritance

Inheritance has only one level is known as single level inheritance



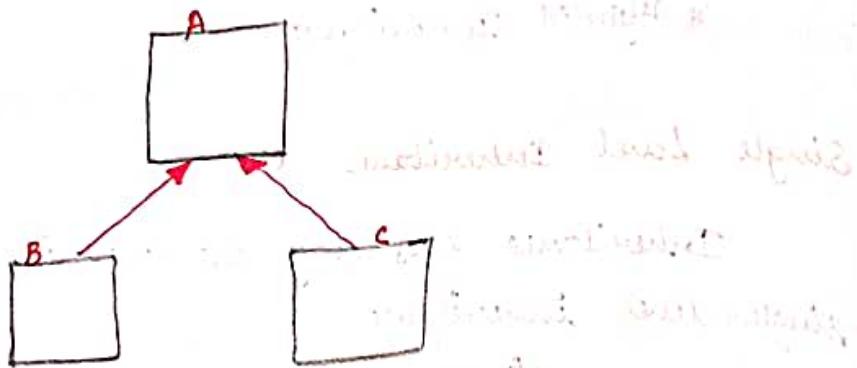
### Multi Level Inheritance

Inheritance of more than one level is known as multi level inheritance



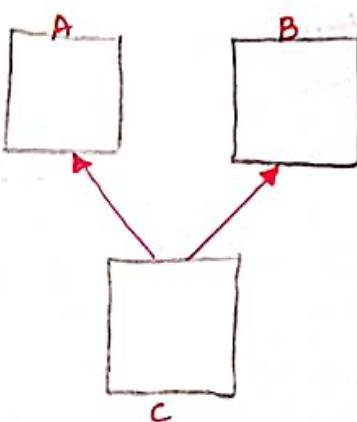
## Hierarchical Inheritance:

If a parent (super class) has more than one child (sub class) at the same level then it is known as hierarchical inheritance.



## Multiple Inheritance:

If a subclass (child) inherits more than one parent (super class) then it is known as multiple inheritance.



### NOTE:

Multiple inheritance has a problem known as the diamond problem.

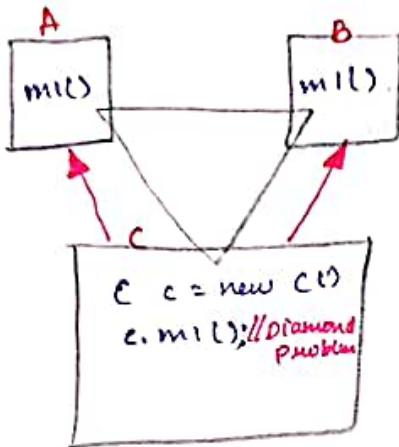
Because of the diamond problem, we can't achieve multiple inheritance with the help of class.

In java, we can achieve multiple inheritance with the help of interface.

## Diamond Problem:

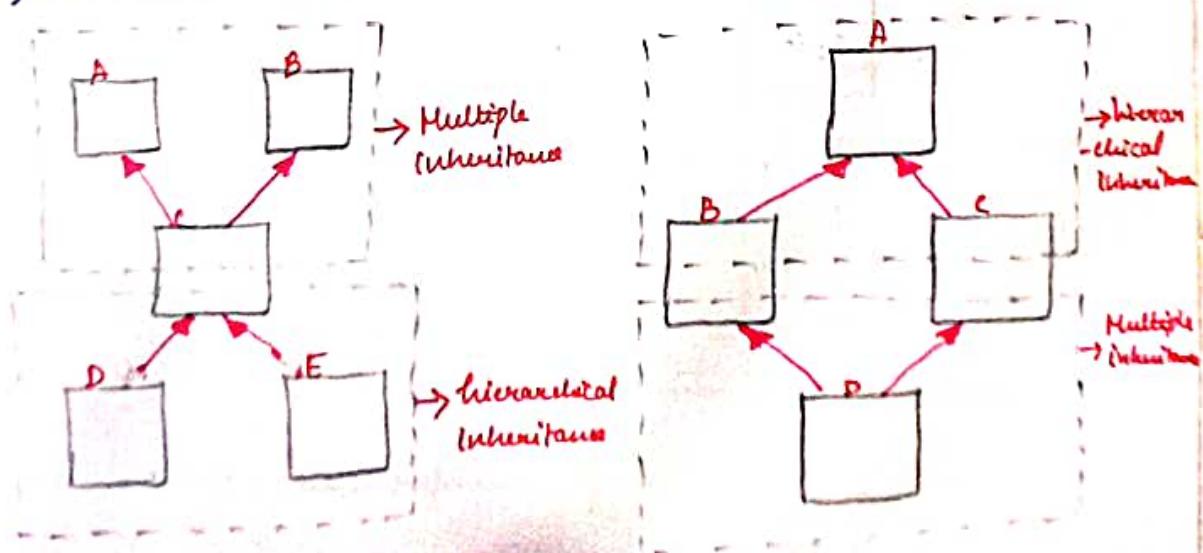
Assume that there are two classes A and B. Both are having the method with the same signature. If class C inherits A and B then both two methods are inherited to C.

Now an ambiguity arises when we try to call the super class method with the help of subclass reference. This problem is known as the Diamond Problem.



## Hybrid Inheritance:

The combination of multiple inheritance and hierarchical inheritance is known as hybrid inheritance.



### University

UniversityName; (E)(W)

UniversityNo; (R)(W)

ChancellorName; (E)(W)

UniversityDetails();

### Student

StudentName; (R)(W)

StudentId; (R)(W)

age; (R)(W)

gender; (R)

StudentDetails()

CS

departmentName;  
headName;

departmentDetails();

civil

departmentName;  
headName;

departmentDetails();

mech

departmentName;  
headName;

departmentDetails();

### Driver

Add Admission

Student details

Cancel admission

Exit.

12/4/22

## understanding OBJECT REFERENCE

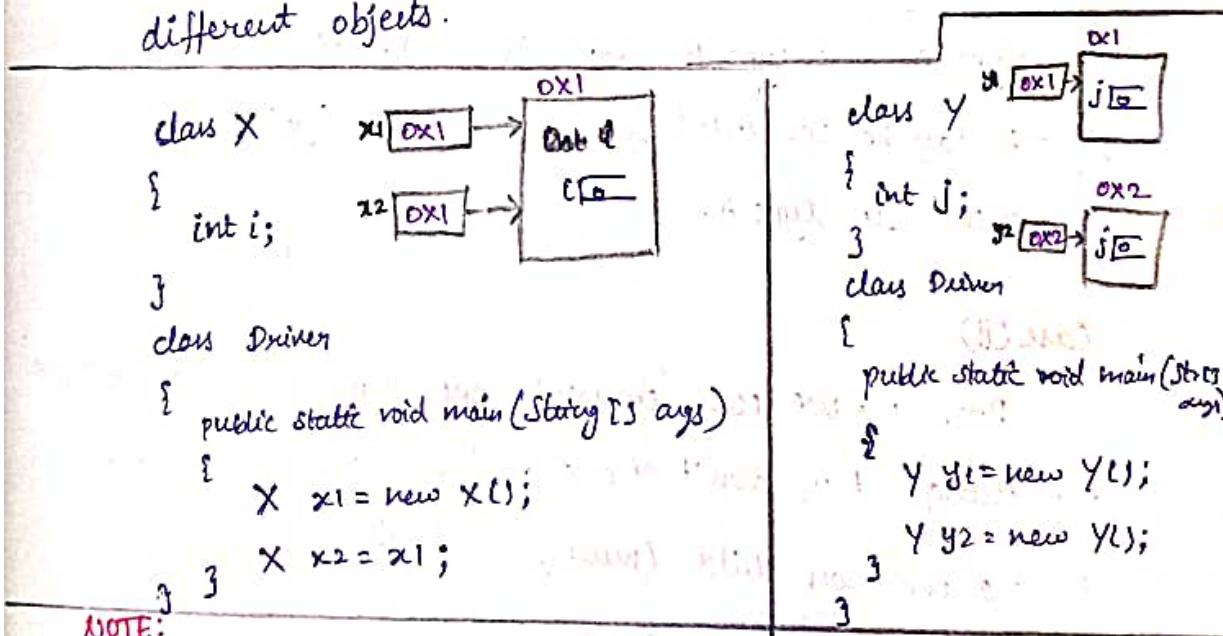
\* we can copy object reference from one reference variable to another reference variable if they belongs to the same family

\* An object can be referenced by any number of reference variables

\* We can verify, whether two reference variables are pointing to the same or different object with the help of equality operator

→ If the equality operator returns true, it means both the variables are pointing to the same object

→ If the equality operator returns false, it means both the variables are pointing to two different objects.



We can pass reference of an object to a method  
The method can return the reference of object

## Non-Primitive Type Casting / Derived Type Casting

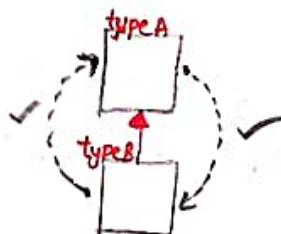
The process of converting one non-primitive data type into another non-primitive data type is known as non-primitive type casting.

### NOTE:

Non primitive type casting can be achieved only in the following situation.

#### Case(i)

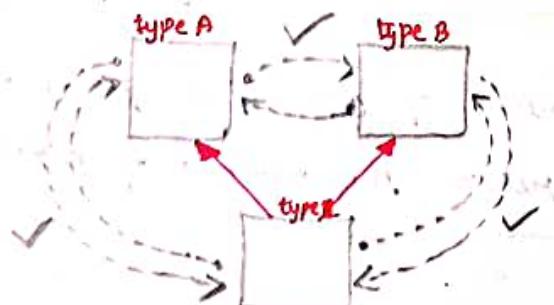
Both the types have is-a-Relationship



Type A and type B have is-a-Relationship. Therefore, type A can be converted to type B and type B can be converted to type A.

#### Case(ii)

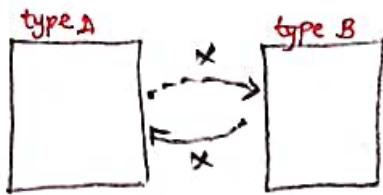
We can convert one type to another type even though they don't have a is-a-Relationship but, have a common child (multiple inheritance).



In the above scenario type A can be converted to type B, type B can be converted to type A even though they don't have is-a relationship. because, type A and type B has a common child type C.

**NOTE:**

We cannot convert one reference type to another reference type if they do not have is-a relationship or they are not parents of common child.

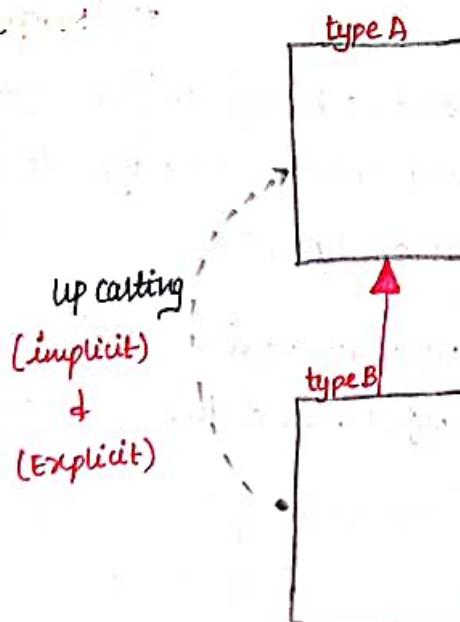


**Types of Non-Primitive type Casting:**

Non-primitive type casting is broadly classified into two types. They are,

\* up casting

\* down casting



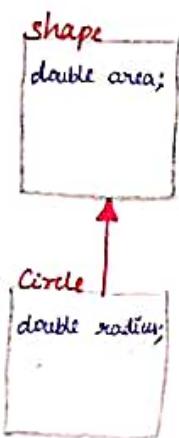
## Upcasting:

\* The process of converting sub class reference type into super class reference type is known as upcasting.

### NOTE:

upcasting can be done by the compiler as well as by the programmer.

### Example:



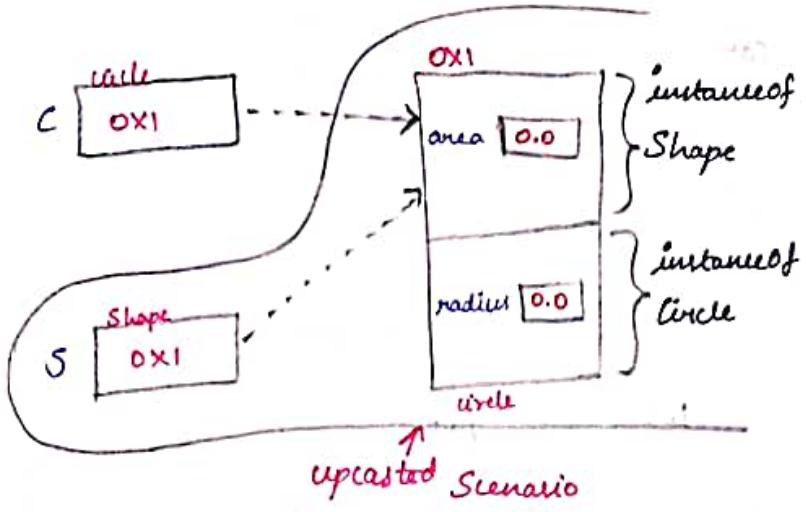
```
circle c = new Circle();
```

```
Shape s = c;
```

### NOTE:

C belongs to circle type and S belongs to Shape type, therefore compiler tries to convert compiler circle type to shape i.e child to parent is known as upcasting.

\* Hence, the parent type reference variable S is having the address of child type object. and this is known as upcasted scenario.



`System.out.println(c == s);`

O/p: `true`

\* It means variable `c` and `s` are having the address of same object, but they belong to different type.

\* With the help of `c` we can use members of `circle` as well as `shape`.

`System.out.println(c.area); //CTS`

`System.out.println(c.radius); //CTS`

O/p: `0.0  
0.0`

\* with the help of `s` we can only use members of `Shape` but, not the members of `circle`.

`System.out.println(s.area); //CTS`

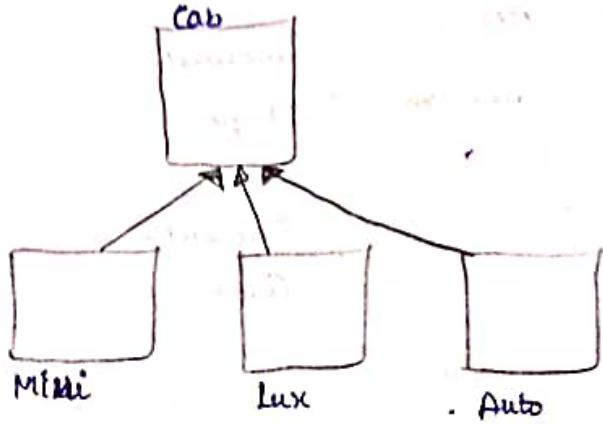
`System.out.println(s.radius); //CTS`

why upcasting? / advantages of upcasting:

Upcasting is used to achieve generalization. i.e., we can create a generalized variable of super class type which can store the reference of any object of its child type.

Eg: OLA

OLA:



case(i)

Lux l;

$l = \text{new Lux}();$  //CTS

$l = \text{new Mini}();$  //CTE

$l = \text{new Auto}();$  //CTE

$l$  is not a generalized variable, inside  $l$  we store only the reference of Lux object. we can't store the reference of Mini or Auto objects.

case (ii)

Cab c;

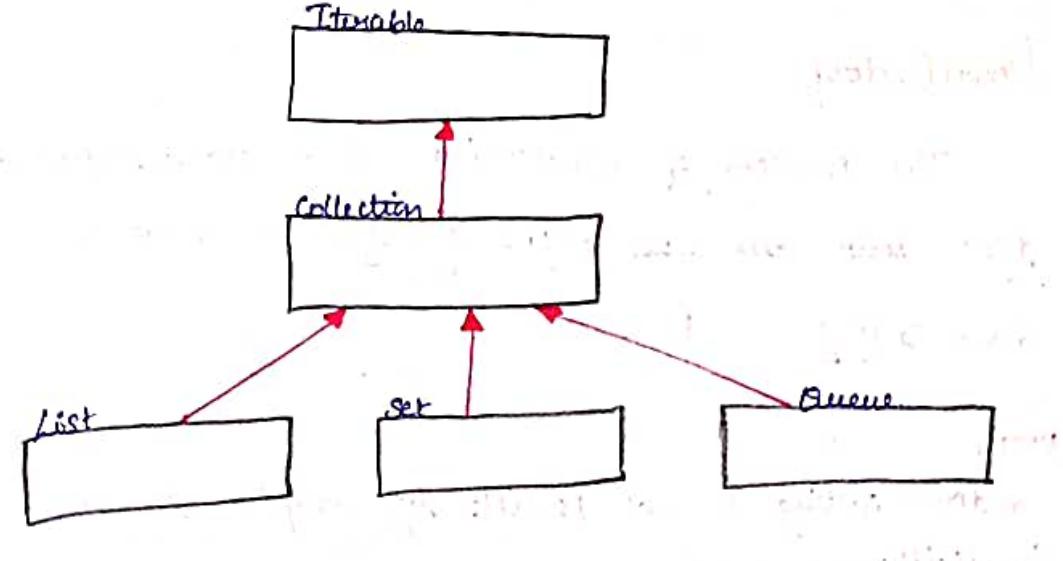
$c = \text{new Lux}();$  //CTS

$c = \text{new Mini}();$  //CTS

$c = \text{new Auto}();$  //CTS

$c$  is a generalized variable, has it belongs to super class Cab type. Therefore, we can store the reference of any of its sub / child object inside  $c$ .

This is possible only because of upcasting.



(case i)

```

List l;
l = new Set(); // CTE
l = new Queue(); // CTE
l = new List(); // CTS

```

*l* is not a generalized variable. Inside *l* we can store only the reference of List object. we can't store the reference of set or queue objects

(case ii)

```

Iterable i;
i = new Set(); // CTS
i = new Queue(); // CTS
i = new List(); // CTS
i = new Collection(); // CTS
i = new Iterable(); // CTS

```

*i* is a generalized variable, as it belongs to super class Iterable type. therefore, we can store the reference of any of its child object inside *i*.

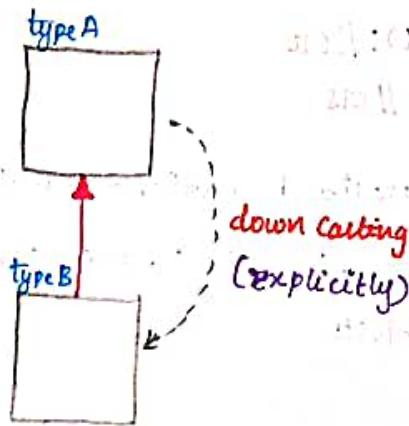
## Down Casting:

The process of converting Super class reference type into sub class reference type is known as down casting.

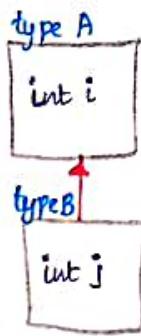
### NOTE:

\* down casting is not possible by compiler to do implicitly

\* down casting can be done only by the programmer explicitly with the help of type casting operators

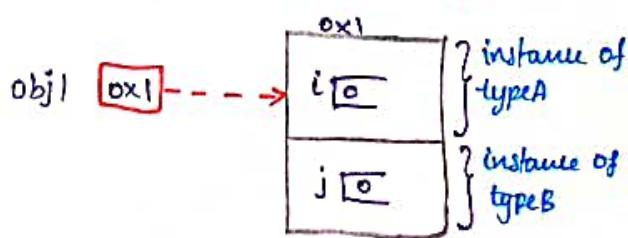


### example :



### Case 1:

```
typeA obj1 = new typeB(); // upcasting
```



class

```

System.out.println(obj1.i); //CTS
System.out.println((TypeB) obj1.j); //CTS
TypeB obj2 = (TypeB) obj1; //downcasting
System.out.println(obj2.j); //CTS

```

*both are same only*

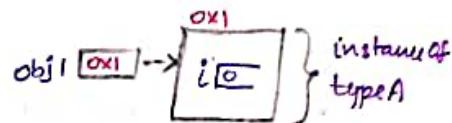
**Case 2:**

```

typeA obj1 = new typeAC();
System.out.println(obj1.i); System.out.println((TypeB) obj1.j);

```

Here we get an run time exception called **Class Cast Exception**



**Exception**

classCast Exception is an exception that occurs when the reference is down casted to a sub class type , and that object does not have instance of the subclass type.

**Example :**

considering case 2 the reference variable obj1 is down casted to to type B . the object refered by obj1 does not have instance of type B. Hence we get class Exception.

## SUPER() CALL STATEMENT

### Super() CALL STATEMENT:

- \* Super is a keyword, it is used to access the members of the superclass.
- \* Super() call statement is used to call the constructor of the parent class from the child class constructor.

### Purpose of super() statement :

- \* When the object is created, the super call statement helps to load the nonstatic members of the parent class into the child object.
- \* We can also use the super() call statement to pass the data from the subclass to the parent class.

### Rules to use super() statement :

- \* Super() call statement should always be the first instruction in the constructor call.
- \* If a programmer doesn't use the super() call statement, then the compiler will add a no-argumented super call statement into the constructor body.

| <u>this()</u>                                                   | <u>super()</u>                                                         |
|-----------------------------------------------------------------|------------------------------------------------------------------------|
| * It is used to call the constructor of the same class.         | * It is used to call the constructor of the parent class (super class) |
| * It is used to represent the instance of child class.          | * It is used to represent the instance of parent class                 |
| * It should be used as a first statement in a constructor body. | * It should be used as a first statement in a constructor body.        |

Example: Here child class only initializing the parent class member

```

class Vehicle
{
    int noOfWheels;
    Vehicle()
    {
        noOfWheels = 4;
    }
}

```



class Bike extends Vehicle

```

{
    boolean kick;
    Bike(boolean kick, int noOfWheels)
    {
        super();
        this.kick = kick;
        this.noOfWheels = noOfWheels;
    }
}

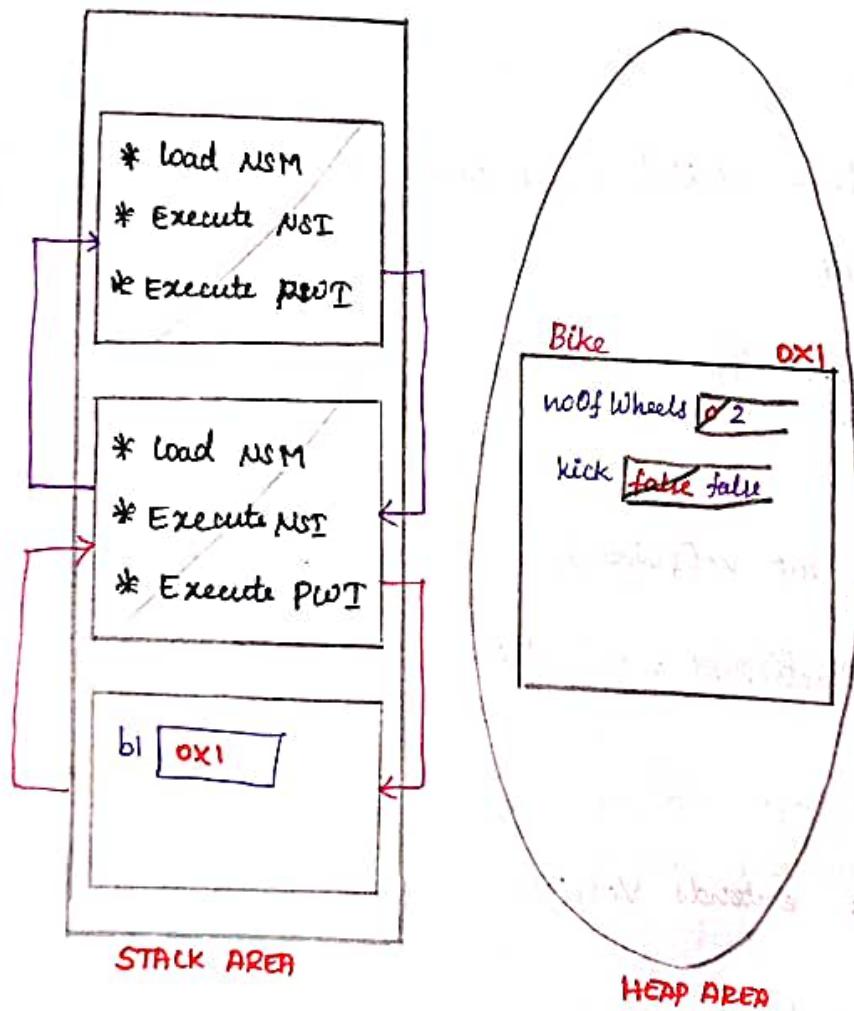
```

DATA STATE

```

class BikeDriver
{
    public static void main(String[] args)
    {
        Bike b1 = new Bike(false, 2);
    }
}

```



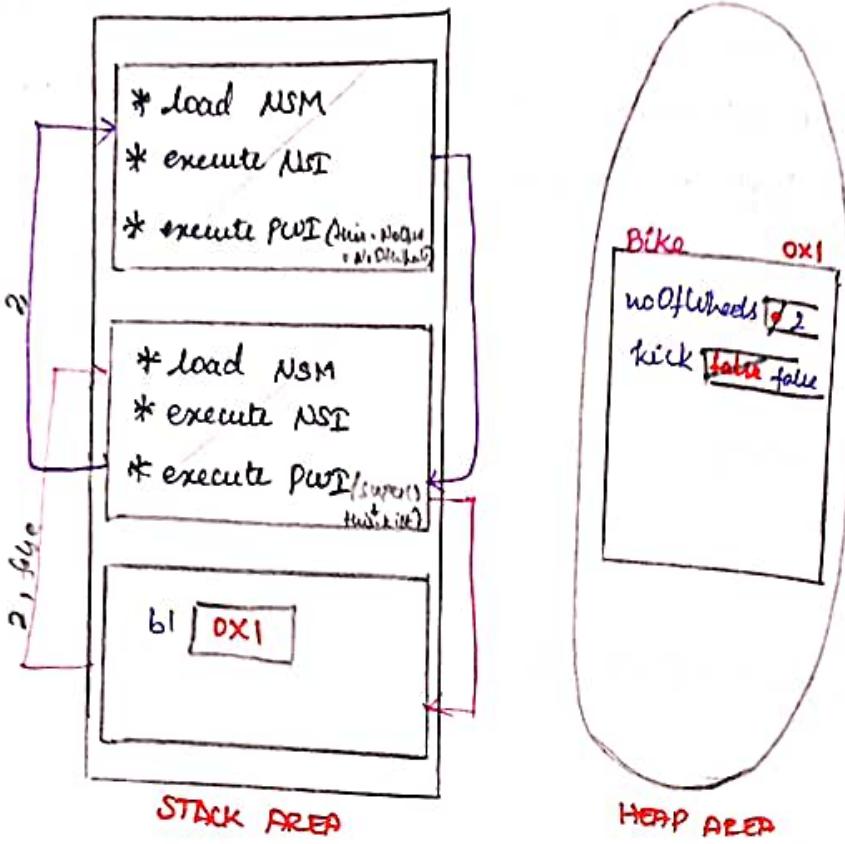
Example: Here the parent class constructor initialize its member and child class initialize its member (no changes between these two examples, just to improve the efficiency).

```
class Vehicle
{
    int noOfWheels;
    Vehicle()
    {
        noOfWheels = 4;
    }
}

Vehicle (int noOfWheels)
{
    this.noOfWheels = noOfWheels;
}

class Bike extends Vehicle
{
    boolean kick;
    Bike(boolean kick, int noOfWheels)
    {
        super(noOfWheels); // call the parameterized constructor of
        this.kick = kick;   parent class.
    }
}

class BikeDriver
{
    public static void main (String [] args)
    {
        Bike b1 = new Bike (false, 2);
    }
}
```



### Example :

```
class A
```

```
{
```

```
    int i;
```

```
    int j;
```

```
    A( int i, int j )
```

```
{
```

```
    this.i = i;
```

```
    this.j = j;
```

```
}
```

```
}
```

```
class B extends A
```

```
{
```

```
    int k;
```

```
    B( int i, int j, int k )
```

```
{
```

```
    super(i,j);
```

```
    this.k = k;
```

```
}
```

```
}
```

```
class Driver
```

```
{
```

```
    public static void main( String [ ] args )
```

```
{
```

```
        B b = new B( 1, 2, 3 );
```

```
}
```

```
}
```

### Example:

```
class Watch
{
    String type;
    String color;
    double price;
    Watch()
    {
    }
    Watch(String type)
    {
        this.type = type;
    }
    Watch(String type, String color)
    {
        this(type);
        this.color = color;
    }
    Watch(String type, String color, double price)
    {
        this(type, color);
        this.price = price;
    }
}

class Cario extends Watch
{
    String brand;
    Cario(String type, String color, double price, String brand)
    {
        super(type, color, price);
        this.brand = brand;
    }
}
```

The diagram illustrates a class hierarchy. At the top is a box labeled 'Watch' containing three fields: 'type', 'color', and 'price'. An arrow points from this box down to another box labeled 'Cario', which contains a single field 'brand'. This visualizes how 'Cario' inherits from 'Watch'.

### Example :

```
class Mobile  
{  
    String brand;  
    String color;  
    double price;  
    Mobile()  
    {}
```

```
Mobile (String brand)
```

```
{  
    this.brand = brand;  
}
```

```
Mobile (String brand, String color, double price)  
{
```

```
    this.brand = brand;
```

```
    this.color = color;
```

```
    this.price = price;
```

```
}
```

```
3
```

```
class Redmi extends Mobile
```

```
{
```

```
    boolean miRemote;
```

```
    Redmi()
```

```
{}
```

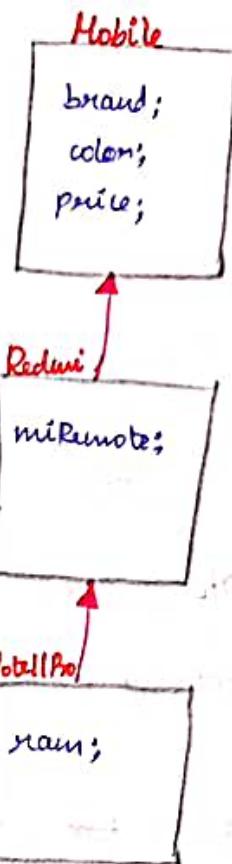
```
    Redmi (String brand, String color, double price, boolean miRemote)  
{
```

```
        super (brand, color, price);
```

```
        this.miRemote = miRemote;
```

```
}
```

```
3
```



```
class Note11Pro extends Redmi
{
    int ram;
    Note11Pro (String brand, String color, double price, boolean milRemote,
               int ram)
    {
        super (brand, color, price, milRemote);
        this.ram = ram;
    }
}
```

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

## Polyorphism:

Polyorphism is derived from two different Greek words 'poly' means many 'morphis' means form which means Many form. polyorphism is the ability of an object to exhibit more than one form with the same name.

## for understanding:

One name  $\Rightarrow$  multiple forms

One variable name  $\Rightarrow$  Different values

One method name  $\Rightarrow$  Different behaviours

## Types Of Polyorphism :

In java we have two types of polyorphism.

(i) compile Time Polyorphism / static polyorphism

(ii) Run Time Polyorphism / dynamic polyorphism

## Compile Time Polyorphism

If the binding is achieved at the compile-time and the same behaviour is executed . it is known as compile-time polyorphism.

### NOTE :

Association of method call to method definition is called as Binding

compile Time polymorphism is achieved through

- \* Method overloading
- \* Constructor overloading
- \* Variable shadowing
- \* Method shadowing
- \* Operator overloading (does not support in Java)

### Run Time Polymorphism

If the binding is achieved at the compile time and different behaviour is executed during runtime is known as run time polymorphism.

It is also known as dynamic binding

It is achieved by method overriding

### method overriding:

If the sub class and super class have the non-static method with same signature. It is known as method overriding.

### Rules to achieve method overriding:

- \* Is-A relationship is mandatory
- \* It is applicable only for non static methods.
- \* The signature of the subclass method and super class method should be same.
- \* The return type of the subclass and superclass method should be same until 1.4 version, but, from 1.5 version co-variant return type in overriding method is acceptable (sub class return type should be same or child to the parent class return type).

## Program : (before overriding)

```
class RajaFather
{
    public void marry()
    {
        System.out.println("Raja will marry Meela");
    }
}
```

```
class Raja extends RajaFather
{
}
```

```
class RajaDriver
{
    public static void main(String[] args)
    {
        RajaFather r = new Raja(); // upcasting
        r.marry(); // Parent class method will be called
    }
}
```

## Program : (After overriding)

```
class RajaFather
{
    public void marry()
    {
        System.out.println("Raja will marry Shela");
    }
}
```

class Raja extends RajaFather

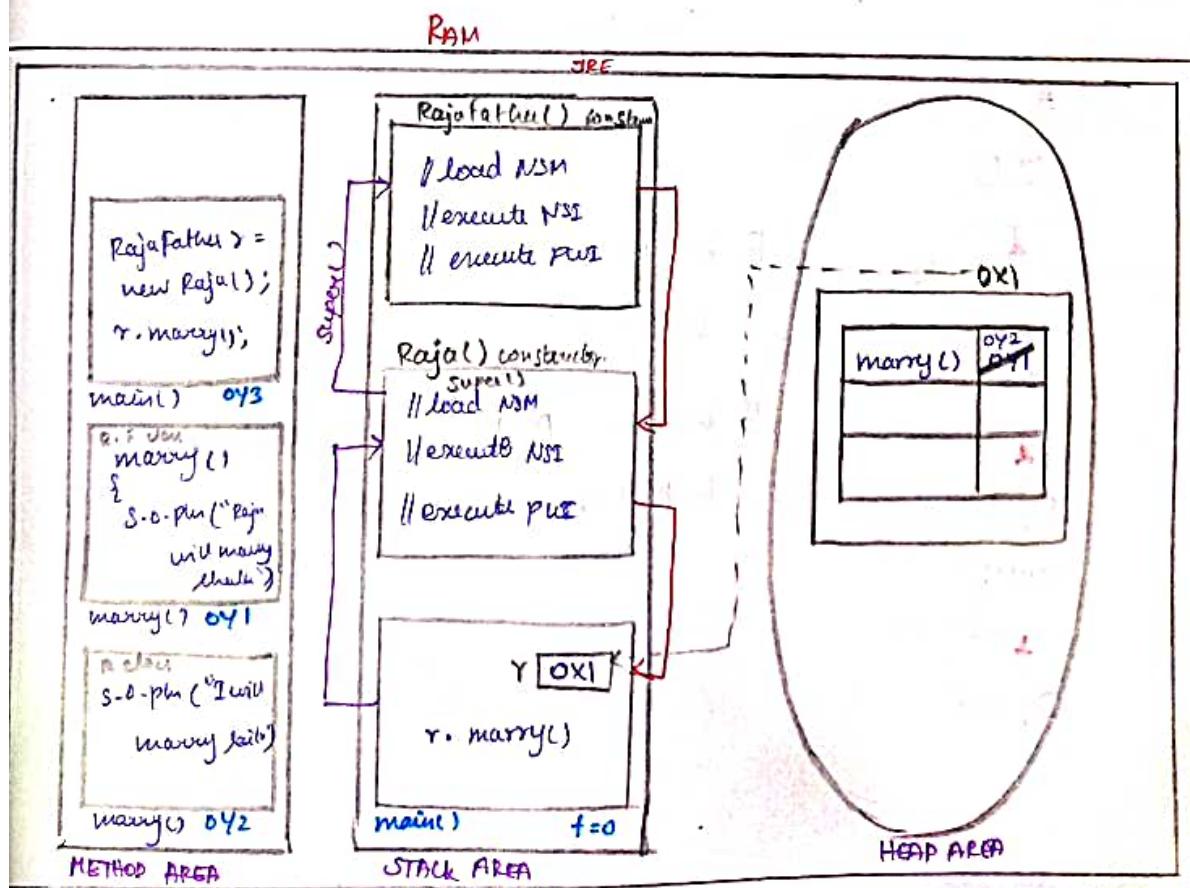
```
{   public void marry() //overriding  
{     System.out.println ("I will marry Laila");  
}  
}
```

class RajaDriver

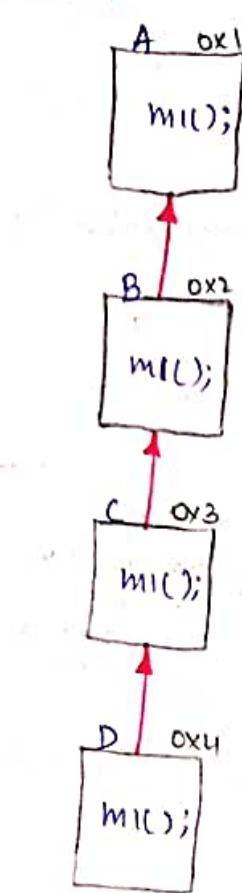
```
{   public static void main (String [] args)  
{     RajaFather r = new Raja(); //upcasting  
     r.marry(); //child class method will be called  
}  
}
```

#### OUTPUT:

I will marry Laila.



# Run Time Polymorphism



A a = new C();      a type A  
0x3  
a.m1(); // C class is called

A a = new D();      a type A  
0x4  
a.m1(); // D class is called

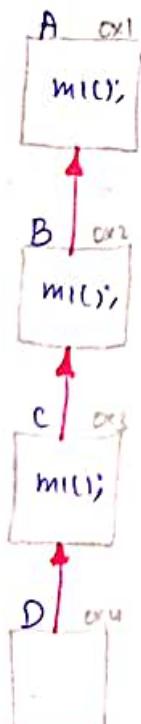
C c = new B(); // CTE  
c.m1();

B b = new D();      b type B  
0x4  
b.m1(); // D class is called

A a = new C();      a type A  
0x3  
a.m1(); // C class is called

B b = (B)a // down casting      b type B  
0x3  
b.m1(); // C class is called

C c = (B)a; // CTE  
c.m1();

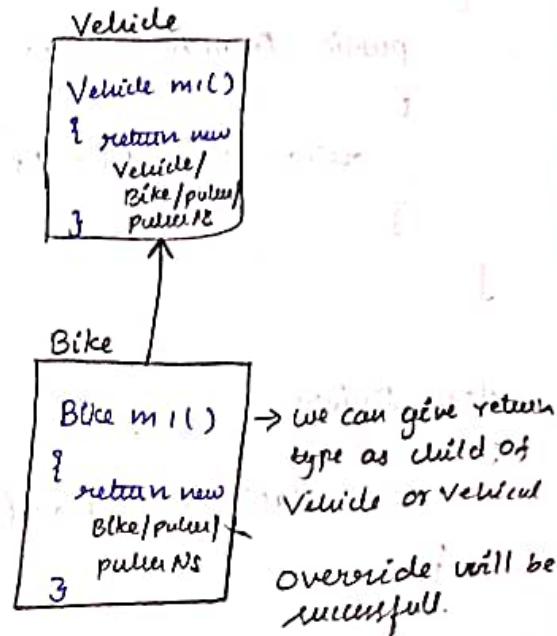
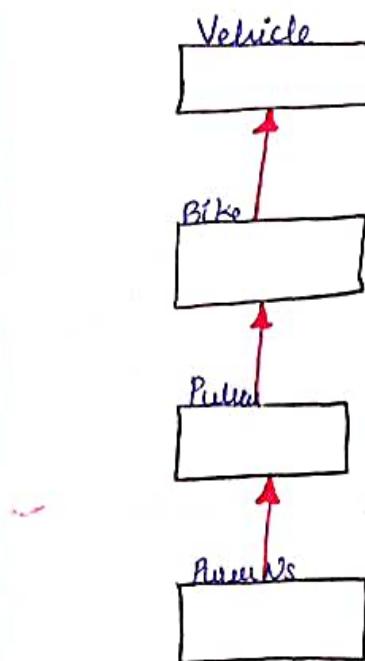


B b = new C();      b type B  
0x3  
b.m1(); // C class is called

B b = new D();      b type B  
0x4  
b.m1(); // C class is called

↓  
because 'D' class doesn't  
over ride.

## co-varient return type (only for Non-Primitive datatype)



Example :

```
class Vehicle
{
    public Vehicle m()
    {
        return new Bike; //CTS (Because Bike is child for Vehicle)
    }
}

class Bike extends Vehicle
{
    public Pulver m()
    {
        return new PulverNs;
    }
}

class Pulver extends Bike
{
    public Vehicle m()
    {
        return new Pulver;
    }
}
```

The code example demonstrates co-variant return types. In the first class, "Vehicle", the method "m()" returns a new object of type "Bike", which is a child class of "Vehicle". In the second class, "Bike", the method "m()" also returns a new object, but this time it's of type "Pulver", which is a child class of both "Vehicle" and "Bike". In the third class, "Pulver", the method "m()" returns a new object of type "Pulver", which is a child class of "Bike". The annotations "CTS" and "private" are handwritten near the code, likely referring to Compiler Type Safety and access modifiers respectively.

```
class Pattern extends Painter  
{  
    public Painter make()  
    {  
        return new Pattern();  
    }  
}
```

```
class Driver
```

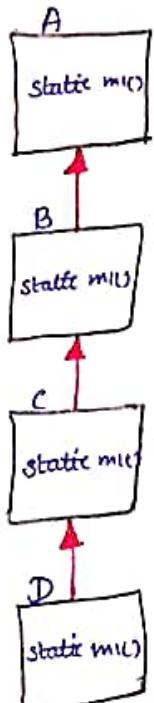
```
{  
    public static void main(String[] args)  
    {  
    }
```

## Method Shadowing:

If a parent and child class have a same static method (same signature) is known as method shadowing

In method shadowing the members always be choosed based on the type of variable

By using method shadowing we can achieve compile time Polymorphism



A `a = new C();`  
`a.m1();`

B `b = new D();`  
`b.m1();`

C `c = (C) new B();`  
`c.m1();`

B `b = new C();`  
A `a = b;`  
`b.m1();`  
`a.m1();`

C `c = new D();`  
`c.m1();`

A `a = c;`  
`a.m1();`

B `b = (B)a;`  
`b.m1();`

### Variable Shadowing :

If the superclass and subclass have variable with same name then it is known as Variable shadowing.

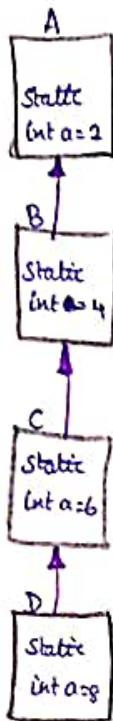
Which variable is used, depend on what?

In variable shadowing binding is done at compile time, hence it is a compile time polymorphism.

Variable used depends on the reference type and does not depend on the type of object created.

### NOTE:

- \* It is applicable for both static and non static variable
- \* It is a compile time polymorphism
- \* Variable usage depends on type of reference and does not depends on type of object created.



A a = new A();  
 a.a;  
  
 B b1 = new B();  
 b1.a;  
  
 C c1 = new C();  
 c1.a;  
  
 B b2 = new B();  
 b2.a;  
 a2.a;  
  
 C c2 = new C();  
 c2.a;  
  
 A a2 = c2;  
 a2.a;  
  
 B b = (B)a;  
 b.a;

What is object class?

- \* Object class is defined in java.lang package.
- \* Object class is a supermost parent class for all the classes in java.
- \* In object class there are 11 non-static methods.

### METHODS IN OBJECT CLASS:

1. public String toString()
  2. public boolean equals(Object o)
  3. public int hashCode()
  4. protected Object clone() throws CloneNotSupportedException  
(give the copy of the object)
  5. protected void finalize()
  6. final public void wait() throws InterruptedException
  7. final public void wait(long l) throws InterruptedException
  8. final public void wait(long l, int i) throws InterruptedException
  9. final public void notify() throws InterruptedException
  10. final public void notifyAll() throws InterruptedException
  11. final public void getClass() (used to check which class is present)
- used in  
multi-threading  
+ we can't  
override  
these methods  
bcz of final  
D.M  
can't override*

### toString() → METHOD :

- \* toString() method returns String
- \* toString() implementation of Object class returns the reference of an object in the String format.

Return format: className @ HexaDecimal

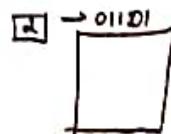
Example: (before overridden)

class Demo

1 public static void main(String[] args)

2 Demo d = new Demo();

3 System.out.println(d); //d.toString() Demo@2f29e04



### NOTE:

- \* Java doesn't provide the real address of an object
- \* Whenever programmer tries to print the reference variable `toString()` is implicitly called.

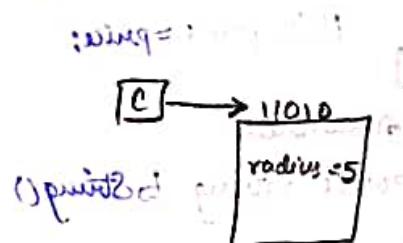
### Example : (overriding) `toString`

#### purpose:

We override `toString()` method to return state of an object instead of returning reference of an object.

eg :

```
class Circle
{
    int radius;
    Circle(int radius)
    {
        this.radius = radius;
    }
    @Override
    public String toString()
    {
        return "radius:" + radius;
    }
    public static void main(String[] args)
    {
        Circle c = new Circle(5);
        System.out.println(c); // toString() --> radius: 5
    }
}
```



## Example 2: (toString() overridden)

```
class Book extends Object  
{  
    String bname;  
    String author;  
    double price;  
    Book(String bname, String author, double price)  
    {  
        this.bname = bname;  
        this.author = author;  
        this.price = price;  
    }  
    @Override  
    public String toString()  
    {  
        return bname;  
    }  
}
```

```
class BookDriver  
{  
    public static void main(String[] args)  
    {  
        Book b1 = new Book("Java", "James Gosling", 3000);  
        Book b2 = new Book("Python", "Tabrez", 6000);  
        System.out.println(b1);  
        System.out.println(b2);  
    }  
}
```

### OUTPUT:

Java  
Python

**Example 3:** (`toString()` overridden)

```

class Student {  
    String sname;  
    int sid;  
}  
  
Student(String sname, int sid)  
{  
    this.sname = sname;  
    this.sid = sid;  
}  
  
@override  
public String toString()  
{  
    return "Student Name: " + sname + " Student id: " + sid;  
}

```

\* always be happy  
 \* always smile  
 \* keep on your face  
 \* with love  
 \* ~~with best regards~~

class StudentDriver

```

public static void main (String [] args)
{
    Student s1 = new Student ("Lavanya", 5037);
    Student s2 = new Student ("Manju", 5035);
    System.out.println(s1); // s1.toString() → internally
    System.out.println(s2); // s2.toString() → internally.
}

```

### OUTPUT:

Student Name: Lavanya Student id: 5037

Student Name: Manju Student id: 5035

## equals (Object) → METHOD

The return type of equals (Object) method is boolean

To equals (Object) method we can pass reference of any Object

The java.lang.Object class implementation of equals (Object) method is used to compare the reference of two objects

Example:

```
class Book
{
    String bname;
    Book (String bname)
    {
        this.bname = bname;
    }
}
```

case 1:

```
Book b1 = new Book ("java");
```

```
Book b2 = b1;
```

```
System.out.println (b1.bname);
```

```
System.out.println (b2.bname);
```

```
System.out.println (b1 == b2); // check address of an object
```

```
System.out.println (b1.equals (b2)); // check the address of two
                                     // objects because equals (Object)
                                     // method is not overridden
```

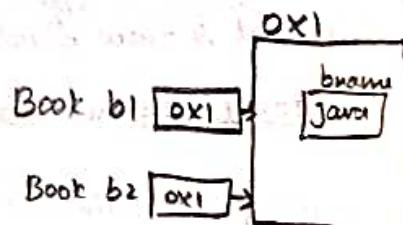
OUTPUT:

java

java

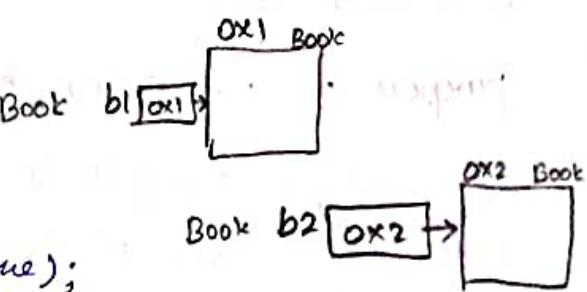
true

true



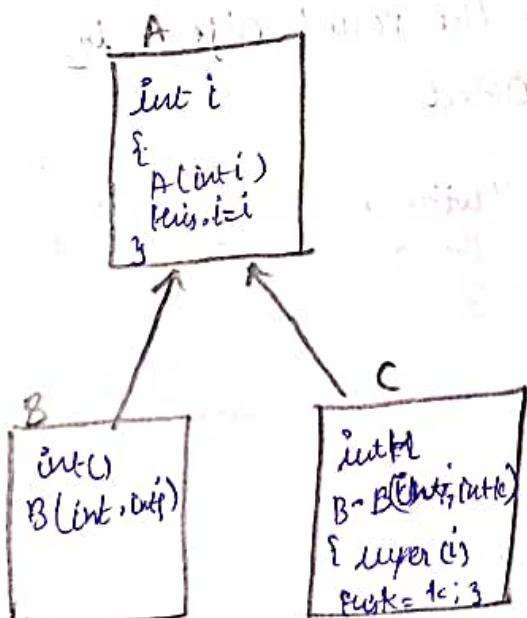
Ques 2:

```
Book b1 = new Book ("Java");
Book b2 = new Book ("Java");
System.out.println(b1.bname);
System.out.println(b1.bname);
System.out.println(b1 == b2);
System.out.println(b1.equals(b2)); // equals(object)
Book b1 = new Book ("Java");
System.out.println(b2) == true;
```



**NOTE:**

- \* If the reference is same == operator will get return true else false
- \* The equals (Object) method is same == operator



**Driver 1:**

```
Object o1 = new B(10, 20);
Object o2 = new B(10, 20);
System.out.println(o1 == o2);
System.out.println(o1.equals(o2));
```

**Driver 2:**

```
Object o1 = new C(7, 2);
Object o2 = new C(7, 2);
System.out.println(o1 == o2);
System.out.println(o1.equals(o2));
```

## ' equals (Object) → METHOD:

Purpose of overriding equals (Object):

We can override to equals (Object) method to compare the state of two objects instead of comparing reference of two objects.

NOTE:

\* If equals (Object) method is not overridden it compares the reference of two objects similar to == operator.

\* If equals (Object) method is overridden it compares the state of two objects. In such case comparing the reference of two objects is possible only by == operator.

DESIGN TIP:

In equals method compare the state of current (this) object with the passed objects by downcasting the parent object.

equals (Object o) Example: (without overriding)

class Student

{

String name;  
int sid;

Student (String name, int sid)

{

this.name = name;  
this.sid = sid;

}

}

gu

3

```

class StudentDriver {
    public static void main(String[] args) {
        Student s1 = new Student("Abhi", 5036);
        Student s2 = new Student("Chethana", 5038);
        System.out.println(s1 == s2); // false (compares address of the
                                     // object)
        System.out.println(s1.equals(s2)); // true (compares content)
    }
}

```

**Example: (equals() method overridden)**

```

class Student {
    String sname;
    int sid;
    Student(String sname, int sid) {
        this.sname = sname;
        this.sid = sid;
    }
    @Override
    public boolean equals(Object o) {
        Student s = (Student) o; // downcasting
        if (this.sname == s.sname && this.sid == s.sid)
            return true;
        else
            return false;
    }
}

```

```

class StudentDriver {
    public static void main(String[] args) {
        Student s1 = new Student("Abhi", 5036);
        Student s2 = new Student("Chethana", 5038);
    }
}

```

```
System.out.println(s1==s2); // false (compares address of the objects)  
System.out.println(s1.equals(s2)); // true (compares states of the objects)  
Student s3 = new Student("Abhi", 5036);  
System.out.println(s1==s3); // false (compare address of the objects)  
System.out.println(s1.equals(s3)); // true (compare states of the objects)
```

3

21-APR-22

### hashCode() METHOD:

The return type of hashCode() method is int.

The java.lang.Object implementation of hashCode()  
method is used to give the unique integer number of every  
object created.

The unique number generated based on the reference  
of an object.

### Purpose Of Overriding hashCode():

If the equals(Object) method is overridden, then it  
is necessary to override the hashCode() method.

### DESIGN TIP:

Call the hashCode() method of java.util.Objects class by  
passing all the attributes of an class as the actual  
arguments.

### Example 1:

```
class Book  
{  
    String bname;  
    Book(String bname)  
    {  
        this.bname = bname;  
    }  
}  
class Main  
{  
    public static void main(String args[]){  
        Book b1 = new Book("Java");  
        Book b2 = b1;  
        System.out.println(b1.bname);  
        System.out.println(b2.bname);  
        System.out.println(b1 == b2);  
        System.out.println(b1.equals(b2));  
        int h1 = b1.hashCode();  
        int h2 = b2.hashCode();  
        System.out.println(h1 == h2);  
    }  
}
```

### case 1:

```
Book b1 = new Book("Java");  
Book b2 = b1;  
System.out.println(b1.bname);  
System.out.println(b2.bname);  
System.out.println(b1 == b2);  
System.out.println(b1.equals(b2));  
int h1 = b1.hashCode();  
int h2 = b2.hashCode();  
System.out.println(h1 == h2);
```

### OUTPUT:

```
Java  
Java  
true  
true  
true
```

## Case 2:

```
Book b1 = new Book("Java");
Book b2 = new Book("Java");
System.out.println(b1.bname);
System.out.println(b2.bname);
System.out.println(b1 == b2);
System.out.println(b1.equals(b2));
int h1 = b1.hashCode();
int h2 = b2.hashCode();
System.out.println(h1 == h2);
```

### OUTPUT:

```
java
Java
false
false
```

### OBSERVATION:

In the above cases it is clear that

\* If the hashCode for two object is same, equals(Object) method should return true

\* If the hashCode for two object is different, equals(Object) method should return false.

**Example : ( hashCode Overridden) without**

```
class Student
{
    String sname;
    int sid;

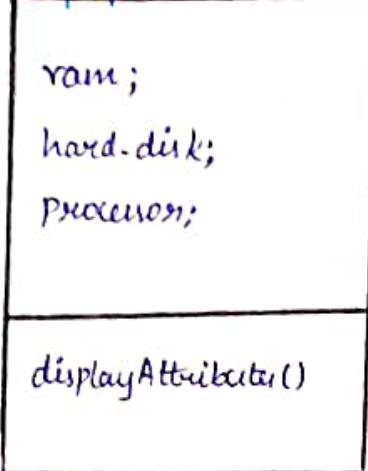
    Student (String sname, int sid)
    {
        this.sname = sname;
        this.sid = sid;
    }

    class StudentDriver
    {
        public static void main (String [] args)
        {
            Student s1 = new Student ("V.P", 5021);
            Student s2 = new Student ("Karthi", 5022);
            int h1 = s1.hashCode ();
            int h2 = s2.hashCode ();
            System.out.println (h1 == h2); // false ( code is generated based on
   // object address)

            Student s3 = new Student ("Karthi", 5022);
            int h3 = s3.hashCode ();
            System.out.println (h2 == h3); // false ( code is generated based on
   // object address)
        }
    }
}
```

Example: (toString(), equals(Object o), hashCode() are overridden)

### Laptop



Create a class Laptop, declare the attributes as private

Create a getter and setter methods for attributes

Create a necessary constructor for the Laptop class

Override the toString(), equals() and hashCode()

```
import java.util.Objects;  
class Laptop
```

```
{  
    int ram;  
    int hard-disk;  
    double processor;
```

//constructor

```
Laptop() {}
```

```
Laptop(int ram, int hard-disk, double processor)
```

```
{  
    this.ram = ram;
```

```
    this.hard-disk = hard-disk;
```

```
    this.processor = processor;
```

```
}
```

@Override

```
public boolean equals(Object o) //equals(Object) overridden
```

```
{
```

```
    Laptop l = (Laptop)o;
```

```
    if (this.ram == l.ram && this.hard-disk == l.hard-disk  
        && this.processor == l.processor)
```

```
{
```

```
        return true;
```

```
}
```

```
else
```

```
    return false;
```

@Override

```
public String toString() // toString() overridden  
{  
    return ram + "," + hard-disk + "," + processor;  
}
```

@Override

```
public int hashCode() // hashCode() overridden  
{  
    return Objects.hash(ram, hard-disk, processor);  
}
```

// behaviour

```
public void displayAttributes()
```

{

```
    System.out.println("ram :" + ram);  
    System.out.println("hard-disk :" + hard-disk);  
    System.out.println("processor :" + processor);
```

}

: Overrided prints

}

class LaptopDriver

{

```
    public static void main(String[] args)
```

{

```
    Laptop l1 = new Laptop("8", 123, 755);
```

```
    Laptop l2 = new Laptop("4", 245, 860);
```

```
    System.out.println(l1);
```

```
    System.out.println(l2);
```

```
    System.out.println(l2 == l1);
```

```
    System.out.println(l1.equals(l2));
```

```
    int h1 = l1.hashCode();
```

```
    int h2 = l2.hashCode();
```

```
    System.out.println(h1 == h2);
```

3

3

10/10

## OUTPUT :

8,123,755

4,245,860

false

false

false



## STRING CLASS

String a class , it is a non primitive data type.

what is not primitive data?

The address / reference of an object/class is called non-primitive data.

## String Literals:

Anything enclosed within the double quote "" in java is considered as String literal

characteristics of String literals:

\* When a String literals is used in java program, an instance of java.lang.String class is created inside a String pool.

\* For the given String literal's if the instance of a String is already present. Then, new instance is not created instead of reference of a existing instance is given.

## String:

\* String is a literal (data). it is a group of character that is enclosed within the double quotes "".

\* It is a Non primitive data.

\* In java we can store a string by creating instance of the following classes.

java.lang.String

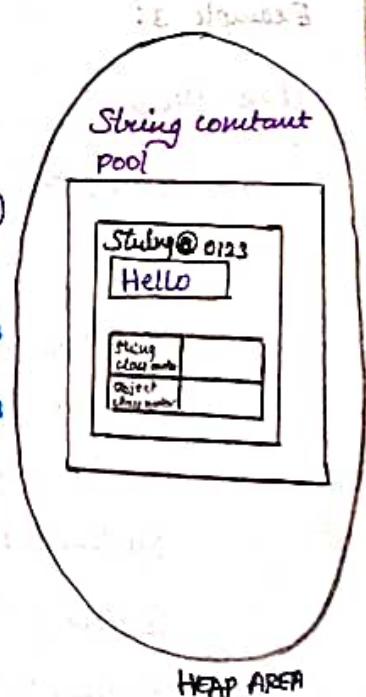
java.lang.StringBuffer

java.lang.StringBuilder

\* In java, whenever we create a string compiler implicitly create an instance for java.lang.String in String pool area / String constant pool (SCP)

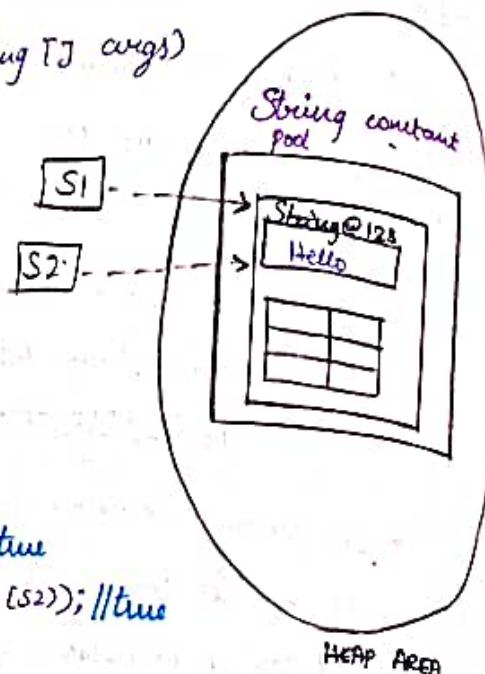
### Example : 1

```
class Demo
{
    public static void main (String [ ] args)
    {
        System.out.println ("Hello"); //String@0123
        System.out.println ("Hello"); //String@0123
    }
}
```



### Example 2:

```
class Demo
{
    public static void main (String [] args)
    {
        String s1, s2;
        s1 = "Hello";
        s2 = "Hello";
        System.out.println(s1);
        System.out.println(s2);
        System.out.println (s1==s2); //true
        System.out.println (s1.equals(s2)); //true
    }
}
```



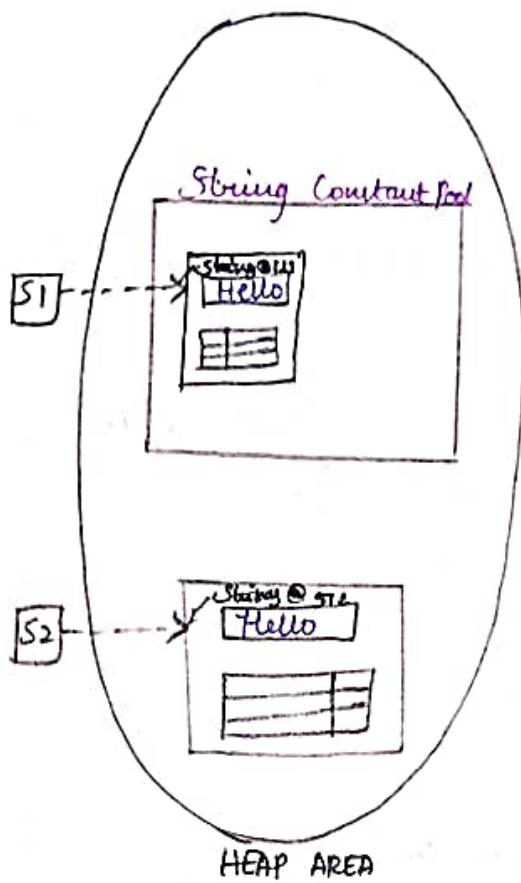
*Example 3:*

```

class Demo {
    public static void main(String[] args) {
        String s1, s2;
        s1 = "Hello";
        s2 = new String("Hello");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s1 == s2); // false
        System.out.println(s1.equals(s2)); // false true (compares state because equals() is overridden)
        System.out.println(s1.hashCode() == s2.hashCode()); // false
    }
}

```

(Compare state and generate int value because hashCode() is overridden)



## Constructor in `String` class:

| CONSTRUCTOR                           | DESCRIPTION                                                                                 |
|---------------------------------------|---------------------------------------------------------------------------------------------|
| <code>String()</code>                 | create an empty <code>String</code> object                                                  |
| <code>String(String literals)</code>  | creates <code>String</code> object by initializing with <code>String</code> literals        |
| <code>String(char[] ch)</code>        | creates <code>String</code> by converting character array into <code>String</code>          |
| <code>String(byte[] b)</code>         | creates <code>String</code> by converting byte array into <code>String</code>               |
| <code>String(StringBuffer sb)</code>  | creates <code>String</code> by converting <code>StringBuffer</code> to <code>String</code>  |
| <code>String(StringBuilder sb)</code> | creates <code>String</code> by converting <code>StringBuilder</code> to <code>String</code> |

## Example :

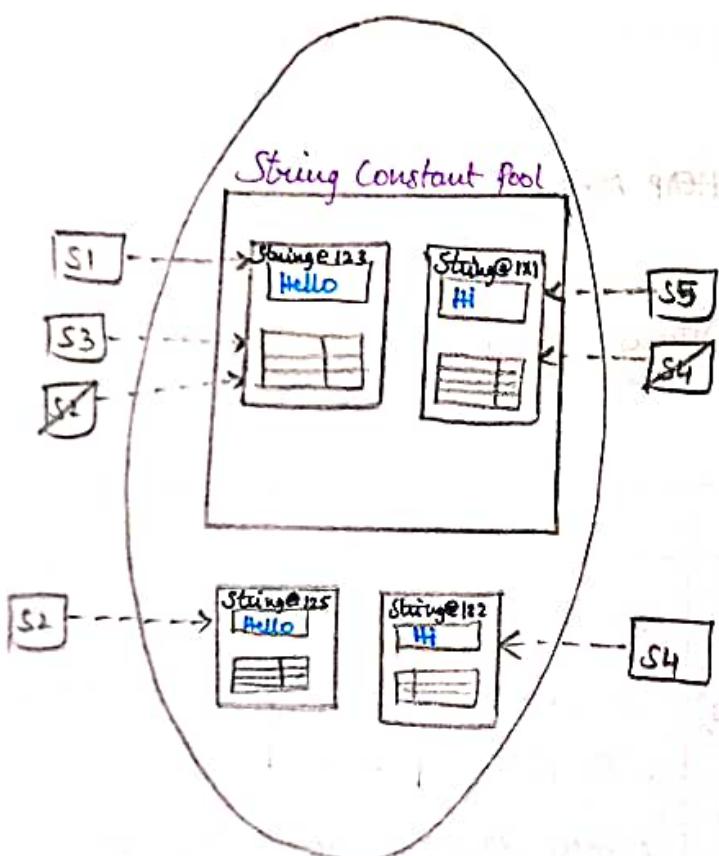
String s1 = "Hello";

String s2 = new String("Hello");

String s3 = "Hello";

String s4 = new String("Hi");

String s5 = "Hi";



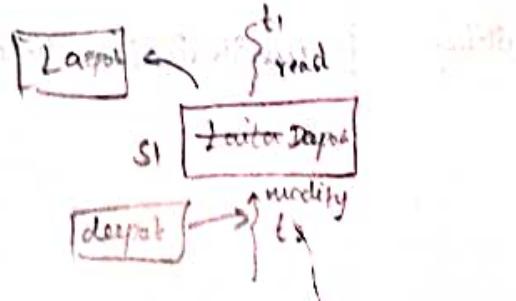
## METHODS OF STRING CLASS:

| RETURN TYPE | METHOD NAME                 | DESCRIPTION                                                                                          |
|-------------|-----------------------------|------------------------------------------------------------------------------------------------------|
| boolean     | equals (Object o)           | Compares state of two strings.                                                                       |
| boolean     | equalsIgnoreCase (String s) | Compares two strings by ignoring its case                                                            |
| boolean     | contains (String str)       | Returns true if specified String is present else it returns false                                    |
| boolean     | isEmpty ()                  | Returns true if String is empty else return false                                                    |
| char []     | toCharArray()               | Converts the specified String into character array                                                   |
| String []   | split (String str)          | Break the specified string into multiple string and returns String array                             |
| byte []     | getByte ()                  | Converts the specified string to byte value and returns byte value and returns byte array            |
| String      | toUpperCase ()              | Converts the specified string to upper case                                                          |
| String      | toLowerCase ()              | Converts the specified String to lower case : gothwakA                                               |
| String      | concat (String s)           | Joins the specified Strings.                                                                         |
| String      | trim ()                     | Remove the space present before and after the string                                                 |
| String      | substring (int index)       | Extract a characters from a String object starts from specified index and ends at the end of strings |

| RETURN TYPE | METHOD NAME                                  | DESCRIPTION                                                                                   |
|-------------|----------------------------------------------|-----------------------------------------------------------------------------------------------|
| String      | substring (int start, int end)               | Extract a character from a string starts from specified index and ends at end-1 index.        |
| char        | charAt (int index)                           | Returns character of the specified index from the string                                      |
| int         | indexOf (char sequence str)                  | Return the index of the specified string by searching from specified index if not return -1   |
| int         | indexOf (char sequence str, int start-index) | Return the index of the specified string by searching from specified index if not returns -1. |
| int         | lastIndexOf (char ch)                        | Returns the index of the character which is occurred at last in the original string.          |
| int         | length()                                     | Returns the specified string its byte value and returns byte array.                           |

### Advantage:

- Whenever we try to modify the String the new object is created so, that we can achieve thread safety in order to avoid data inconsistency.



## Disadvantages:

### Performance:

whenever we try to do the modifiable the string / change the string for a number of time. for each iteration new object is created so, the performance is less.

12-4-22

## STRING BUILDER

String Builder is a build in class which is used to store String data in Java.

**NOTE:** why we go for String Builder?

String Builder helps the programmer to store string data/literal in mutable object to overcome the performance issue of String class.

Is `toString()` is overridden in `StringBuilder`?

```
class Demo{
    public static void main(String[] args)
    {
        StringBuilder city = new StringBuilder("Bangalore");
        System.out.println(city);
    }
}
```

**OUTPUT:**

Bangalore

Conclusion.

`toString()` is overridden in `StringBuilder` class.

Can we store String literal directly into the  
StringBuilder type variable?

No, we can't store string literal directly into  
the StringBuilder type variable.

eg:

```
class Demo
{
    public static void main(String [] args)
    {
        StringBuilder city = "Bangalore"; // CTE
    }
}
```

reason:

The String literal was an object in String type  
so, we can't store String type data into StringBuilder type.  
Therefore here is an incompatible type error will occur.

How to store String in a StringBuilder type variable?

We can a constructor in StringBuilder class which can  
accept String literal. so, by using that constructor we  
can store the String data.

eg:

```
class Demo
{
    public static void main(String [] args)
    {
        StringBuilder city = new StringBuilder("Bangalore");
        System.out.println(city); // toString() is
    }
}
```

Output:

Bangalore.

Can we concat two `StringBuilder` data by using '+' operator?

No, we can not concat.

eg:

```
class Demo
{
    public static void main (String [] args)
    {
        StringBuilder s1 = new StringBuilder ("Manju");
        StringBuilder s2 = new StringBuilder ("Laranya");
        System.out.println (s1+s2); // CTE - bad operand type
    }
}
```

Then, How to concat two `StringBuilder` type data?

By using `append (StringBuilder sb)` we can add (concat) the `StringBuilder` data. Not only `StringBuilder` data we can pass any type of data to append with `StringBuilder` data. Because `append` method is overloaded in order to accept all the data.

eg:

```
class Demo
{
    public static void main (String [] args)
    {
        StringBuilder s1 = new StringBuilder ("Manju");
        StringBuilder s2 = new StringBuilder ("Laranya");
        s1.append (s2); // Manju CTS → append () accepting StringBuild
                        // type data
        System.out.println (s1); // Manju Laranya
        s1.append ("chethana"); // CTS → append () accepting String data
        System.out.println (s1); // Manju Laranya chethana
    }
}
```

## STRING BUFFER

### java.lang.StringBuffer:

- \* It is a inbuilt class defined in java.lang Package.
- \* It is a final class
- \* It helps to create mutable instance of String
- \* StringBuffer does not have String Constant Pool.
- \* It inherits java.util.Object class.
- \* In StringBuffer equals(), hashCode() method of java.lang.Object class are not overridden.

It implements:

- ⇒ Serializable
- ⇒ CharSequence

### constructors:

| CONSTRUCTORS             | DESCRIPTION                                    |
|--------------------------|------------------------------------------------|
| StringBuffer()           | creates empty String with initial capacity 16  |
| StringBuffer(String str) | create String buffer with the specified String |

### Example 1:

```
class Demo
{
    public static void main (String [] args)
    {
        StringBuffer s1, s2;
        s1 = new StringBuffer ("Hello");
        s2 = new StringBuffer ("Hello");
        System.out.println (s1); // Hello
        System.out.println (s2); // Hello
    }
}
```

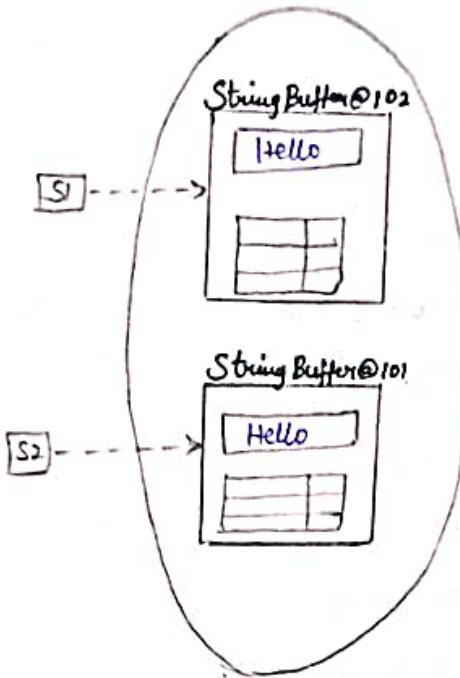
System.out.println(s1==s2); //compare address  
System.out.println(s1.equals(s2)); //compare address

}

}

OUTPUT:

false  
false



Example 2:

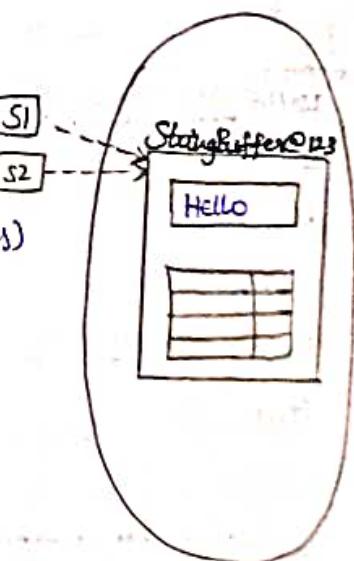
```
class Demo
{
    public static void main(String[] args)
    {
        StringBuffer s1, s2;
        s1 = new StringBuffer("Hello");
        s2 = s1;

        System.out.println(s1); //Hello
        System.out.println(s2); //Hello
        System.out.println(s1==s2); //compare address
        System.out.println(s1.equals(s2)); //compare address
    }
}
```

}

Output:

true  
true



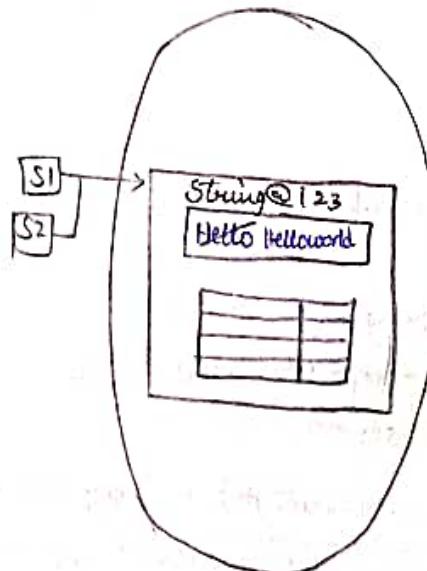
```

class Demo
{
    public static void main(String[] args)
    {
        StringBuffer s1, s2;
        s1 = new StringBuffer("Hello");
        s1 = s2;
        System.out.println(s1); //Hello
        System.out.println(s2); //Hello
        s1.append(" world");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}

```

OUTPUT:

Hello  
Hello  
Hello world  
Hello world  
true  
true



## IMPORTANT METHODS OF STRING BUFFER

| RETURN TYPE  | METHOD NAME                           | DESCRIPTION                                                                           |
|--------------|---------------------------------------|---------------------------------------------------------------------------------------|
| int          | capacity()                            | Returns current capacity                                                              |
| int          | length()                              | Returns length of the String                                                          |
| char         | charAt(int index)                     | Returns the character of the specified index                                          |
| StringBuffer | append(String s)                      | join anytype of data with StringBuffers . (overloaded method)                         |
| StringBuffer | insert(int index, String s)           | insert a specified String into original String of specified index (overloaded method) |
| StringBuffer | delete(int begin, int end)            | Delete String from specified beginning index to end-1 index                           |
| StringBuffer | deleteCharAt(int index)               | Delete the character present in the specified index                                   |
| StringBuffer | reverse()                             | Reverse the string literal                                                            |
| StringBuffer | setLength(int length)                 | only specified length in a string is exist remaining get removed                      |
| StringBuffer | subString(int begin)                  | Returns the subString from the specified beginning index                              |
| StringBuffer | subString(int begin, int end)         | Returns the subString from the specified beginning index to end-1 index               |
| StringBuffer | replace(int begin, int end, String s) | Replace a specified string from the beginning index to end-1 index                    |

| RETURN TYPE | METHOD NAME                        | DESCRIPTION                                                                |
|-------------|------------------------------------|----------------------------------------------------------------------------|
| Void        | trimToSize()                       | Removes the unused capacity or set the capacity till length of the string. |
| Void        | setCharAt(int index, char newchar) | Replace the new character in a string of specified index                   |
| Void        | ensureCapacity(int capacity)       | Sets capacity for storing a string                                         |

### DIFFERENCE BETWEEN STRINGBUFFER AND STRINGBUILDER

| STRING BUFFER                                                                                      | STRING BUILDER                                                                                             |
|----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| All the method present in StringBuffer is synchronized                                             | All the method present in String Builder is non synchronized                                               |
| At a time only one thread is allowed to access String Buffer Object. Hence it is Thread safe       | At a time multiple thread is allowed to access String Builder Object. Hence it is not Thread safe          |
| Threads are required to wait to operate a StringBuffer object. Hence Relatively performance is low | Threads are not required to wait to operate a String Builder object. Hence Relatively performance is high. |
| Less efficiency than StringBuilder                                                                 | Efficiency is high compared to StringBuffer                                                                |
| introduced in 1.0v                                                                                 | introduced in 1.5v                                                                                         |

## ABSTRACTION

abstraction:

It is a design process of hiding the implementation and showing only the functionality (**only declaration**) to the user is known as abstraction.

How to achieve abstraction in java?

\* In java we can achieve abstraction with the help of abstract classes and interfaces

\* we can provide implementation to the abstract component with the help of inheritance and method overriding

**abstract modifier:**

\* abstract is a modifier, it is a keyword

\* It is applicable for methods and classes.

**abstract method:**

\* A method that is prefixed with an abstract modifier is known as the abstract method

\* This is also said to be an incomplete method.

\* The abstract method doesn't have a body (**it has the only declaration**)

Syntax to create abstract method:

abstract [access modifier] returnType methodName(IFAA)

**NOTE:**

only child class of that class is responsible for giving implementation to the abstract method.

## Abstract class:

- \* If the class is prefixed with an abstract modifier then it is known as abstract class.
- \* We can't create the object (*instance*) for an abstract class.

### NOTE:

- \* We can't instantiate an abstract class
- \* We can have an abstract class without an abstract method.
- \* An abstract class can have both abstract and concrete method.
- \* If a class has atleast one abstract method either declared or inherited but not overridden it is mandatory to make that class as abstract class.

### Example:

```
abstract class Atm
{
    abstract public double withdrawal();
    abstract public void getBalance();
    abstract public void deposit();
}
```

3) // hiding implementation by providing only functionality

```
class AtmDriver
{
    public static void main (String[] args)
    {
        Atm a = new Atm(); // CTE
    }
}
```

**NOTE:**

only subclass of Atm is responsible for giving implementation to the method declared in a Atm class.

### Implementation of abstract method:

- \* If a class extend abstract class then it should give implementation to all the abstract method of the super class.
- \* If inheriting class doesn't like to give implementation to the abstract method of superclass then it is mandatory to make subclass as an abstract class.
- \* If a subclass is also becoming an abstract class then the next level child class is responsible to give implementation to the abstract method.

### STEPS TO IMPLEMENT ABSTRACT METHOD:

**STEP 1:**

Create a class.

**STEP 2:**

Inherit the abstract class / component

**STEP 3:**

Override the abstract method inherited (provide implementation to the inherited abstract method.)

### CONCRETE CLASS:

The class which is not prefixed with an abstract modifier and doesn't have any abstract method, either declared or inherited is known as concrete class

**NOTE:**

In java we can create object only for the concrete class

Ab.

### CONCRETE METHOD:

The method which give implementation to the abstract method is known as concrete method.

11

### Example :

```
o abstract class WhatsApp
{
    abstract public void send();
}

class Application extends WhatsApp // concrete class
{
    public void send() // concrete method
    {
        System.out.println("send() method is implemented");
    }
}

class WhatsAppDriver // concrete class.
{
    public static void main(String[] args)
    {
        WhatsApp w = new Application();
        w.send(); // send method is implemented (i.e) overridden
    }
}
```

E

send() method is implemented

w.send(); // send method is implemented (i.e) overridden

to send() of Application class will be executed

### Example:

```
abstract class Parent // we can't create object for Parent.
```

{

```
    abstract public void grandSon();
```

```
    abstract public void grandDaughter();
```

}

```
abstract class Raja extends Parent // we can't create object for Raja.
```

{

```
    public void grandSon() // concrete method
```

{

```
        System.out.println (" your dreams fulfilled");
```

}

}

```
class Rajason extends Raja // concrete class
```

{

```
    public void grandDaughter() // concrete method
```

{

```
        System.out.println (" I am better than my Father Raja");
```

}

}

```
class Driver
```

{

```
    public static void main (String [] args)
```

{

```
        Rajason p = new Rajason();
```

```
        p.grandSon();
```

```
        p.grandDaughter();
```

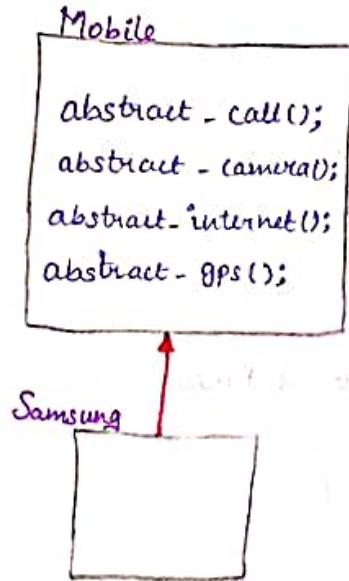
}

}

**OUTPUT :**

**Example:**

**Ab**



abstract class Mobile

{

    abstract public void call();

    abstract public void camera();

    abstract public void internet();

    abstract public void gps();

}

class Samsung extends Mobile

{

    public void call()

    {

        System.out.println ("calling option is available");

    }

    public void camera()

    {

        System.out.println ("camera is on");

    }

    public void internet()

    {

        System.out.println ("4G internet");

    }

```
public void gps()
{
    System.out.println("GPS is available");
}

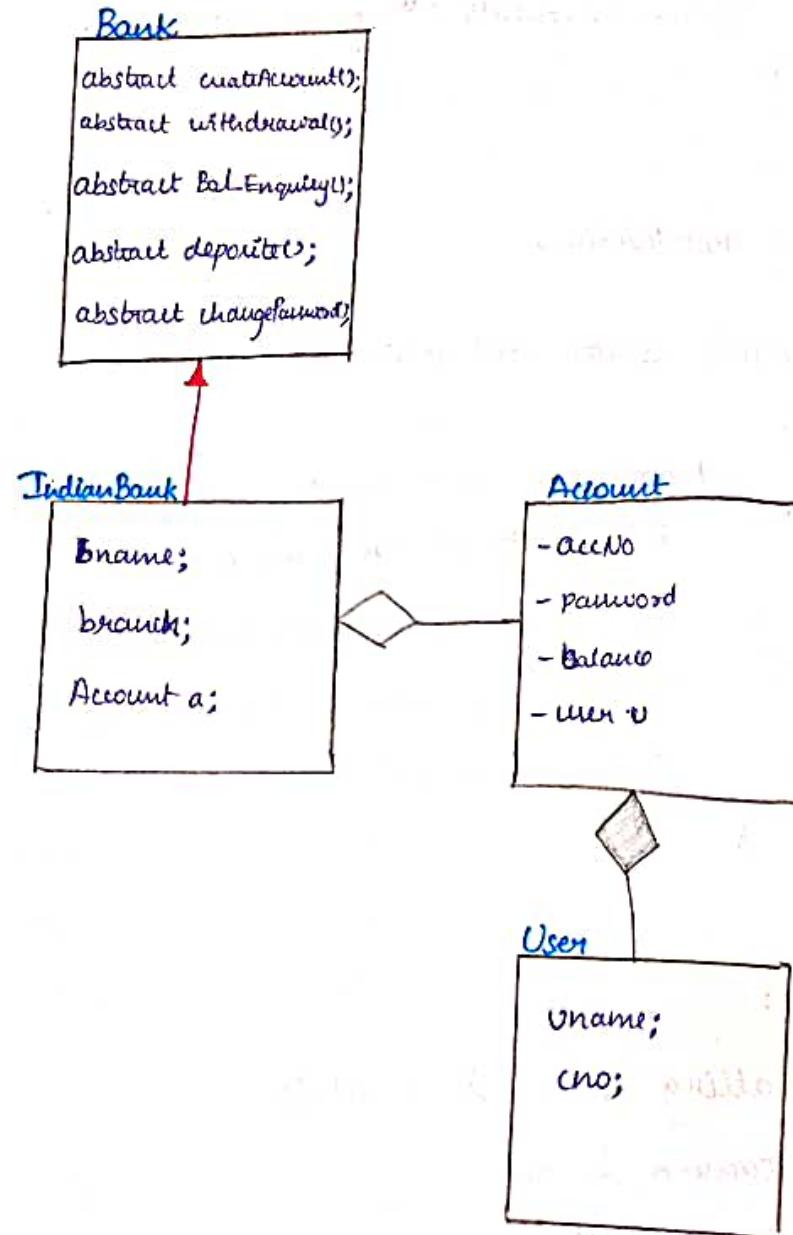
class MobileDriver
{
    public static void main(String [] args)
    {
        Mobile m = new Samsung();
        System.out.println(m.call());
        System.out.println(m.camera());
        System.out.println(m.internet());
        System.out.println(m.gps());
    }
}
```

#### OUTPUT:

Calling option is available  
camera is on  
4G internet  
GPS is available

### Example :

At



E

**Interface:** It is a blue print of class

It is a component in java which is used to achieve 100% abstraction with multiple implementation.

Syntax to create an interface

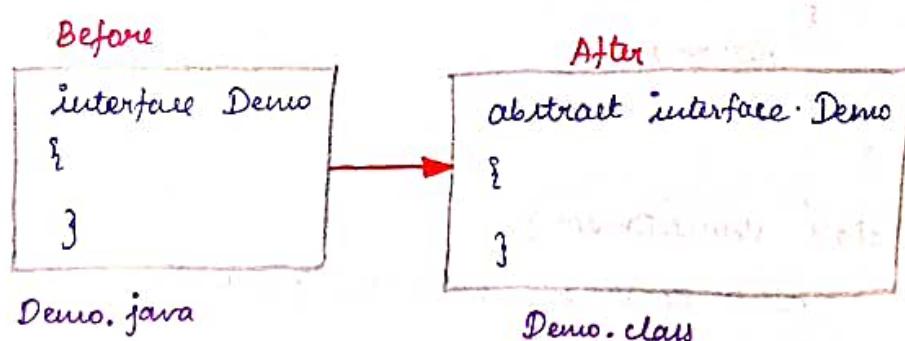
```
[Access Modifier] interface InterfaceName
{
    //declare members
}
```

when an interface is compiled we get a class file with extension .class only

**Example:**

```
interface Demo
{
}
```

Demo is an interface



Q what all are the members are not inherited from an interface?

A) only static methods of an interface is not inherited to both class and interface.

Q why do we need an interface?

To achieve 100% abstraction. concrete non static methods are not allowed (but after 1.8 version default non static methods are allowed and we can override that method by prefixing with public access modifier).

E Example:

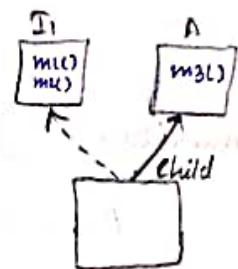
```
interface Vehicle
{
    int getNoOfWheels();
}

class Bike implements Vehicle
{
    public int getNoOfWheels()
    {
        return 2;
    }
}

class VehicleDriver
{
    public static void main(String[] args)
    {
        Bike b = new Bike();
        System.out.println(b.getNoOfWheels());
        Vehicle v = new Bike();
        System.out.println(v.getNoOfWheels());
    }
}
```

Interface I:

```
{  
    void m1();  
    void m2();  
}  
  
class A  
{  
    public void m3()  
    {  
        System.out.println("m3() from class A");  
    }  
}  
  
class Child extends A implements I  
{  
    public void m1()  
    {  
        System.out.println("m1() from child");  
    }  
  
    public void m2()  
    {  
        System.out.println("m2() from child");  
    }  
  
    public void m3()  
    {  
        System.out.println("m3() from child");  
    }  
}  
  
class Driver  
{  
    public static void main(String[] args)  
    {  
        Child c = new Child();  
        c.m1();  
        c.m2();  
        c.m3();  
  
        I i = new Child();  
        i.m1();  
        i.m2();  
    }  
}
```



Ab

i.m3(); // CTE

A a = c;

tt

a.m1(); // CTE

a.m2(); // CTE

a.m3();

c

}

}

## INHERITANCE

Inheritance with respect to interface :

An interface can inherit any number of interfaces with the help of extend keyword.

Example :

E

interface I1

{

}

interface I2

{

}

interface I3 extends I1, I2

{

NOTE :

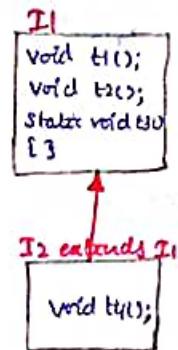
The interface which is inheriting an interface should not give implementation to the abstract methods.

### Example 1:

Interface I1 have 3 methods

2 - non static (t1(), t2())

1 - static (t3())



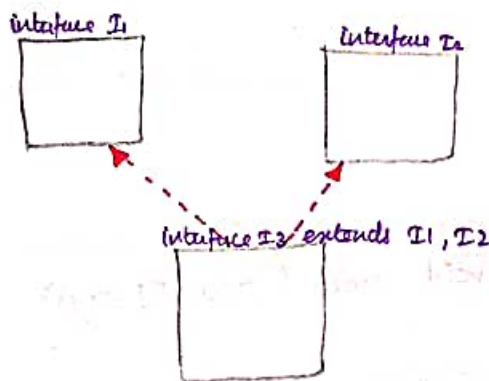
Interface I2 have 3 methods

2 - inherited non-static methods (t1(), t2())

1 - declared non-static method (t4())

### Example 2:

Interface can inherit multiple interfaces at a time



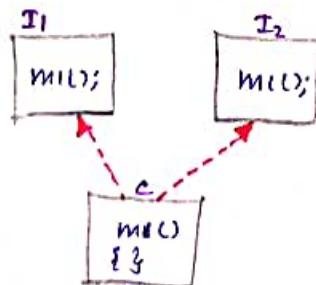
#### NOTE :

with respect to interface there is no diamond problem

#### Reason ,

- \* They don't have a constructor
- \* Non static methods are abstract (do not have implementation)
- \* Static methods are not inherited

### Example to overcome Diamond Problem



Ab

class Care(i)

interface I1

{

    void m();

}

interface I2

{

    void m();

}

class C implements I1, I2

{

    public void m()

{

        System.out.println ("m from child");

}

}

class Driver

{

    public static void main (String [ ] args)

{

    C c = new C();

    c.m(); // C class m() is called

    I1 i = new C();

    i.m(); // C class m() is called

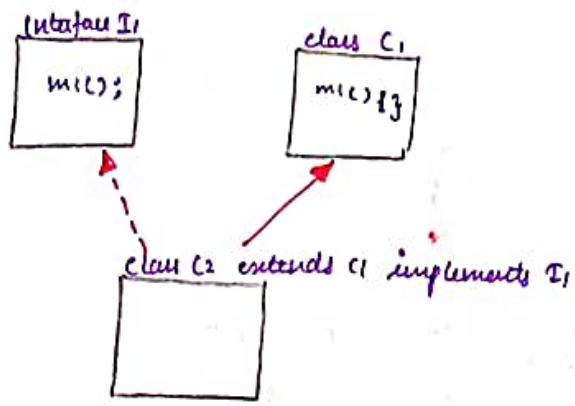
    I2 ii = c;

    ii.m(); // C class m() is called.

}

}

Case (ii)



Interface I

```
{ void m(); }
```

class C1

```
{ public void m()
    {
        System.out.println("m() from class C1");
    }
}
```

class C2 extends C1 implements I

```
{ }
```

class Driver

```
{ public static void main (String [] args)
    {
        C1 c = new C1();
        c.m();
    }
}
```

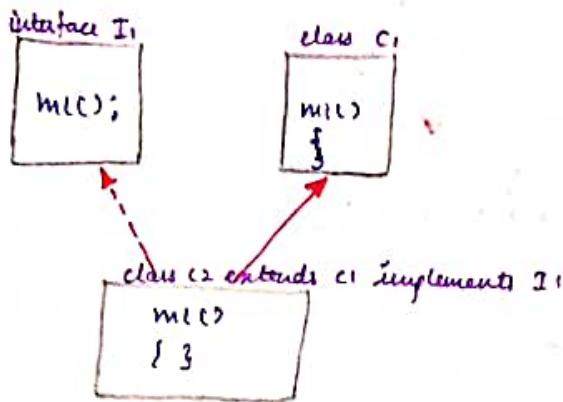
```
    C2 c2 = new C2();
    c2.m();
}
```

```
II i = new C2();
i.m();
}
```

}

A1

case (iii)



interface I1

{

    void m1();

}

class C1

{

    public void m1()

{

        System.out.println("m1 from class C1");

}

}

class C2 extends C1 implements I1

{

    public void m1()

{

        System.out.println("m1 from class C2");

}

}

class Driver

{

    public static void main(String[] args)

{

        C1 a = new C1();

        a.m1();

        C2 b = new C2();

        b.m1();

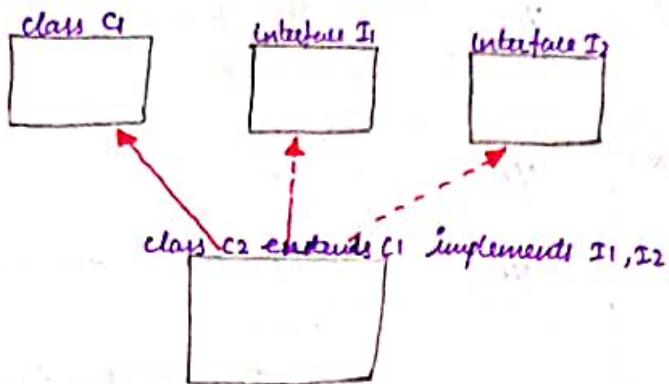
    I1 i = new C2();

    i.m1();

}

### Example 3:

class can inherit interface and class at a time



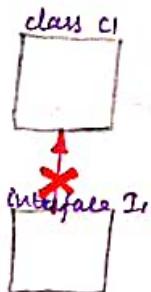
### Rule:

use the `extends` first and then `implements`

### NOTE:

\* class can inherit multiple interface but it can't inherit multiple class at a time.

\* Interface can't inherit a class



### Reason:

class has concrete non static methods , so  
class can't be a parent to the interface .

What all are the members that can be declared in an interface

A.

| MEMBERS                                     | CLASS | INTERFACE                                                                                                                  |
|---------------------------------------------|-------|----------------------------------------------------------------------------------------------------------------------------|
| static variables                            | yes   | Yes, but only final static variable                                                                                        |
| Non-static variables                        | yes   | No                                                                                                                         |
| Static methods                              | yes   | yes, from JDK 1.8v<br>NOTE:<br>They are by default public in nature                                                        |
| Non static methods                          | yes   | Yes, but we can have only abstract non static methods<br>NOTE: non static methods are by default<br>* public<br>* abstract |
| constructors                                | YES   | No                                                                                                                         |
| initializers<br>(static + non-static block) | YES   | No<br>NOTE: -                                                                                                              |

NOTE:

In Interface all members are by default have public access modifier.

## PACKAGES

### Packages :

A package in java is used to group a related classes, interfaces and subclasses. In a simple word it is a folder / directory which consists of several classes and interfaces.

#### NOTE:

Package contains only class file

### why Package?

- \* Package are used to avoid name conflict
- \* It increases maintainability
- \* It is used to categorize classes and interfaces
- \* It increases the access protection
- \* It is used to achieve code reusability.

### Types of packages:

we have two types of packages

- \* Built in packages
- \* User defined packages

### Built-in packages:

In java we have many built in packages like `java`, `lang`, `util`, `io`, `sql`, `swing`, `awt`, etc., which are included in java Development kit.

### Example:

`java.lang.Math`

`java` → package

`lang` → sub package

`Math` → class

A.

### Subpackage:

Package inside a package is called as sub package.

It should created to categorize a folder further.

### User Defined Package:

In java we can create our own package

### Syntax to create a package:

Package package-name;

### Subpackage:

We can create subpackage by using following syntax.

### Syntax to create package along with subpackage:

Package package-name.subpackage-name;

### RULE:

- \* Package should be the first statement in a java program.
- \* A java source file should contain only one package statement
- \* A package can contain multiple classes / Interfaces but atmost one class / interface should be public.
- \* If you want to add more than one public class inside the same package then, create a separate source file for each public class with same package name.
- \* If a package contains public class / interface then it is mandatory to use public classes / interfaces name as a source file name. Otherwise, we will get compile time error

**NOTE:**

If we want to use package and import statement in a same program then we should follow a sequence of programs.

package  
import  
class/interface/enum

---

24/4/22

## ARRAY

Array is a continuous block of memory which is used to store multiple values.

Characteristics of an array:

- \* The size of an array must be defined at the time of declaration
- \* Once declared, the size of an array can't modified
- \* Hence array is known as fixed size.
- \* In an array we can access the elements with the help of an index or subscript. It is an integer number that starts from 0 and ends at length of the array - 1.
- \* In an array we can store only homogeneous type value. It is also known as homogeneous collection of an object.

**NOTE:**

In java array is an object.

## A. Declaring An Array

Syntax to declare an array:

datatype[] variable; → convention

(or)

datatype variable[];

Example:

int a[] → single dimensional array reference variable of  
int type

float f[] → single dimensional array reference variable of  
float type

String s[] → single dimensional array reference variable  
of String type.

## INSTANTIATING AN ARRAY

Syntax to instantiate an array:

new datatype [size];

Example:

new int[5];

| ox1 |   |
|-----|---|
| 0   | 0 |
| 1   | 0 |
| 2   | 0 |
| 3   | 0 |
| 4   | 0 |

new String[5];

| ox1 |      |
|-----|------|
| 0   | null |
| 1   | null |
| 2   | null |
| 3   | null |
| 4   | null |

new boolean[4];

| ox1 |       |
|-----|-------|
| 0   | false |
| 1   | false |
| 2   | false |
| 3   | false |
| 4   | false |

NOTE:

Once the array is instantiated it is assigned with default value.

## INITIALIZING AN ARRAY

Adding elements:

we can add an element into an array with the help of array index.

Syntax to add an element into an array:

array-ref-variable [index] = value;

Example:

int [] arr = new int[5]; //declaration

arr[0] = 2;

arr[1] = 4;

arr[2] = 7;

arr[3] = 0;

arr[4] = 3;

} //initialization.

| arr---> 0x1 |
|-------------|
| 0 2         |
| 1 4         |
| 2 7         |
| 3 0         |
| 4 3         |

## ACCESSING ELEMENTS FROM AN ARRAY

Accessing elements:

we can access an element from an array with the help of array reference variable and index.

Syntax to access an element from an array:

array-ref-variable [index];

Example:

System.out.println(arr[0]); //2

System.out.println(arr[2]); //7

System.out.println(arr[4]); //3

System.out.println(arr[5]); //ArrayIndexOutOfBoundsException  
Exception

| arr---> 0x1 |
|-------------|
| 0 2         |
| 1 4         |
| 2 7         |
| 3 0         |
| 4 3         |

## A Declaring, Instantiating, Initializing An Array in a Single Line.

We can also declare, instantiate and initialize an array by using single line.

Syntax:

datatype[] arr-ref-var = { element1, element2, ... etc., };

Example:

int arr[] = { 1, 2, 3, 4, 5 };

| arr ---> 0x1 |
|--------------|
| 1            |
| 2            |
| 3            |
| 4            |
| 5            |

Characteristics of Array:

- If the size of the array is fixed we can't able to change the size

It is

YANNA OR MORE STURMELI NUNZESSA

elements present

number of bytes are

size of array is same as size of memory

Notable example

ref.

### Binary Search:

```

import java.util.Scanner;
class Search
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        int arr[] = {12, 54, 68, 75, 76, 98};
        int low = 0;
        int high = arr.length - 1;
        System.out.println("Enter the number to check present or not:");
        int key = s.nextInt();
        while (low <= high)
        {
            int mid = (low + high) / 2;
            if (key == arr[mid])
            {
                System.out.println("Present");
                return;
            }
            else if (key > arr[mid])
            {
                low = mid + 1;
            }
            else
            {
                high = mid - 1;
            }
        }
        System.out.println("Not present");
    }
}

```

### OUTPUT:

Enter the number to check present or not

12

Present

Enter the number to check present or not

28

Not Present

A

### Tracing:

key = 12  
low = 0

Case (i) : (key < mid)      high = 5

(i)      mid =  $\frac{0+5}{2} = 2$

else (key > mid)

12 > 68 (T)

else (high = mid - 1)

$$2 - 1 = 1$$

(ii)      mid =  $\frac{0+1}{2} = \frac{1}{2} = 0$

if (mid == key)

$$12 = 12$$

S.O.PLn ("Present");

Case (ii) (key == mid)

key = 68

low = 0

high = 5

(i)      low <= high

0 <= 5 (T)

$$\text{mid} = \frac{0+5}{2} = \frac{5}{2} = 2$$

If (key == arr[mid])

68 == 68 (T)

S.O.PLn ("Present");

Case (iii) (key > mid)

key = 98

low = 0, 5

high = 5

(i)      low <= high

0 <= 5 (T)

$$\text{mid} = \frac{0+5}{2} = \frac{5}{2} = 2$$

✓

case (iii) continue  
 if (key == arr[mid])  
 $98 == 68$  (F)  
 else if (key > mid)  
 $98 > 68$  (T)  
 $low = mid + 1;$

(iv) low <= high  
 $3 \leq 5$  (T)  
 $mid = \frac{3+5}{2} = 8/2 = 4$   
 if (key == arr[mid])  
 $98 == 76$  (F)  
 else if (key > arr[mid])  
 $98 > 76$  (T)  
 $low = mid + 1;$   
  
 (v) low <= high  
 $5 \leq 5$  (T)  
 $mid = \frac{5+5}{2} = 10/2 = 5$   
 if (key == arr[mid])  
 $98 == 98$  (T)  
 $s.o.\text{.put} ("Present");$

case (iv) continue..

(ii) low <= high  
 $3 \leq 5$  (T)  
 $mid = \frac{3+5}{2} = 8/2 = 4$

if (key == arr[mid])  
 $99 == 76$  (F)  
 else if (key > arr[mid])  
 $99 > 76$  (T)  
 $low = mid + 1;$

(iii) low <= high  
 $5 \leq 5$  (T)  
 $mid = \frac{5+5}{2} = 10/2 = 5$   
 if (key == arr[mid])  
 $99 == 98$  (F)  
 else if (key > arr[mid])  
 $99 > 98$   
 $low = mid + 1;$

(iv) low <= high  
 $6 \leq 5$  (F)

case (iv) key not in array

key = 99  
 $low = 0 \ 3 \ 5 \ 6$   
 $high = 5$

$s.o.\text{.put} ("Not present");$

(i) low <= high  
 $0 \leq 5$  (T)  
 $mid = 0 + 5/2 = 2$   
 if (key == arr[mid])  
 $99 == 68$  (F)  
 else if (key > mid)  
 $99 > 68$  (T)  
 $low = mid + 1;$

7/4/22

## ACCESS MODIFIER

A

### Access Modifier:

Access modifiers are used to change the accessibility of a member

- \* private
- \* default
- \* protected
- \* public

#### Private :

\* It is a class level modifier, it is applicable for variable, method and constructor.

\* If the member of a class is prefixed with private modifier then it is accessible only within the class, accessing outside the class is not possible.

#### Example :

```
class A
{
    private static int i;
}

class B
{
    public static void main (String [] args)
    {
        System.out.println (A.i); // CTE
    }
}
```

### default:

- \* The accessibility of default modifier is only within the package. It can't be accessed from outside the package.
- \* If you don't declare any access modifiers then it is considered as a default modifier.

### Example:

```
package myPack;  
public class Demo  
{  
    static int i;  
}  
import myPack.Demo  
class Driver  
{  
    public static void main (String[] args)  
    {  
        Demo.i = 5; // CTE  
    }  
}
```

### protected:

- \* The access level of protected modifier is within the package and outside the package through child class.
- \* If you do not make the child class, it cannot be accessed from outside the class.

### Example:

```
package com.ty.demo1;  
public class A  
{  
    protected void display()  
    {  
        System.out.println("good morning");  
    }  
}
```

A

```
Package com.ty.demo9
import com.ty.demo1.*;
class B extends A
{
    public static void main(String[] args)
    {
        B b = new B();
        b.display();
    }
}
```

**OUTPUT :**

good morning

**public:**

The access level of a public modifier is anywhere.  
It can be accessed from within the class, outside the class,  
as well as within the package as well as outside the  
package.

**Example :**

```
package Amazon1;
public class User
{
    public void message()
    {
        System.out.println("Hi");
    }
}
```

```
package Amazon2;
import Amazon1.*;
class Driver
{
    public static void main(String[] args)
    {
        User u = new User();
        u.message();
    }
}
```

## SCOPE OF AN ACCESS MODIFIER

| ACCESS MODIFIER | WITHIN THE CLASS | WITHIN THE PACKAGE | OUTSIDE THE PACKAGE | OUTSIDE THE PACKAGE BY CHILD CLASS |
|-----------------|------------------|--------------------|---------------------|------------------------------------|
| private         | Yes              | No                 | No                  | No                                 |
| default         | Yes              | Yes                | No                  | No                                 |
| protected       | Yes              | Yes                | No                  | Yes                                |
| public          | Yes              | Yes                | Yes                 | Yes                                |

## SORTING ALGORITHM.

sorting:

The arrangements of elements in an defined order is known as sorting.

eg: ascending order  
descending order

Some of the important sorting algorithms:

- \* Bubble sort
- \* Selection sort
- \* Insertion sort
- \* Quick sort
- \* Merge sort
- \* Heap sort etc.,

## A) Bubble Sort:

Design a method which can accept an integer array and sort it in ascending order using bubble sort algorithm.

```
Public static int int bubbleSort ( int arr )  
{  
    for( int i = 0 ; i < arr.length - 1 ; i++ )  
    {  
        for( int j = 0 ; j < (arr.length - i) - 1 ; j++ )  
        {  
            if ( arr [j] > arr [j+1] )  
            {  
                int temp = arr [j];  
arr [j] = arr [j+1];  
                arr [j+1] = temp;  
            }  
        }  
    }  
    return arr;  
}
```

## Assignment:

(-1) primitive methods overload  
Design a class Arrays . The class Arrays must have sort method to sort any type of primitive array in ascending order.

7/4/22

## Arrays. sort()

- \* It is a static method
- \* It sorts the elements of the array in ascending order.

### Sorting Primitive Array:

```
import java.util.Arrays;  
public class E6 {  
    public static void display (int [] a)  
    {  
        for (int e: a){  
            System.out.println (e + ", ");  
        }  
    }  
    public static void main (String [] args) {  
        int [] a = {10, 2, 3, 6, 20};  
        System.out.println ("before sorting");  
        display (a);  
        Arrays.sort (a);  
        System.out.println ("after sorting");  
        display (a);  
    }  
}
```

148

A

To sort non-primitive array:

Arrays.sort() can sort non primitive array in two ways

⇒ by using compareTo(Object o) method of Comparable interface.

⇒ by using compare(object, object) method of Comparator interface.

By Using compareTo(Object o) of Comparable interface,

\* Arrays.sort() method can sort all the objects in the array in natural order. if, the objects are comparable type.

\* If the objects are not comparable type then arrays.sort() method throws classCastException.

NOTE:

all the objects in the array must be of same type.

case(i)

```
public class Pen {  
    public static .  
        String ink;  
        double price;  
        //constructor  
        Pen (String ink, double price)  
    {  
        this.ink = ink;  
        this.price = price;  
    }  
}
```

```

import java.util.Arrays;
public class PenDriver {
    public static void main (String [] args)
    {
        Pen [] pens = { new Pen ("black", 10),
                        new Pen ("red", 15),
                        new Pen ("Blue", 5) };
        Arrays.sort (pens); // CCE class cast exception
        for (Pen e : pens)
            System.out.println (e.ink + " " + e.price);
    }
}

```

**Conclusion:**

Here The pen is not a comparable type  
So, we get Class Cast Exception.

natural order  
while writing  
it self we are  
deciding to compare  
the object by which  
property . according  
to that comparable  
will compare

**case (ii)**

```

public class Marker implements Comparable // now Marker
{   class become
    String ink;
    double price;
    // constructor
    Marker (String ink, double price)
    {
        this.ink = ink;
        this.price = price;
    }
    // giving implementation to the compareTo (Object o) method
    abstract // public "int compareTo (Object o)" { // method from
    method // according to price comparable interface.
    Marker temp = (Marker)o; // down casting
    if (this.price == temp.price)
        return 0;
    else if (this.price > temp.price)
        return 1;
    else
        return -1;
}

```

```
A import java.util.Arrays  
public class MarkerDriver  
{  
    public static void main(String[] args)  
    {  
        Marker[] markers = { new Marker("black", 10),  
                             new Marker("blue", 7),  
                             new Marker("red", 5) };  
  
        Arrays.sort(markers);  
  
        for (Marker e : markers)  
            System.out.println(e.ink + ", " + e.price);  
    }  
}
```

Comparable

A

## By java.util.Comparator;

\* Comparator is an interface, which has an abstract method compare (Object, Object)

\* We can use comparator for the following <sup>reason</sup>

**REASON 1:** \* To sort an array or collection of objects, which is having non comparable objects.



\* We can sort an array of objects of non comparable type by using a method

Arrays.sort (array-to-be-sorted, instance-of-comparator);

**REASON 2:** we can use compare to sort an objects in different order other than Natural order.



**Example:** Let us consider an Employee object which is of comparable type, if the array of Employees are sorted there will be arranged in the natural order of compareTo (Object) method of Comparable interface let us say it is sorted based on employee n.

If the requirement demands to sort the employees according to the salary or any other property. In this case we can take the help of comparator.

## STEPS TO USE COMPARATOR:

**STEP 1:** Design a concrete implementing class for the comparator interface.

**STEP 2:** provide implementation to compare (object, object) method

**STEP 3:** call Arrays.sort (array-to-be-sorted, instance-of-comparator);

28/4/22

## Two Dimensional Array: (2D-Array)

Syntax To create A reference variable to store the address of 2D array

```
datatype[][] variable;
```

Syntax to create 2D array

Object :

```
new datatype [row-size][col-size];
```

NOTE:

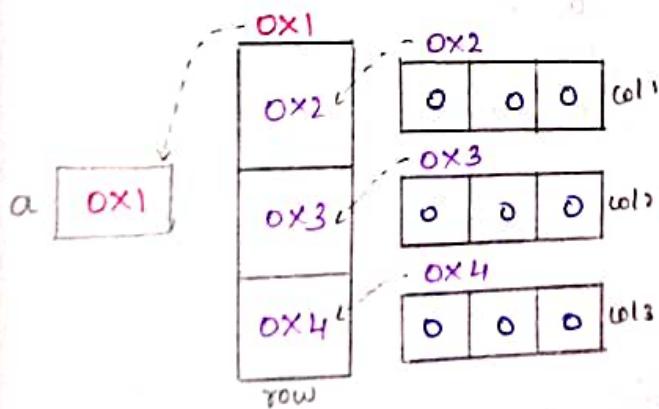
Row size is mandatory.

Column size is optional.

If both row size and column size is provided complete 2D array is created.

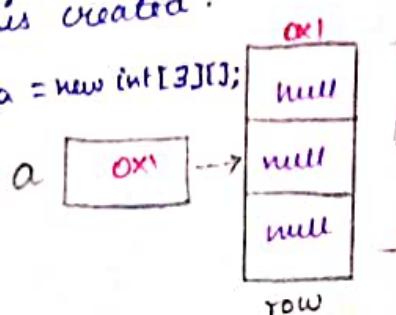
eg:

```
int[][] a = new int[3][3];
```



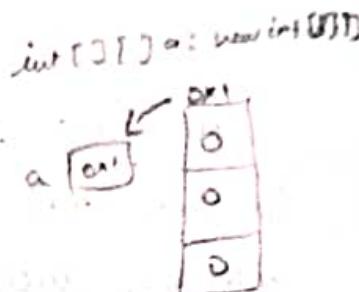
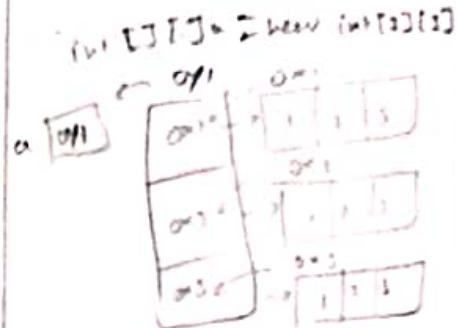
If the column size is not provided. Then, partial 2D array is created.

```
int[][] a = new int[3][];
```



→ array of int[]

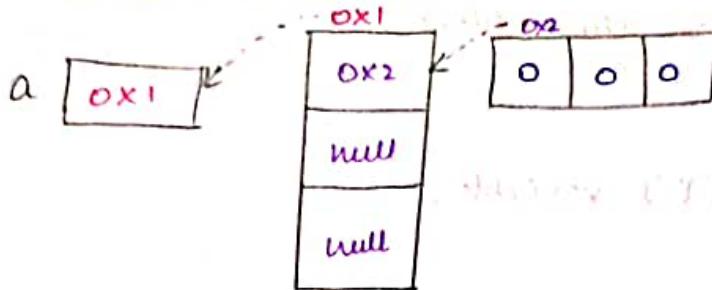
we can't  
go for 6th  
element of 2nd  
row with 6x3  
matrix position.



A

The row object should can be created separately  
as follows

$a[0] = \text{new int}[3];$



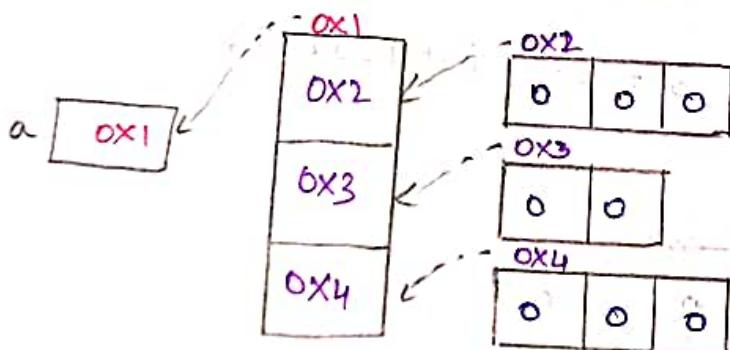
In java 2D arrays, we can design a array  
in such that every row can have different number of  
columns. Such an array is known as jagged array.

eg:  $\text{int}[\text{ }][\text{ }] a = \text{new int}[3][\text{ }];$

$a[0] = \text{new int}[3];$

$a[1] = \text{new int}[2];$

$a[2] = \text{new int}[3];$



## Lambda Expression: (feature of Java 8)

We can provide the implementation directly without writing our concrete class.  
The concrete class is called by compiler implicitly.

Lambda expression help us to provide implementation of an abstract method of an functional interface. So, that we no need to create the separate concrete class to provide the implementation for that abstract method of an interface.

### What is functional interface?

~~Lambda~~ An interface which has only one abstract method is called functional interface.

### What is the advantage of Lambda expression?

- \* To provide the implementation of functional interface

- \* Less coding

### Components of Lambda expression:

#### List of arguments:

It can be empty or non-empty as well.

#### Arrow operator:

It is used to link arguments-list and body of expression.

#### Body:

It contains expression and Statement for the abstract method.

Why we need interface

eg: 1

interface Test

{ void demo(); }

3

to call the  
method of Test.  
interface is defined  
so that create  
our concrete class  
and implement  
Test interface  
and give implem-  
entation of demo  
and create object  
for that class.

class A implements Test

3

or demo();  
S-o. reusability

3

class Driver

{ public static void main()

{ Test t = new

A();

t.demo();

3

eg: 2

Comparator

A

Syntax: for Lambda Expression:

([list-of-arguments])  $\rightarrow \{ //\text{statements} \}$ ;

e.g.:  $\text{for } \lambda \text{ expression}$

Execution Environment is destroyed  
when the function exits  
the scope of the function definition

Abstract Syntax Tree is used

for returning values and creation  
of functions and return of a  
function definition

Qualified name  
is mapped to memory  
location of variable to use

if there happens to be  
multiple definitions of same name

then the one which is closer to  
the current scope is used

multiple definitions of same name  
are placed based on LIFO

new variables are given to different  
multiple instances of same name

for

## File Handling:

\* In java we go for file handling to read the data from the file and to write the data <sup>into</sup> from the file.

\* By using file handling we can store the data output of the java program

## File InputStream:

\* InputStream is the abstract class which is used to read a data as a ~~data~~ input stream of byte.

\* This class is defined in `java.io` package

## Sub classes:

\* `audioInputStream`

\* `ByteArrayInputStream`

\* `FileInputStream`

\* `FilterInputStream` etc.,

## FileInputStream class:

\* `FileInputStream` is used to read the data from the file in the form of byte streams.

\* This is the subclass for `InputStream` class.

## constructor of `FileInputStream` class:

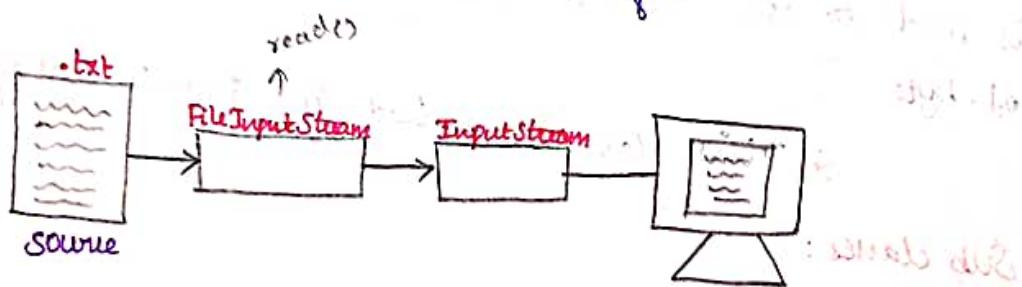
\* `FileInputStream(File file)`

\* `FileInputStream(String filePath)`

## A Important Methods:

int read() - This method reads a byte of data from input Stream  
return type      method sign  
This method returns -> once file reaches end  
return type is int

void close() - This method close file input Stream and releases any stream resources associated with close the file.



Example:

```
import java.io.FileInputStream; // Java file handling
import java.io.IOException;
public class Read {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("C:\\" + "text.txt");
        int num = fis.read();
        while (num != -1) {
            System.out.print((char) num);
            num = fis.read();
        }
        fis.close();
    }
}
```

## OutputStream:

- \* OutputStream is a abstract class which is used to write an output to the destination.
- \* This class is defined in java.io.package

## FileOutputStream:

- \* FileOutputStream is a sub class of java.io.OutputStream class
- \* This class is defined in java.io package.
- \* This class is used to write a data to the file

## Constructors of FileOutputStream:

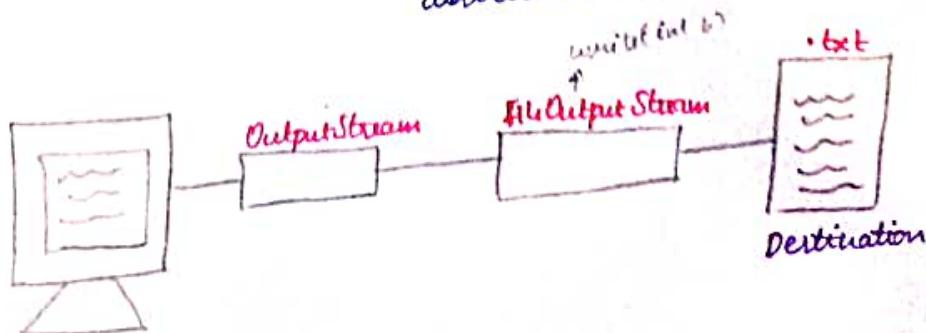
- \* FileOutputStream(File file)
- \* FileOutputStream(File file, boolean append)
- \* FileOutputStream(String filePath)
- \* FileOutputStream(String filePath, boolean append)

## Important Methods:

void write(int b) - writes the specified byte to this file output stream.

void write(byte[] b) - writes b.length bytes from the specified byte array to this file output stream.

void close() - This close file output stream and releases any stream resources associated with closed the file.



## A Example:

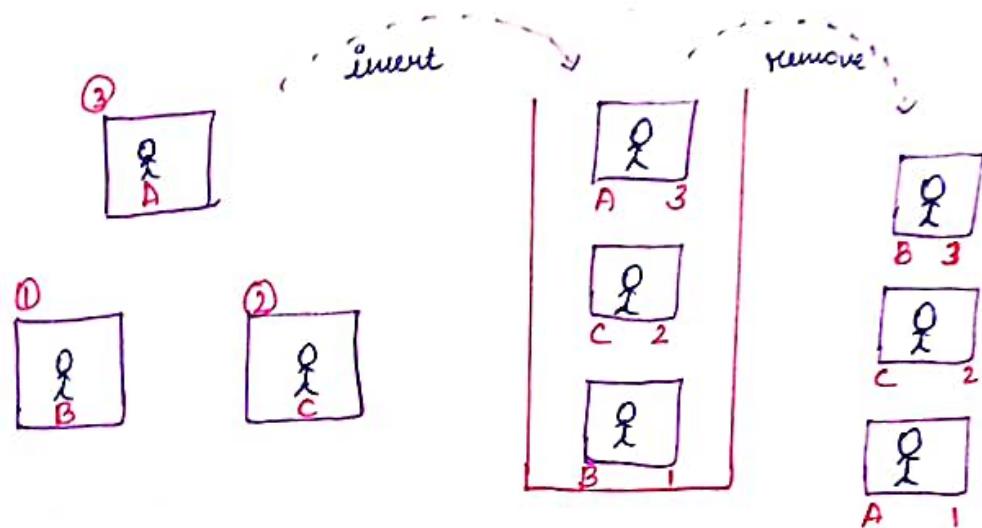
```
import java.io.FileOutputStream;
import java.io.IOException;

public class Write {
    public static void main (String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream ("c:\\text\\apple.txt", true);
        fos.write(65);
        fos.write(114);
        fos.write(114);
        fos.write(108);
        fos.write(101);
        fos.flush();
        fos.close();
        System.out.println("written");
    }
}
```

## OUTPUT:

Apple written

Stack implementation will have only one end called as "top".  
The insertion, accessing and removal is done with the help of 'Top'.



## Data Structure and Collections

\* Data structure provides a methodology of to store and access multiple data together

\* Data structure is purely organization of memory and provides mechanism to store and access multiple data together in different styles.

eg: Arrays, List, sets, Trees, Graphs, Maps, etc., are different types of data structure that helps to store multiple data and access that.

### List :-

List is a data structure which is used to store multiple items (or) elements.

#### Characteristics:

\* List maintains the order of insertion of elements.

\* In list, we can store duplicate elements.

\* It does not have any removal order and access order.

\* It preserves the insertion order.

### Set :-

Set is a data structure which is used to store multiple elements.

#### Characteristics:

Set does not allow duplicate elements.

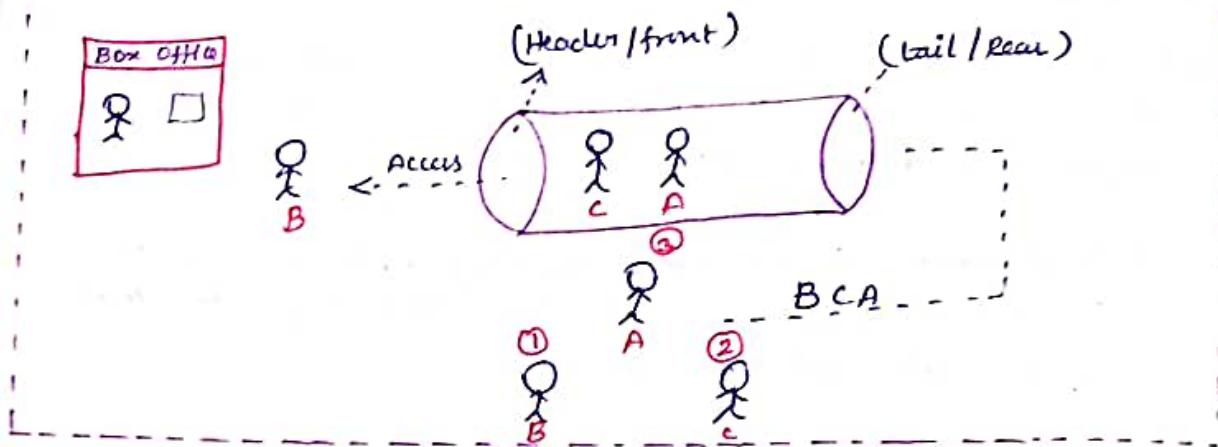
### Queue:

queue is a data structure which is used to store multiple elements.

#### Characteristics:

\* It preserves the insertion order

\* we can access the elements in first in first out



- \* In queue data structure, the element inserted first should be removed first (FIFO)
- \* The elements inserted at the last can be removed only at the last. (LIFO)
- \* Therefore in queue the order of removal is equal to order of insertion.

Explanation for diagram:-

Therefore the elements B, C, A are getting inserted in order of  $B \rightarrow 1$ ,  $C \rightarrow 2$ ,  $A \rightarrow 3$  and also they will be exiting in the same order B, C, A.

Stack:-

It is a Data structure which is used to store multiple objects characteristics:-

It preserves insertion order.

It can be removed or accessed in the reverse of its insertion order.

That is the element inserted first can be removed only at the last. (FILO).

The element inserted at last can be accessed or removed first (LIFO).

## LIST:

List is an interface defined in java.util package

List is a sub interface of collection interface. therefore it will have all the methods inherited from collection



Methods of List interface: (concrete class for List)  
(ArrayList, LinkedList)

|                          | Purpose               | Signature                                                                                                                                 |
|--------------------------|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| * In<br>be one           | to add<br>elements    | inherited:<br>add (Object) // collection<br>addAll (Collection) // collection<br>its own<br>add (int, Object)<br>addAll (int, Collection) |
| * Th<br>at .             | to remove<br>elements | inherited:<br>remove (Object) // collection<br>removeAll (Collection) // collection<br>retainAll (Collection) // collection<br>clear()    |
| * T<br>he<br>new<br>Expd |                       | its own<br>remove (int index)                                                                                                             |
| on<br>i<br>star<br>ch    | to access<br>elements | inherited:<br>iterator() // iterable<br>its own<br>listIterator()<br>listIterator (int index)<br>get (int index)                          |
| Miscellaneous            |                       | inherited:<br>size()<br>isEmpty()<br>toArray()<br>equals (Object)<br>hashCode()                                                           |

NOTE:

implementing classes of List Interface.

ArrayList

LinkedList

vector

### ArrayList:

It is a concrete implementing of List interface

ArrayList follows all the characteristics of list

\* It preserves the insertion order

\* duplicates are allowed

\* can insert, access or remove in any order  
with the help of index.

### constructors of ArrayList:

ArrayList()

It creates an empty ArrayList object  
with the default capacity (default capacity is 10)

ArrayList(int initialCapacity)

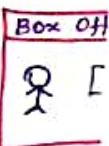
It creates an empty ArrayList object  
with the given capacity.

ArrayList(Collection)

It constructs an ArrayList and copies  
all the elements present in the given collection into  
the new ArrayList in the same order written by  
the collections iterator.

To access the element in the ArrayList:

We can access the elements in the arrays with the help of

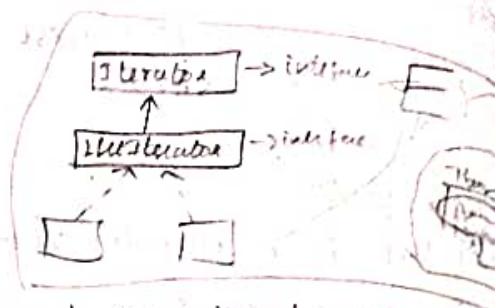


Iterator

ListIterator

getMethod

for each loop



Iterator:

Iterator hierarchy is defined in java.util package.

Iterator is used to access the elements present in the collection object.

Iterator interface provides the following abstract methods.

hasNext()

next()

remove()

hasNext():

hasNext() method is used to check whether

there are more elements for iteration. If yes, it returns true, if not it returns false.

next():

next() method returns an element pointed by the cursor and moves the cursor forward.

remove():

It removes an element from the collection which was most recently returned by the iterator.

**NOTE:**

We can access the elements of any collection with the help of iterator by creating an object of Iterator type.

We can create an object of Iterator type by calling iterator() of the collection object.

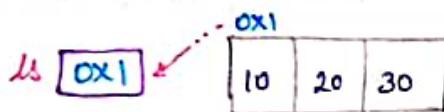
Eg:

```
ArrayList ls = new ArrayList();
```

```
ls.add(10);
```

```
ls.add(20);
```

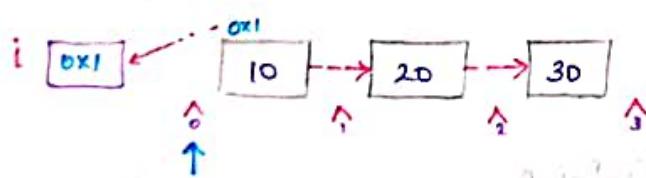
```
ls.add(30);
```



To access the elements of ArrayList ls using Iterator

**Step 1:** Create an iterator object by calling iterator() method in the ArrayList.

```
Iterator i = ls.iterator();
```



**Step 2:** Call the methods of Iterator, next() to access the element.

```
while (i.hasNext())
{
    System.out.println(i.next());
}
```

## NoSuchElementException:

In the Iterator, when we try to access an element using next() method and if there is no element present, we get NoSuchElementException.



eg:

```
class Sim  
{  
    String service-provider;  
    double call-rate;  
    Sim (String service-provider, double call-rate)  
    {  
        this.service-provider = service-provider;  
        this.call-rate = call-rate;  
    }  
}
```

- \* In queue  
be removed
- \* The  
at the  
end
- \* The  
insert  
insert

xplained

```
@Override  
public String toString()  
{  
    return "service provider :" + service-provider;  
}
```

order

in

ask

I

have

```
class SimDriver
```

```
{ public static void main (String [] args) {  
    ArrayList ls = new ArrayList ();  
    ls.add (new Sim ("Airtel", 0.50));  
    ls.add (new Sim ("jio", 0.75));  
    ls.add (new Sim ("vi", 0.37));  
}
```

```
Iterator i = ls.iterator();
```

```
System.out.println (i.next());
```

```
System.out.println(i.next());  
System.out.println(i.next());  
System.out.println(i.next());
```

{

}

### OUTPUT:

Service Provider: Airtel

Service Provider: jio

Service provider: vi

NoSuchElementException.

**Ques** To overcome this exception first we have to check whether the element is present in the collection/ not. by using hasNext().

hasNext() method will check and return true if the element is present in collection else return false.

eg:

```
class SimDriver  
{  
    public static void main (String[] args) {  
        ArrayList ls = new ArrayList();  
        ls.add (new Sim("Airtel", 0.50));  
        ls.add (new Sim("jio", 0.75));  
        ls.add (new Sim("vi", 0.37));  
        Iterator i = ls.iterator(); // object creation  
        while (i.hasNext()) {  
            System.out.println(i.next());  
        }  
    }  
}
```

### Tracing

- (i) true  
S.O.println(i.next())  
Airtel
- (ii) true  
S.O.println(i.next())  
jio
- (iii) true  
S.O.println(i.next())  
vi
- (iv) false //loop end

5/5/22

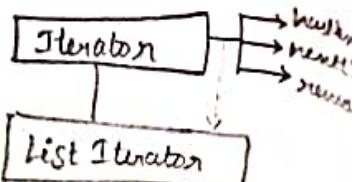
## Disadvantage of Iterator:

- \* we can iterate only once
- \* we can't access the elements in the reverse order
- \* we can only remove the elements, we can't add or replace the elements.

### List Iterator

#### ListIterator():

ListIterator() is a method that belongs to List interface, ListIterator() method creates an object of ListIterator type.



### List Iterator interface:

ListIterator is a sub interface (child) of Iterator interface, it is defined in java.util package.

#### Methods of ListIterator:

hasNext() to check whether next element to be accessed is present or not.

next() It gives the element, and moves the cursor forward.

hasPrevious() to check whether previous element to be accessed is present or not.

Previous() It gives the previous element pointed by the cursor and moves the cursor backward.

remove(Object) It removes the current object pointed from the collection, which is under iterator.

`add (Object)` It is used to add new object in the collection.

**NOTE :**

`listIterator()` method is overloaded.

`listIterator()`

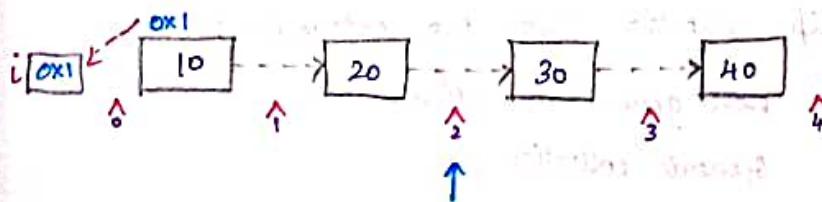
creates an `ListIterator` object type, and cursor will be pointing to first element.

`listIterator (int)`

creates an `ListIterator` object type, and cursor will be pointing to the position provided.

Eg:

`ListIterator l = ls.listIterator (2);`



`System.out.println(i.next()); //30`

## For Each Loop:

Syntax:

```
for (type variable : reference of collection/Array)  
{  
    // statements.  
}
```

eg:

```
for (Object o : u)  
{  
    System.out.println(o);  
}
```

NOTE:

We can classify collection into two categories:

- : non-generic collection
- : Generic collection

Non-Generic collection:

It is a heterogeneous (different type) collection of elements.

Every element is converted and stored as java.lang.Object class type.

Example in laptop

for heterogeneous in collection part  
eg. min. class

Generic collection:

**Homogeneous collection:**  
It is homogeneous collection of elements, (collection of same type of element)

Syntax to create Generic collection:

Syntax to create reference variable for

Generic collection.

collection-type (Non-primitive Datatype) > variable

eg:

ArrayList<Integer> ls;

Syntax to create Generic collection object:

`new Collection_name < Datatype > ();`

`new ArrayList < Integer > ();`

`ArrayList < Integer > ls = new ArrayList < Integer >();`

ArrayList where we can store only integers

From JDK 7 onwards,

`ArrayList < Integer > ls = new ArrayList < > ();`

#### NOTE:

\* The elements are not converted to `java.lang.Object` class type, instead they are tried to be converted for the generic Type.

\* The return type of the element in Generic collection will be same as the given generic type.

\* In an ArrayList if we try to insert any element or remove element then the remaining elements will shift its position. so, the time taken to insert or remove an element will be more in ArrayList.

\* In order to overcome this disadvantage we go for LinkedList.

## Linked List:

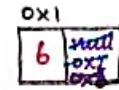
- \* LinkedList is an implementing class for List interface.
- \* It is defined in java.util package.
- \* In a Linked List all the elements are stored in the form of nodes.
- \* Node contains two parts, one is used to store the data/value/elements and other one is used to store the address of next node.
- \* Every node has an address.

eg:

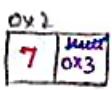
```
import java.util.LinkedList  
class LinkedList1  
{  
    public static void main(String[] args)  
    {
```

```
        LinkedList l = new LinkedList();
```

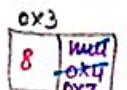
```
        l.add(6);
```



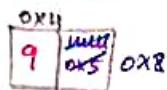
```
        l.add(7);
```



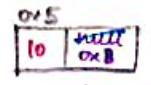
```
        l.add(8);
```



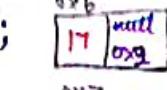
```
        l.add(9);
```



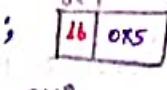
```
        l.add(10);
```



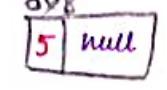
```
        l.add(1,17);
```



```
        l.add(4,16);
```



```
        l.add(5);
```



```
}
```

17 b  
16 7 8 9 10  
0 9 2 3 4 4

16 11 7 8 16 9 10 5

6/5/22

## SET

### Set :

- \* Set is an interface, it is a sub interface of collection, therefore, all the methods of collection is inherited.
- \* Set is defined in java.util package.

### characteristics of set :

- \* It is an unordered collection of elements (The order of insertion is lost).
- \* Set does not allow duplicate elements.
- \* Set does not have indexing. Therefore, we cannot access, insert or remove based on index.
- \* we can access the elements of set only by using iterator()

### concrete implementing classes of set :

\* HashSet

\* LinkedHashSet

\* TreeSet

### HashSet :

- \* It is a concrete implementing class of set interface.

- \* It has all the methods inherited from Collection interface.

## Characteristics:

- \* It is unorderd
- \* Duplicates are not allowed
- \* Indexing is not possible
- \* Only one null is allowed.

## Constructors:

1) HashSet():

creates an empty HashSet objects.

2) HashSet(Collection c):

creates an HashSet and will copy all the elements of the collections into the HashSet.

## Methods:

To add an element

add(Object)

addAll(Collection)

To search an element

contains(Object)

containsAll(Collection)

To remove an element

remove(Object)

removeAll(Collection c)

(remove the common element present in the passed collection)

retainAll(Collection c)

(remove all the element except the common element present in the passed collection).

clear()

iterator()

using for each loop

To access the elements

Miscellaneous

isEmpty()

size()

toArray()

equals()

hashCode()

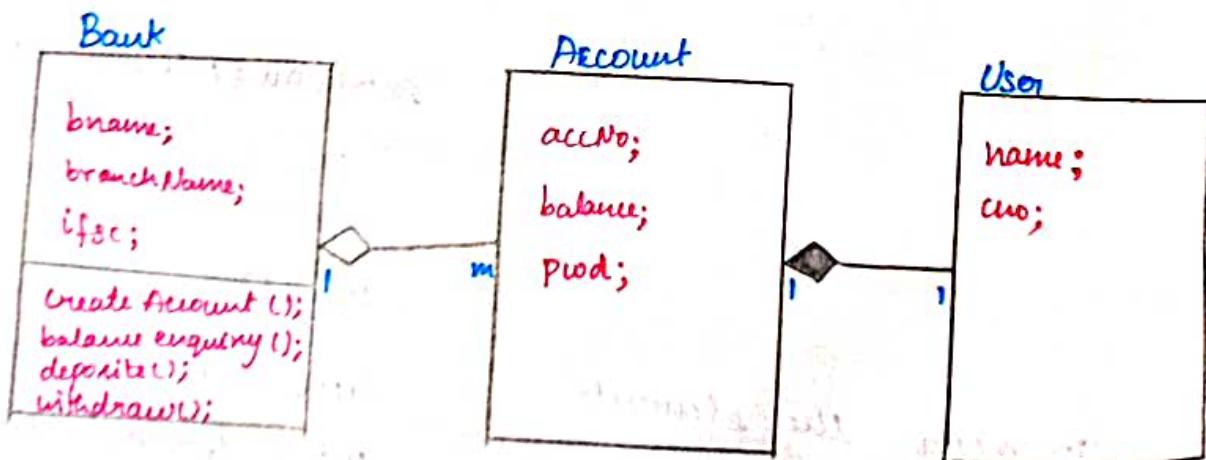
### NOTE:

\* HashSet will identify the duplicate elements with the help of equals() and hashCode() methods of an Object.

\* If we don't override the equals(Object o) and hashCode() methods it will identify the duplicate elements based on the address of an Object.

If wants to identify the duplicate elements based on the states of an Object, Then it is mandatory to override both equals(Object o) and hashCode() method.

\* That is why it is always highly recommended to override both equals() and hashCode() method together.



is sup  
collection, set.

## TreeSet :

- \* It is a concrete implementing class of Set interface
- \* TreeSet will also have all the methods of Collection interface.

## characteristics:

- \* The elements will be sorted by default.
- \* The elements to be added in TreeSet must be comparable type , else we get ClassCastException.
- \* All the elements in TreeSet should be of same type (**Homogeneous**), if not we get ClassCastException because to sort the element it compares the object / data.
- \* Duplicate elements are not allowed.
- \* Indexing is not possible. Therefore, we cannot add , remove elements using index.

## constructor :

### 1) TreeSet()

It creates empty TreeSet Object.

### 2) TreeSet(Comparator)

It creates Empty TreeSet, sorted According to the specified comparator

### 3) TreeSet(Collection)

Creates new tree set containing the elements in the specified collection , sorted according (90%<sup>b</sup>)

TreeSet will identify the duplicate elements based on .compareTo()

If the element is not of comparable type then we can store object or element with the help of comparator type interface.

```
class Student implements Comparable {
    String sname;
    int sid;
    Student () { }
    Student (String sname, int sid) {
        this.sname = sname;
        this.sid = sid;
    }
    @Override
    public String toString () {
        return sname + " " + sid;
    }
    public int compareTo (Object o) {
        Student s = (Student) o;
        return this.sid - s.sid;
    }
}
class P2 {
    public static void main (String [] args) {
        TreeSet ts = new TreeSet ();
        ts.add (new Student ("Raja", 2));
        ts.add (new Student ("Ramana", 1));
        ts.add (new Student ("Kanju", 3));
        System.out.println (ts);
    }
}
```

## MAP

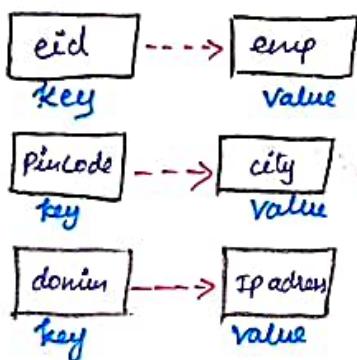
map:

map is a data structure, which helps the programmer to store data in the form of key , values pairs where every value is associated with a unique key.

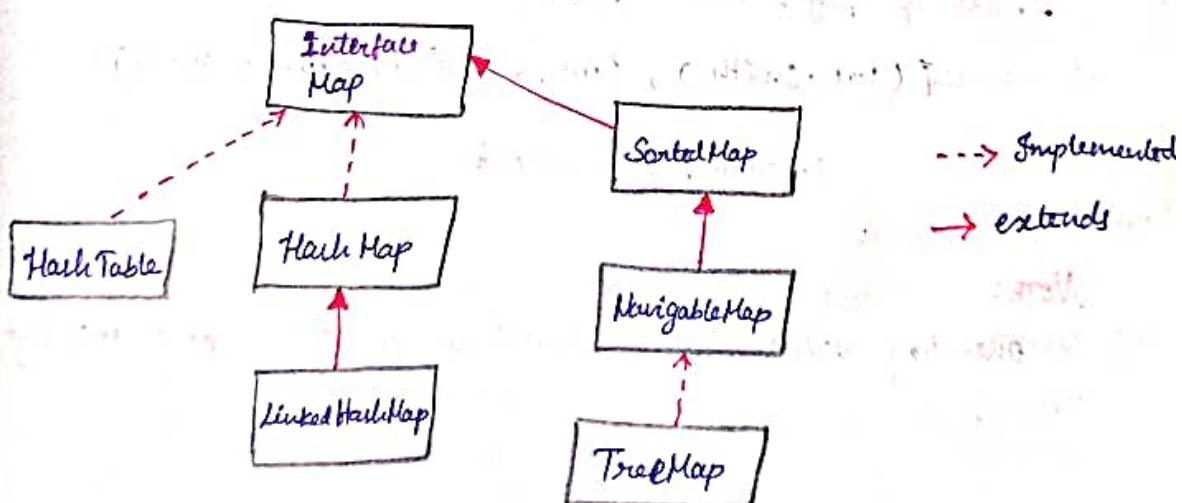
NOTE:

- \* Key cannot be duplicated.
- \* One key can be associated with only one value
- \* maps helps us to access the values easily with the help of its associated key.

eg:



### Map Interface



- \* Map is an interface in java defined in `java.util` package
- \* we can create generic map by providing the type for both key as well as value. `<key-type, value-type>, <k,v>`
- \* we can obtain 3 different views of map

- \* we can obtain a list of values from a map
- \* we can obtain a set of keys from a map
- \* we can obtain a set of key - values pairs

| key | = | Value  |
|-----|---|--------|
| 101 | = | smith  |
| 102 | = | martin |
| 103 | = | miller |

a. list of values

[smith, martin, miller]

b. set of keys

[101, 102, 103]

c. set of key - value pairs

[(101, smith), (102, martin), (103, miller)]

↓  
element / key value pair

NOTE:

one key value pair is called as an entry or a mapping

## Method of map interface :

| method signature     | purpose                                                                                                                                                      |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| put(key, value)      | add an entry to the map (key, value pair)                                                                                                                    |
| putAll(Map)          | replace the older value with a new value of an existing entry in the map                                                                                     |
| containsKey(key)     | it will copy all the entries from the given map into the current map                                                                                         |
| containsValue(value) | if the key is present or not. return type is boolean.                                                                                                        |
| remove(key)          | if the value is present it returns true, else it returns false.                                                                                              |
| clear()              | if the key is present the entry is removed from the map and the value is returned. if the key is not present nothing is removed and null is returned.        |
| get(key)             | it removes all the entries in the map                                                                                                                        |
| values()             | it is used to access the value associated with a particular key .<br>if the key is present it returns the value . if the key is not present it returns null. |
| keySet()             | it returns a collection of values present in the map return type collection<K,V>                                                                             |

`size()`  
`isEmpty()`  
`equals()`  
`hashCode()`

### HashMap:

It is a concrete implementing class of Map interface.

- \* data is stored in the form of key-value pair
- \* Order of the insertion is not maintained
- \* key can't be duplicate, values can be duplicate
- \* key can be null only once.

- \* value can be null

eg:

```
class PMS {
    Student {
        String sname;
        int sid;
    }
    student () { }
    student (String sname, int sid) {
        this.sname = sname;
        this.sid = sid;
    }
    @Override
    public String toString () {
        return sname + " " + sid;
    }
}
```

```
public class P1 {
```

```
    public static void main (String [] args) {  
        HashMap hm = new HashMap();  
        hm.put(5, new Student ("Raja", 5));  
        hm.put(1, new Student ("Ramana", 1));  
        hm.put(2, new Student ("Balaji", 2));  
        hm.put(4, new Student ("Manjuk", 4));  
        hm.put(3, new Student ("Lavanya", 3));  
        hm.put(null, new Student ("Chethana", 6));  
        hm.put(3, new Student ("Divya", 8)); //replaced  
        hm.put(null, new Student ("Karthi", 7)); //replaced  
        System.out.println(hm);  
    }  
}
```

### TreeMap:

It is a concrete implementing class of Map interface

- \* It also stores data in key-value pair
- \* It will sort the entries in the map with respect to keys in ascending order.
- \* The key in TreeMap must be Comparable Type. If it is not comparable type we get ClassCastException
- \* In TreeMap key can't be null. If it is null we get NullPointerException
- \* A value in TreeMap can be null.

7/5/22

## EXCEPTION

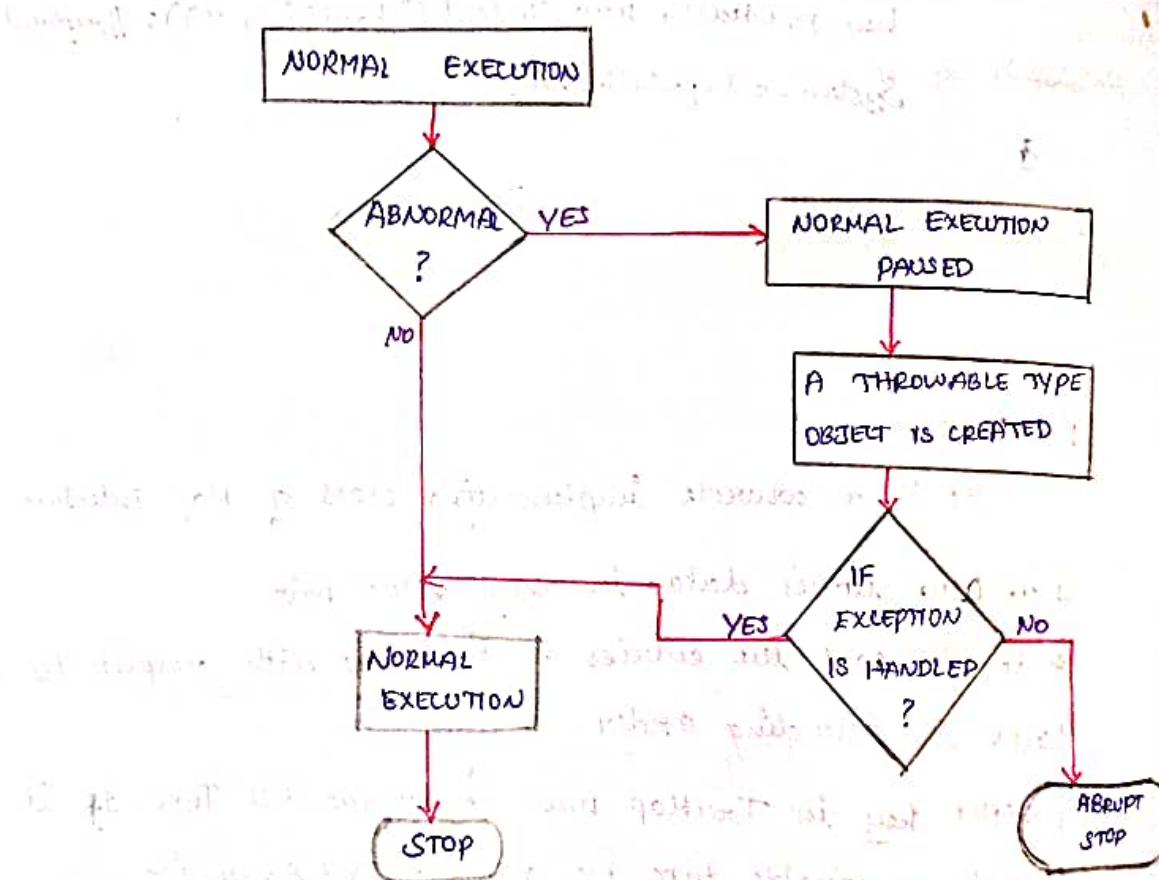
Exception:

The exception is a problem that occurs during the execution of a program (**Runtime**) when an exception occurs. The execution of program stops abruptly (**unexpected stop**).

**NOTE:** (Can't predict what will happen)

Every exception in java is a class of **Throwable type**

What happens if an exception occurs?



**NOTE:**

- \* Every Exception occurs because of a statement
- \* A statement will throw an exception during abnormal situation.

### Example:

```
import java.util.Scanner;  
class Demo  
{  
    public static void main(String[] args)  
    {  
        Scanner s = new Scanner(System.in);  
        int a, b;  
        a = input.nextInt();  
        b = input.nextInt();  
        int c = a/b; // statement might cause Exception.  
        System.out.println("The division of "+a+" "+b+" = "+c);  
    }  
}
```

}

case 1

a=5, b=10

normal situation

NO exception

case 2

a=5, b=0

abnormal situation

exception occurs.

### IMPORTANT EXCEPTION AND STATEMENTS

| Statement               | Exception                       |
|-------------------------|---------------------------------|
| a/b                     | ArithmaticException             |
| reference.member        | NullPointerExeption             |
| (ClassName) reference   | ClassCastException              |
| array-ref [index]       | ArrayIndexOutOfBoundsException  |
| String.charAt(index)    | StringIndexOutOfBoundsException |
| String.substring(index) | StringIndexOutOfBoundsException |

### CHECKED EXCEPTION:

The compiler aware exception is known as the checked exception. i.e., the compiler knows the statement responsible for abnormal situations (exception). Therefore, the compiler forces the programmer to either handle or declare the exception. If it is not done we will get an unreported compile time error.

Eg: FileNotFoundException

### UNCHECKED EXCEPTION:

The compiler unaware exception is known as the unchecked exception. i.e., the compiler doesn't know the statements which are responsible for abnormal situations (Exception). Hence, the compiler will not forced force the programmer either to handle or declare the exception.

Eg: ArithmeticException

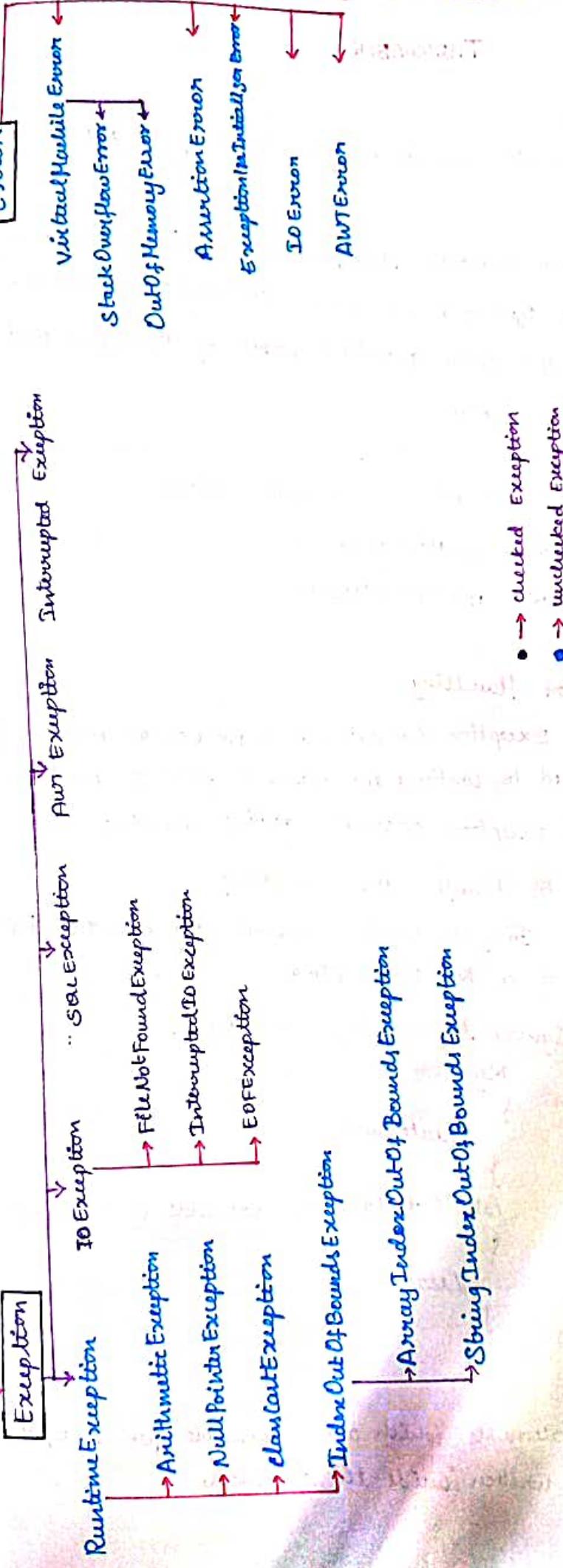
#### NOTE:

\* In Throwable hierarchy Error class and its subclasses, RuntimeException class and its subclasses are all considered as Unchecked Exception.

\* All the subclasses of the Exception class except RuntimeException are considered as checked Exceptions.

\* Throwable and Exception classes are partially checked and partially unchecked.

## Throwable



- → checked Exception
- → unchecked Exception

## THROWABLE

### Throwable:

Throwable class is defined in `java.lang.package`

### NOTE:

In the Throwable class, all the built-in classes of the Throwable type are overridden `toString()` such that it returns the fully qualified name of the class and reason for the exception.

### Important methods of Throwable class:

- \* `String getMessage()`
- \* `void printStackTrace()`

## Exception Handling

Exception handling is a mechanism used in java that is used to continue the normal flow of execution when the exception occurred during Runtime

### How to handle the Exception?

In java, we can handle the exception with the help of a try catch block.

### Syntax to use try catch block:

```
try  
{  
    //Statements  
}  
catch ( declare one variable of throwable type)  
{  
    //Statements.  
}
```

### try{} :

The statements which are responsible for exceptions should be written inside the try block

when an exception occurs,

- \* execution of try block is stopped.
- \* A throwable type obj is created
- \* The reference of throwable type obj created, is passed to the catch block.

eg:

```
try
```

```
{
```

```
    statement 1;
```

```
    statement 2; // exception occurs - - - - ->
```

```
    statement 3; stmt 3 is not executed
```

```
}
```

```
catch (variable)
```

```
{
```

```
}
```

throwable  
type  
obj is created

Reference of Throwble type object is  
thrown to the catch blocks

catch () { } :

The catch block is used to catch the throwable type reference thrown by the try block.

\* if it catches, we say the exception is handled.  
statements inside the catch block are get executed and  
the normal flow of the program will continue.

\* If it doesn't catch, we say the execution is not  
properly handled. Statements written inside the catch block  
are not executed and the program is terminated.

case 1: execution occurs not caught.

```
try
```

```
{
```

```
    10/0 // arithmetic exception occurred - - - - ->
```

```
}
```

```
catch (NullPointerException e) {
```

// not executed

AE @

```
}
```

// not executed

NOT caught by the catch block

## case 2: Exception occurred and caught

try

{

10/0; //Arithmetical exception occurs.

}

catch (ArithmeticalException e) <----- catch by the catch block

{

//executed

}

//executed

AEO

Example:

Exception occurs but not handled

```
System.out.println("main begins");
try
{
    int c=5/0;
}
catch (NullPointerException)
{
    System.out.println("Divisible by '0'
        is not possible");
}
System.out.println("main end");
```

Console:

Main Begins  
Exception in thread "main"  
java.lang.ArithmeticalException

Exception occurs and handled

```
System.out.println("main begins");
try
{
    int c=5/0;
}
catch (ArithmeticalException e)
{
    System.out.println("Divisible by '0'
        is not possible");
}
System.out.println("main end");
```

Console:

Main Begins  
Divisible by '0' is not possible  
main end

Try with Multiple catch block:

Try block can be associated with more than one catch block

Syntax:

```
try
{
}
catch(...)
```

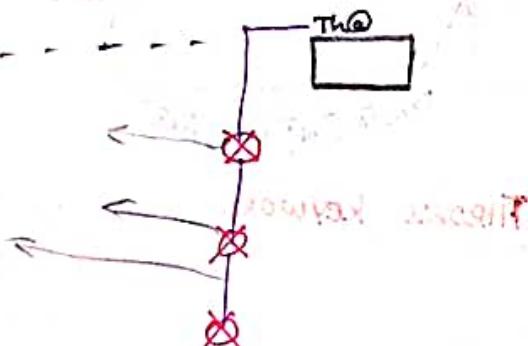
### NOTE:

The exception type object thrown from top to bottom.

### Example

#### WORK FLOW:

```
try
{
}
catch (...) {
}
catch (...) {
}
}
```



if the exception is caught by the block then try block does not throw the exception obj to the below block

#### Rule:

The order of the catch block should be maintain such that, child type should be one the top and parent type at the bottom.

#### Example :

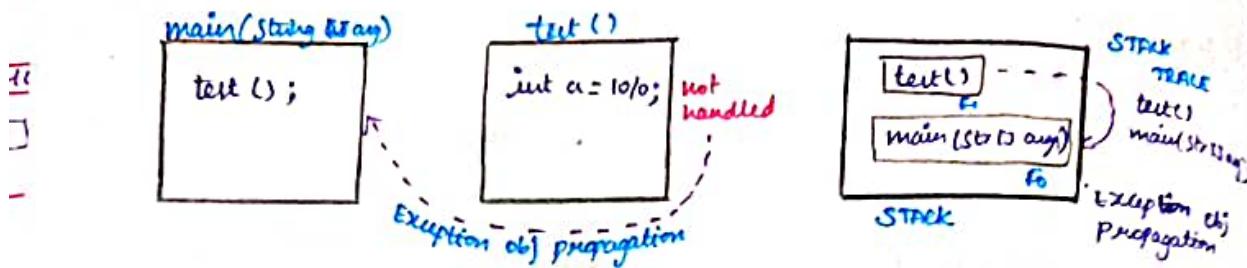
| case 1                                                                                                                                                              | case 2                                                                                                                                                               |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>try {     1/0; } catch (Exception e) {     catch (ArithmeticException e) {}</pre> <p>CTE :- parent type is declared on the top and child type is on bottom</p> | <pre>try {     1/0; } catch (ArithmeticException e) {     catch (Exception e) {}</pre> <p>CTS :- parent type is declared on the bottom and child type is on top.</p> |

### STACK TRACE

#### Stack Trace:

It provides the order in which the exception is occurred and thrown from top of the stack to the bottom of the stack.

It contains fully qualified class name of the exception, reason for the exception, method name and line number.



## THROWS KEYWORD

throws:

- \* It is a keyword
  - \* It is used to declare an exception to be thrown to the caller

**NOTE:**

throws keyword should be used in the method declaration statement.

## Syntax:

[modifier] return-type methodName ([Formal argument]) throws  
    excep1, excep2, ...

3

## 11. Statements;

3

**NOTE :**

If a method declares an exception using throws keyword, then caller of the method must handle or throw the exception. If not we will get compile time error.

Example:

| Declaration                                                | Purpose                                                                                   |
|------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| public static void sleep(long) throws InterruptedException | The purpose of this method is to pause the execution of a program for the specified time. |

**NOTE:**

If any method declares any exception then the caller should either can declare the exception or can handle the exception.

eg: case (i) only throws

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class FileDemo {
    public static void main(String[] args) throws FileNotFoundException {
        System.out.println("main start");
        m();
        System.out.println("main end");
    }
    public static void m() throws FileNotFoundException {
        System.out.println("m is called");
        openfile();
    }
    public static void openfile() throws FileNotFoundException {
        FileInputStream fis = new FileInputStream("d:\\abc.txt");
    }
}
```

case (ii) handling

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class FileDemo {
    public static void main(String[] args) {
        System.out.println("main start");
        try {
            m();
        } catch (Exception e) {
            System.out.println("Exception handled successfully");
        }
    }
}
```

```
public static void m1() throws FileNotFoundException  
{  
    System.out.println("m1 is called");  
    openfile();  
}  
public static void openfile() throws FileNotFoundException  
{  
    FileInputStream fi = new FileInputStream("D:\\abc.txt");  
}
```

## THROW KEYWORD

### throw:

It is a keyword

It is used to throw an exception manually

By using throw we can throw checked exception, unchecked exception. It is mainly used to throw the custom exception

### Syntax:

```
throw Exception;
```

```
throw new CustomException ("String");
```

### eg:-

```
class ThrowDemo  
{  
    public static void main (String [] args)  
    {  
        int a=15;  
        int b=10;  
        if (a>b)  
            throw new ArithmeticException ("manually");  
        else  
            System.out.println ("NO Exception");  
    }  
}
```

**OUTPUT:**

Exception in thread main. java. lang.ArithmeticException  
manually thrown.

**NOTE:**

If we don't handle the manually thrown exception  
we will get unreported exception.

**Custom Exception:-**

class NotValidException extends Exception

{ public String getMessage()

{ return "Not valid";

}

}

Raising and handling the not valid exception.

class Driver

{ public static void main (String [] args)

{ try {

throw new NotValidException();

}

catch (NotValidException e) {

System.out.println ("custom exception is  
handled");

}

}

## Programs :

```
public class IndexException {
    public static void main(String[] args) {
        int a[] = {};
        String s = null;
        try {
            System.out.println(s.equals("Hello"));
        --*
        int b = a[2];
        System.out.println(b);
        }
        catch (NullPointerException e) {
            System.out.println("Null Pointer Exception handled");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Exception handled");
        }
        catch (Exception e) {
            System.out.println("Exception handled ");
        }
        catch (Throwable e) {
            System.out.println("Throwable exception handled");
        }
    }
}
```

```
public class Calculator {
    public static void main(String[] args) {
        try {
            findLength(null);
        } catch (NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void findLength(String s) {
        System.out.println("start");
        System.out.println(s.length());
        System.out.println("end");
    }
}
```

}

### Creating our own Exception:

```
public class HavePatienceException extends RuntimeException
{
    public String getMessage()
    {
        return "Have a patience";
    }
}

public class OutOfServiceException extends RuntimeException
{
    return "Validity Expired";
}

if (age < 21)
    throw a new HavingException();
else if (age > 10)
    throw new PatienceException();
```