Part - II Index 01/09/23

1. oop's
2. static members
3. Non- static members
4. Constructors
5. Encapsulation
6. Inheritance
7. Package * modifier * Scanner class
8. Non- primitive typecasting
9. Polymorphism
10. Abstraction

04/08/23

Oops

* Oops stands for object Oriented Programming.
* Ooops is an one way of programming using class and objects
* Oops Consists of 4 pillars (or) principles (or) Concepts

i) Encapsulation
ii) Inheritance
iii) Polymorphism
iv) Abstraction

Advantages of oops

* Code reusability
* Data security
* Easy code maintenance
* loose coupling

class

* class is a blueprint of an object.
* class → In Java, we can have following class members

They are

i) Variable
ii) Methods
iii) Initializer
iv) Constructor

* class members are classified into 2 types
  i) Static members
  ii) Non-static members

Static members

* Static members are the class members which are prefixed with static keyword.
* Static members are also called as class members.
* In Java, we have 3 static members they are
  i) static variable
  ii) static method
  iii) static initializer.

Non-static members

i) Non-static members are the class members which are not prefixed with static keyword.

ii) Non-static members are also called as instance members (or) object members.

iii) In Java, we have 4 non-static members they are

i) Non-static variable.
ii) Non-static method
iii) Non-static initializer.
iv) Constructor.

class member	static members	Non-static members	local members
① Variables	① static variable `static datatype identifier;`	② Non-static variable `datatype identifier;`	③ local variable `datatype identifier;`

② Method	① static method
	`[modifier] static RE MN([args])`
	`{ //stmt }`
		② Non-static method
		`[modifier] RE MN([args])`
		`{ //stmt }`
			-

③ Initializer	① static Initialization block `static { //stmt }`	② Non-static Initialization block `{ //stmt }`	-
④ Constructor	-	`[modifier] classname([args])` `{ //stmt }`	-

Example

```
class classMembers
{
  // static variable
  static int id;
  // Non-static variable
  int age;
  // static method
  public static void sheela()
  {
    // local variable
    int phone;
```

```
}
//Non-static method
public void mala()
{
    S.o.pln("from mala");
}
//static initializer
static
{
    S.o.pln("static Initialization block");
}
//Non-static Initializer
{
    S.o.pln("Instance Initialization Block");
}                    (or) object members
//Constructor
public classMembers()
{
    S.o.pln("I'm a Constructor");
}
}.
```

05/09/23

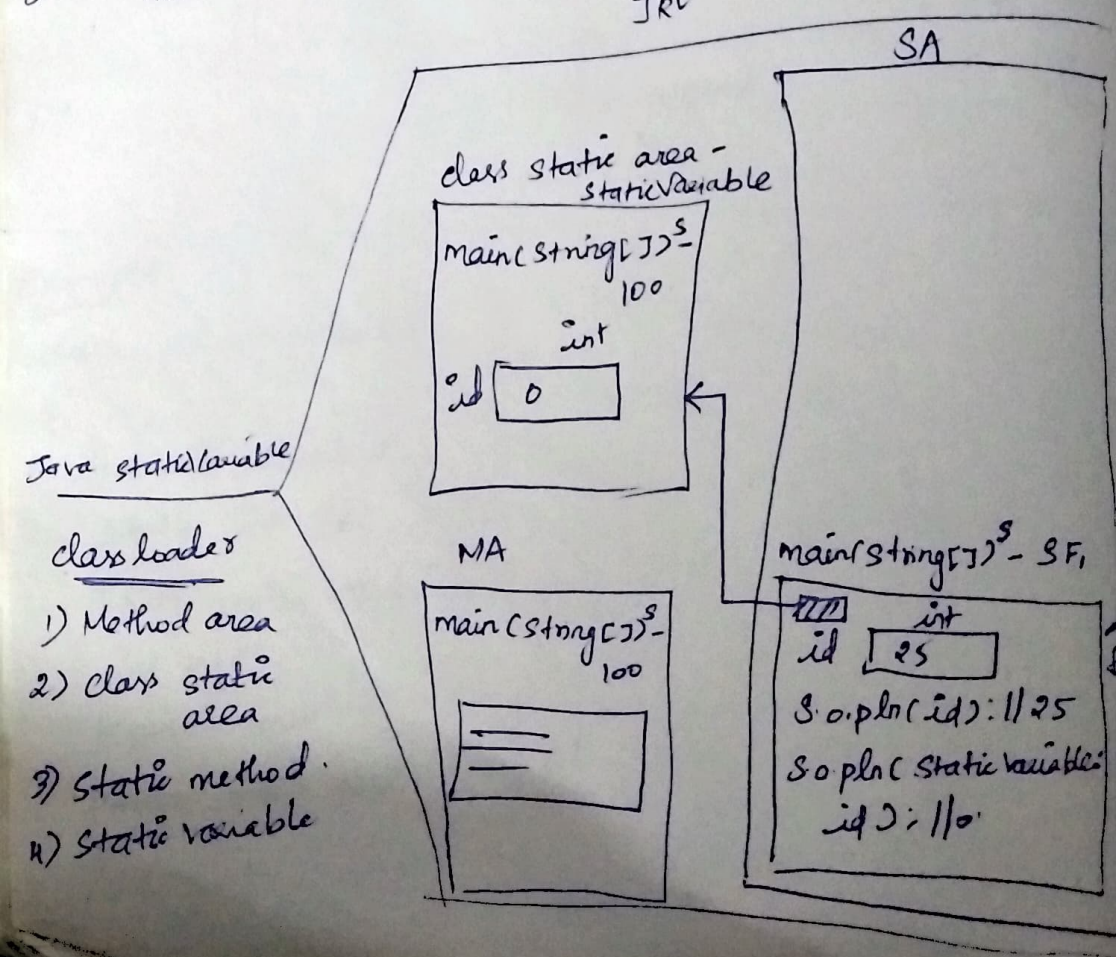
Static variable

* class variable which are prefixed with static keyword are known as static variable
* Static variables are stored in class static area
* We can access static variable using class name

```
class StaticVariable
{
    static int id;
    Public static void main(String[] args)
    {
        int id = 25; "local var id"
        S.o.pln(id); //25
        S.o.pln(" static var id: " + staticVariable.id);
    }
}
```

. → Dot / Access / member selection operator

JVM → class loader → main(String[])



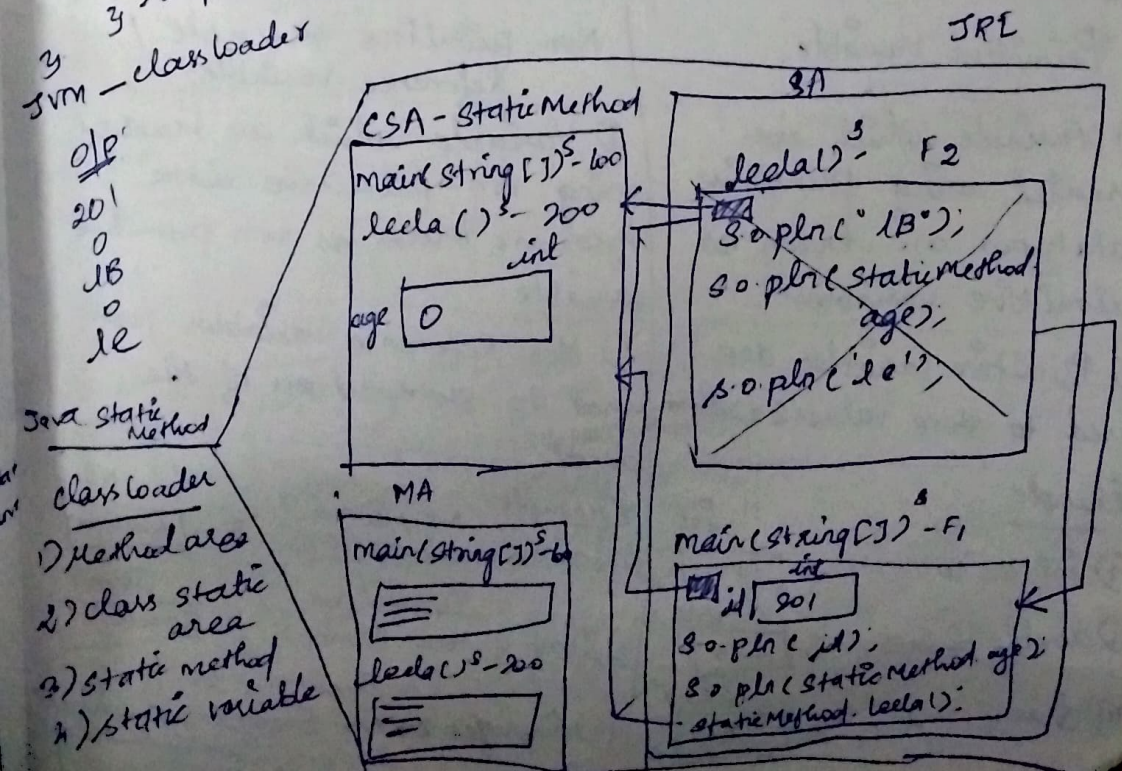
Static Method

* Methods which are prefixed with static keyword is called static methods
* Static methods are stored in class static area
* We can access static methods directly or using classname

Note

From static block (or) static context, we can access static members directly.

```
class staticMethod
{
    static int age;
    public static void main(String[] args)
    {
        int id = 201;
        S.o.pln("local var id : " + id); //201
        S.o.pln("static var age :" + staticMethod.age); //0
        leela();
    }
    public static void leela()
    {
        S.o.pln("leela Begin");
        S.o.pln("static var age:" + staticMethod.age);
        S.o.pln("leela end");
    }
}
```



06/09/23

Non-static members

* class members which are not prefixed with static keyword are known as non-static members
* Non-static members are also called as object members (or) instance members
* Non-static members are stored in heap area.
* We can access non-static members using object reference only.

Object

* Object is a real-time entity which has states and behaviours
* In Java, object is a memory location created in the cheap area to store non-static members.

Syntax

new Constructor

⇓

new classname();

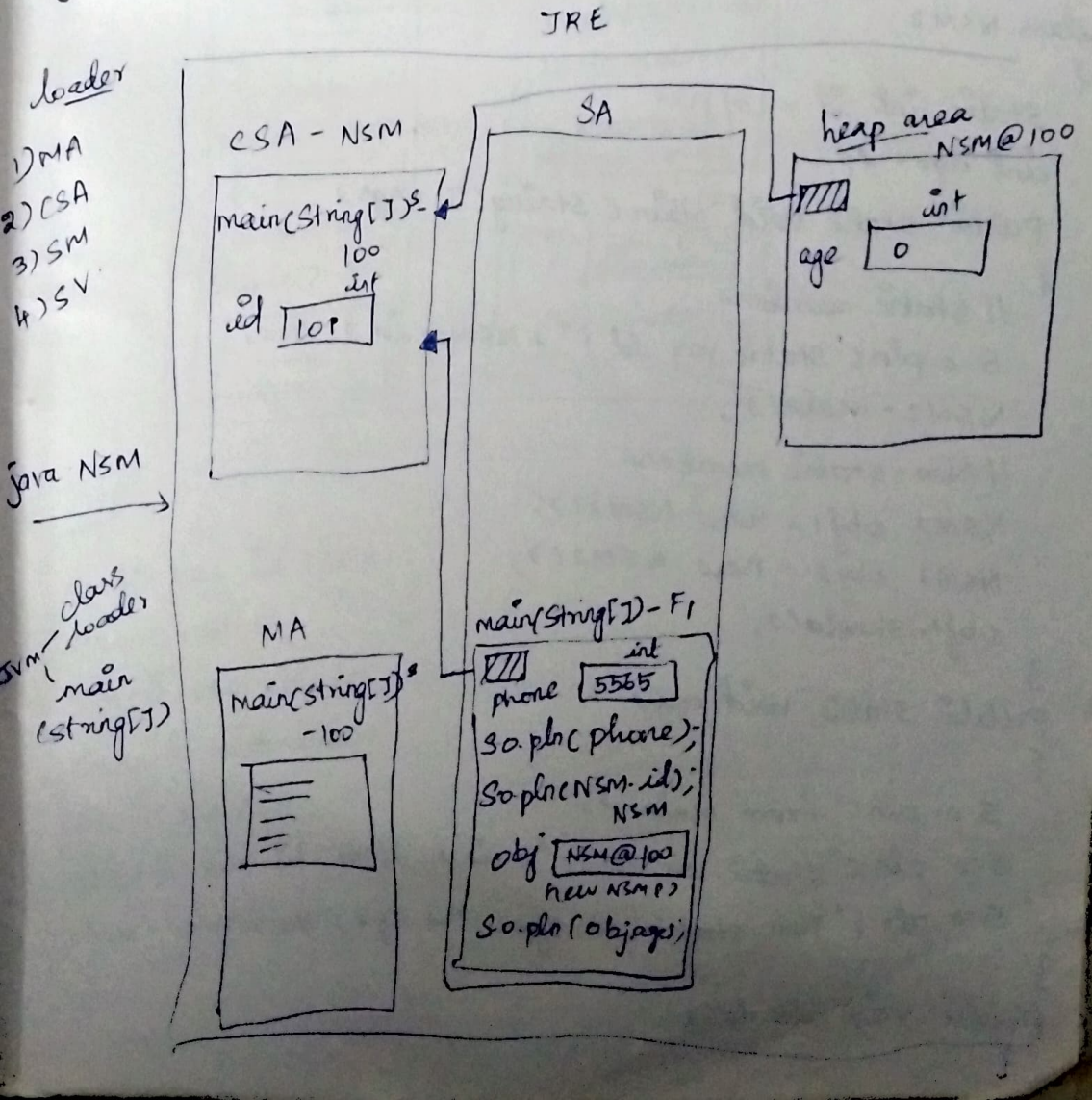
* We can store the address of object using non-Primitive variables (or) reference variables

Primitive variable	Non-primitive variable / Reference variable
i) Variable which are created using primitive datatypes are known as primitive variables.	i) Variable which are created using non-primitive data types are known as non-primitive variable.
ii) Primitive variables are used to store values (or) data	ii) Non-Reference variables are used to store address of the object
Example	Example
i) int a=10; a: int [10]	i) NSM x=new NSM(); NSM@100 x: NSM [NSM@100]
ii) double b=20.5; b: double [20.5]	ii) Demo y=new Demo(); Demo@200 y: Demo [Demo@200]
iii) char c='A'; c: char ['A']	iii) Employee z=new Employee(); Employee@300 z: Employee

Example

```
class NSM
{
    static int id=101;
    int age;
    public static void main (String[] args)
    {
      int phone= 5565;
      // Accessing local var directly
      S.o.pln("local var phone:" + phone);
      // Accessing static var using classname
      S.o.pln(" Static var id :" + NSM.id);
      // Accessing non-static var using Object reference
      S.o.pln("Non-static var age:" + obj.age);
      NSM obj= new NSM();
    }
}
```

o/p
5565
101
0



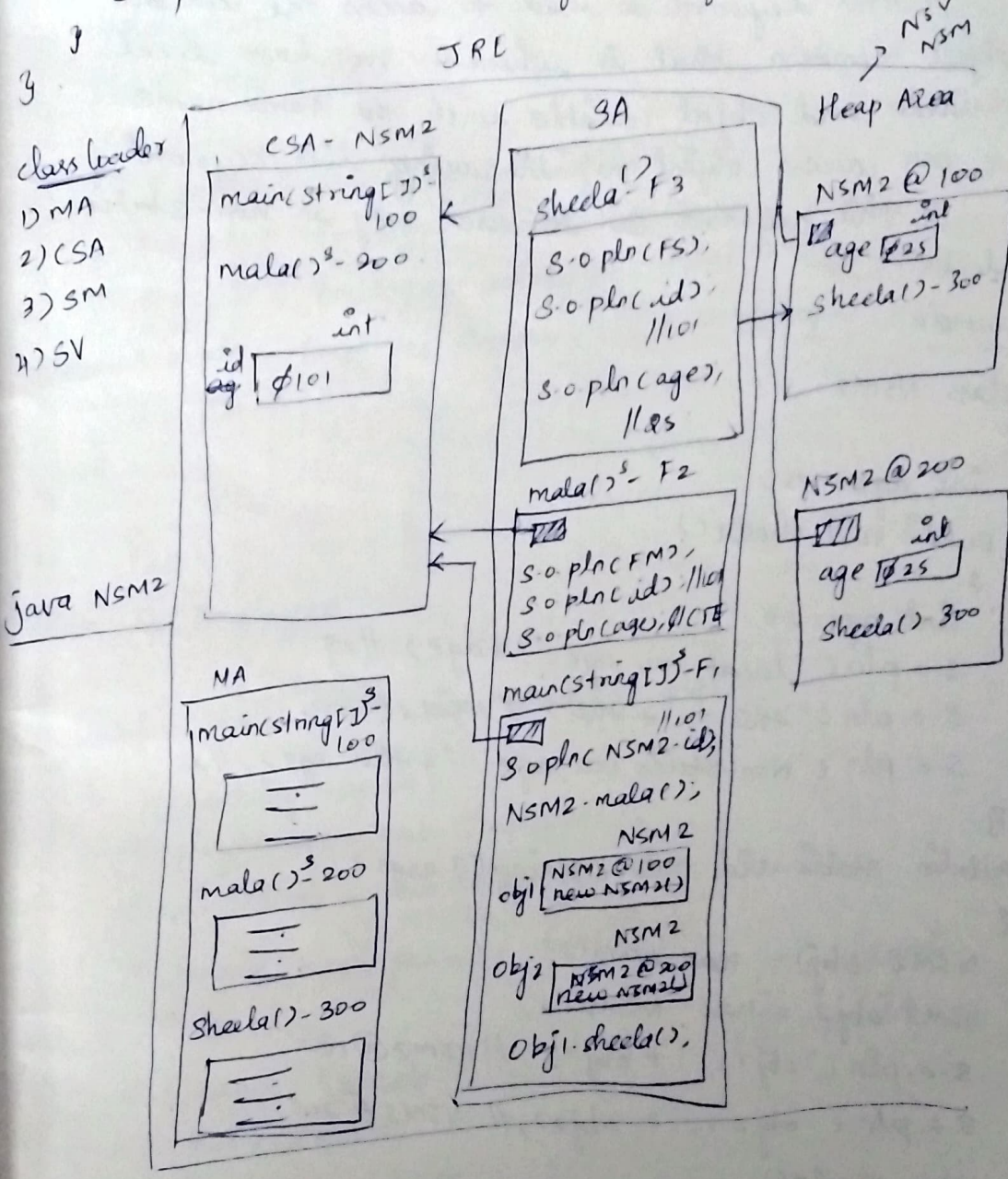
07/9/23

Methods

Static method	Non-static method
* Methods which are prefixed with static keyword is known as static method	* Method which is not prefixed with static keyword is known as non-static method
* Static methods are stored in class static area	* Non-static methods are stored in heap area
* We can access static methods using class name	* We can access non-static methods using object reference
* Inside static block, we can access only static members directly, that is we cannot access non-static directly inside static context	* Inside non-static context, we can access both static members and non-static members directly

```
class NSM2
{
  static int id = 101;
  int age = 25;
  public static void main (String[] args)
  {
    // static members
    S.o.pln("static var id : "+NSM2.id); //101
    NSM2.mala();
    // Non-static members
    NSM2 obj1 = new NSM2();
    NSM2 obj2 = new NSM2();
    obj1.sheela();
  }
  public static void mala()
  {
    S.o.pln("from mala");
    S.o.pln("static var id :: "+id); //101
    S.o.pln("Non-static var age :- "+age); //CTE
  }
  public void sheela()
  {
```

```
S.o.pln("From sheela");
S.o.pln("static var id : "+id); //101
S.o.pln("Non static var age : "+age); //25
}
}
```



O/P

Static var id : 101
From mala.
Static var id 101.
Compile time error
From sheela
Static var id. 101.
Non. static var age: 25

this keyword

* this is a keyword
* this keyword is used to access the current object members that is whenever we have local variable and object variable with the same name we can access object variable using this keyword
* this keyword can be used only in non-static blocks

Example

```
class NSM2
{
  int age = 25;
  public void sheela()
  {
    int age = 28;
    S.o.pln("local var age: " + age); //28
    S.o.pln("this :: " + this), // NSM2@100
    S.o.pln("Non-static var age: " + this.age), //25
  }
  public static void main(String[] args)
  {
    NSM2 obj1 = new NSM2();
    NSM2 obj2 = new NSM2();
    S.o.pln("obj1 :: " + obj1); //NSM2@100
    S.o.pln("obj2 :: " + obj2), // NSM2@200
    obj1.sheela();
  }
}
```

1) WAJPT create blueprint of an employee with states and behaviours and test the blueprint using test class by creating 2 to 3 objects.

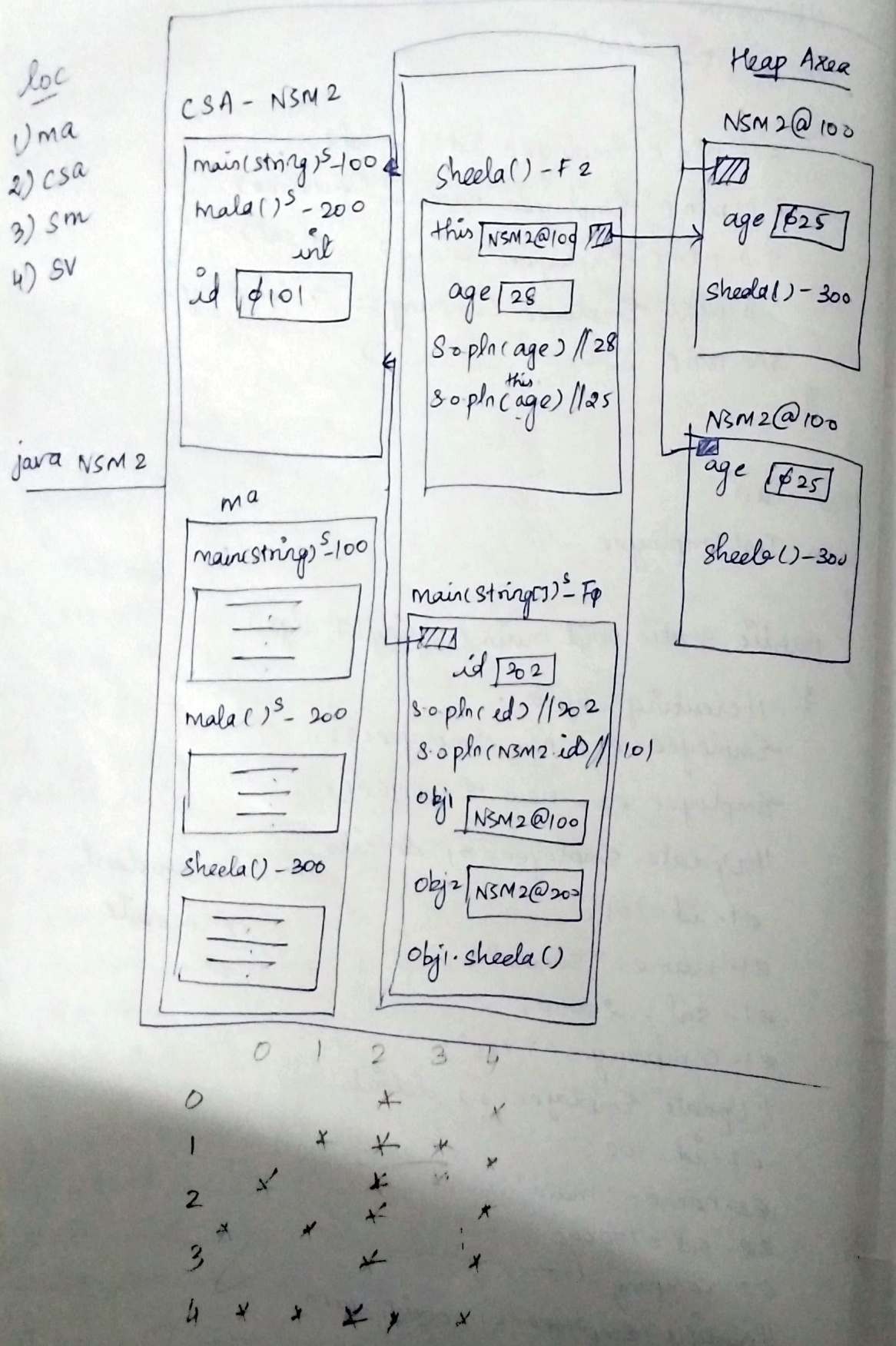
```
// blueprint class
class Employee
{ //states
  int id;
  String name;
```

```
double sal;
Static String company;

//Behaviours
public void info()
{
   S.o.pln("Employee Id::"+id);
   S.o.pln("Employee name::"+name);
   S.o.pln("Employee salary::"+sal);
   S.o.pln("Employee Company::"+company);
   S.o.pln("-----------");
}
}
//Test class
class TestEmployee
{
public static void main(String[] args)
{ //creating objects
Employee e1 = new Employee();
Employee e2 = new Employee();
// Update Employee-01 details,
e1.id=101;
e1.name="sheela";
e1.sal=25000;
e1.company="TYSS";
// Update Employee-02 details
e2.id=102;
e2.name="mala";
e2.sal=15000;
e2.company="TYSS";
//modify Employee-01 object data.
e1.sal=35000;
Employee.company="DYSS";
//Printing Employee-01 object details
e1.info();
//Printing Employee-02 object details
e2.info();
}}
```

① student
② Mobile

This Keyword tracing



11/9/23

Initializers:

Static Initializer	Non-static Initializer
* An anonymous block Prefixed with static keyword is known as static initializer (or) static initialization block.	* An anonymous block which is not prefixed with static keyword is known as non-static initializer (or) non Instance Initialization block. (declared in class area).
* static block is executed during loading of the class.	* Non-static block is executed for each and every object created.

Example:

```
class Initializers
{
  static
  {
     s.o.pln("class loaded");
  }
  {
     s.o.pln("object created");
  }
  public static void main(String[] args)
  {
     s.o.pln("From main");
     new Initializers();
     new Initializers();
     s.o.pln("Main End");
  }
}
```

O/P:

```
class loaded
From main
Object created
Object created
Main end.
```

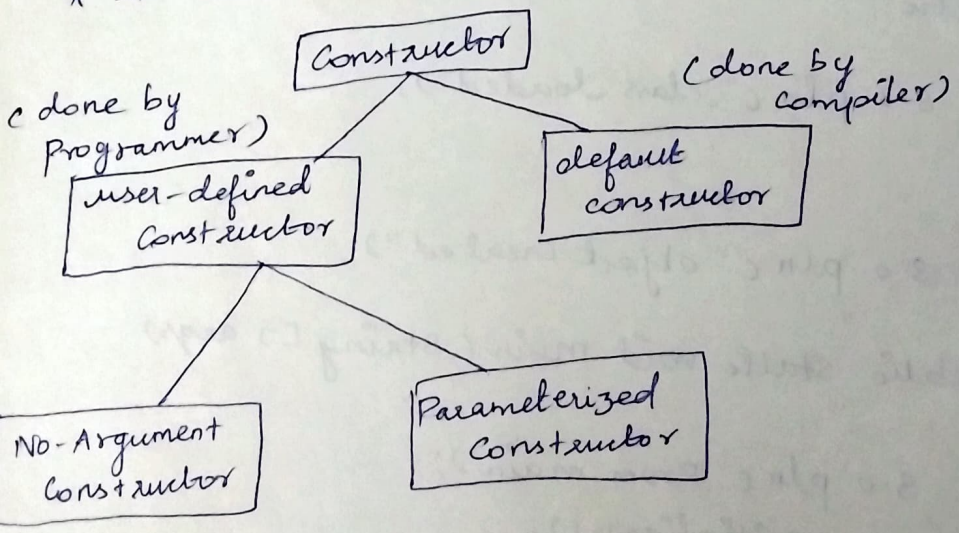
Constructor

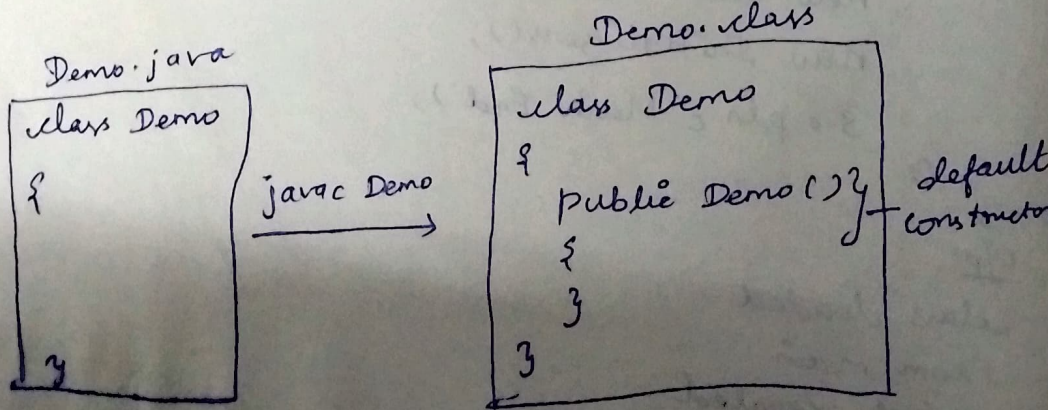
* Constructor is a non-static block which is used to load and initialize object members
* Constructor name must be classname
* Constructor doesn't have returntype
* We cannot call constructor explicitly.

Syntax

```
[modifier] classname ([Args])
{
  // load Non-static variable
  // load Non-static method
  // load Non-static Initializer (IIB)
  // user written instructions
}
```

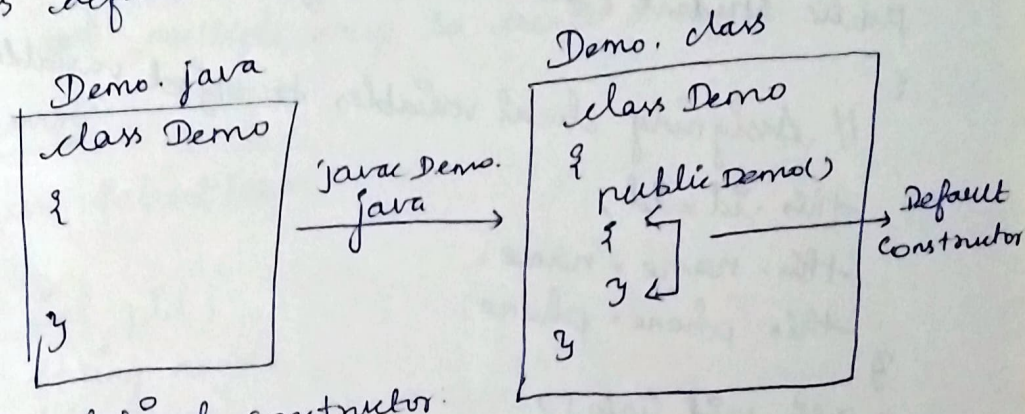
* In Java, constructor is classified as below





Default constructor

* In Java, every class that are created must have one constructor. If user doesn't define any constructor, the compiler will add the no-argument constructor which is known as default constructor.



User-defined constructor

* Constructor which are defined by the programmer are known as user-defined constructor.

* User-defined constructor is classified into two types.

i) No-Argument constructor.
ii) Parameterized Constructor

* Whenever we want to initialize states of an object after object creation we go for no-argument Constructor.

* Whenever we want to initialize states of an object during object creation we go for Parameterized constructor.

* used to load and initialize non-static members.

Example:

```
// Blueprint
class student
{
    int id;
```

```
String name;
long phone;
//No-Argument Constructor
public Student() { }
// Parameterized constructor
public student (int id, String name, long phon
{
    // Assigning local variables to object variables
    this.id = id;
    this.name = name;
    this.phone = phone;
}
public void info()
{
    S.o.pln("Student id :: " + id);
    S.o.pln("Student name :: " + name);
    S.o.pln("Student phone :: " + phone);
    S.o.pln("-----------");
}
}
```

Student Testclass

```
class Test student
{
  public static void main(String[] args)
  {
    // using No-Argument Constructor
    Student s1 = new Student();
    s1.id = 102;
    s1.name = "mala";
    s1.phone = 88992;
    // parameterized constructor using
    Student s2 = new Student(101, "sheela", 55442);
    s1.info();
    s2.info();
  }
}
```

12/9/23

Constructor overloading

* Designing multiple constructors with different arguments is known as constructor overloading
* The advantage of constructor overloading is we get multiple ways to create a object

Example

```
class PatientForm
{
    int pid;
    String name;
    int age;
    public PatientForm(){ }
    public PatientForm(int pid)
    {
      this.pid = pid;
    }
    public PatientForm(int pid, String name)
    {
      this.pid = pid;
      this.name = name;
    }
    public PatientForm(int pid, String name, int age)
    {
      this.pid = pid;
      this.name = name;
      this.age = age;
    }
}
class TestPatientForm
{
  public static void main(String[] args)
  {
    PatientForm p1 = new PatientForm();
    PatientForm p2 = new PatientForm(101);
```

```
PatientForm p3 = new PatientForm(102, "Allen");
PatientForm p4 = new PatientForm(103, "Smith");
p1.info();
p2.info();
p3.info();
p4.info();
}
}
```

① Non-static method block

this ✓
super ✓
this() x
super() x

① Static method block

this x
super x
this() x
super() x

② Instance Initialization block

this ✓
super ✓
this() x
super() x

② ~~Non~~-Static method Initialization block.

this x
super x
this() x
super() x

③ Constructor

this ✓
super ✓
this() ✓
super() ✓

Keyword

this keyword (vs) super this statement

* used to access object members
* used with non-static blocks

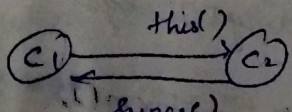
* this() used to call one Constructor from another Constructor.
* used only inside Constructor

Keyword

this (or) super

Statement

this (or) super.



13/9/23

Constructor chaining:

* One constructor calling another constructor is known as Constructor chaining
* We can achieve constructor chaining using Constructor call statements they are
  i) this ([args])
  ii) super ([args])
* Constructor chaining is used to achieve code reusability

this()

* this() is a Constructor call statement
* this() is used to call the constructor of same class
* this() must be the first statement inside Constructor
* this() can be used only inside Constructor
* this() should not invoke constructor recursion

Example

```
class student
{
  int sid;
  String name;
  int phone;
  public student(){ }
  public student (int sid)
  {
    this();
    this.sid = sid;
  }
  public student (int sid, String name)
  {
    this(sid);
    this.name = name;
  }
  public student (int sid, String name, int phone)
  {
    this.sid=sid;
    this.phone=phone; } }
```

19/9/23

Encapsulation

* Encapsulation is a process of combining states and behaviours of an object in a class.
* Encapsulation used to achieve data security.
* We can achieve data security using data hiding
* Data hiding is a process of restricting direct access to the data members from outside the class and providing controlled access

Step 1: Make the data members private
Step 2: Provide getter and setter methods

* Private members cannot be accessed from outside the class
* Getter method is used to read private data member from outside the class.
* Setter method is used to modify the private member from outside the class
* The best example for encapsulation is Java bean class.
* Real time example for encapsulation:
  i) ATM Machine
  ii) Bag
  iii) Bank Account
  iv) Automated Snacks Vending machine.

Java Bean.

* Java Bean is an encapsulated class which is used to achieve data security

Java Bean rules:

i) All the variables must be private.
ii) class must have getter and setter methods

iii) class must have zero parameterized constructor

Example

```
class student
{
  private int sid;
  private string name;
  private int phone; public student(){}.
  // getter methods
  public int getsid()
  {
    return sid;
  }
  public string getName()
  {
    return name;
  }
  public int getPhone()
  {
    return phone;
  }
  // Setter methods
  public void setsid(int sid)
  {
    this.sid = sid;
  }
  public void setName(string name)
  {
    this.name = name;
  }
  public void setPhone(int phone)
  {
    this.phone = phone;
  }
}
```

```
class TestStudent
{
  public static void main (String[] args)
  {
    Student s1 = new Student();

    // update data
    s1.setSid(101);
    s1.setName("sheela");
    s1.setPhone("5566");

    // Printing data
    S.o.pln("student id::" + s1.getSid()); //101
    S.o.pln("Student name::" + s1.getName()); //sheela
    S.o.pln("Student phone::" + s1.getPhone()); //5566
  }
}
```

15/9/23

Relationship

* Relationship is a Connection between two objects
* In Java, relationships are classified into 2 types they are
  (i) Is-a relationship
  (ii) Has-a relationship

Has-A relationship:

* Has-A relationship is one object having another object
* An Example
  (i) Car has a Engine
  (ii) Mobile has a sim
  (iii) Library has a book.

* In Java, we can achieve has-A relationship using association

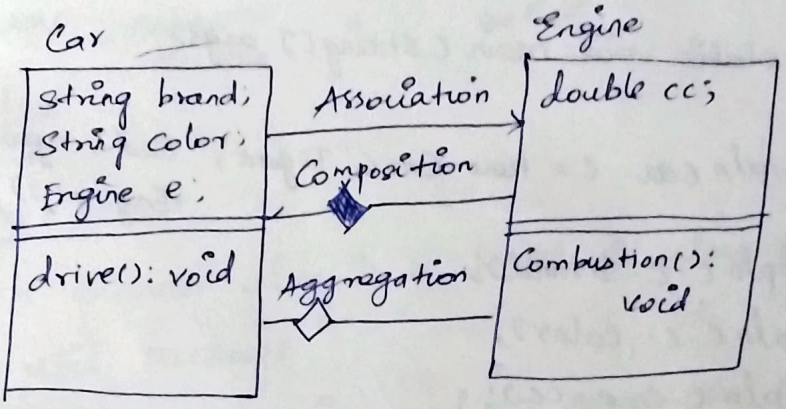
Association

* Association is a design technique to achieve Has-A relationship by creating reference variable of one class inside another class

* In Java, we have 2 types of association
  (i) Composition (Early Instantiation)
  (ii) Aggregation (lazy Instantiation)

Example

class diagram



Program

Instantiation - object creation

```
class Car
{
  String brand;
  String color;
  Engine e;
  Public Car(String brand, String color, Engine e)
  {
    this.brand = brand;
    this.color = color;
    this.e = e;
  }
  public void drive()
  {
    S.o.println("long drive");
  }
}
```

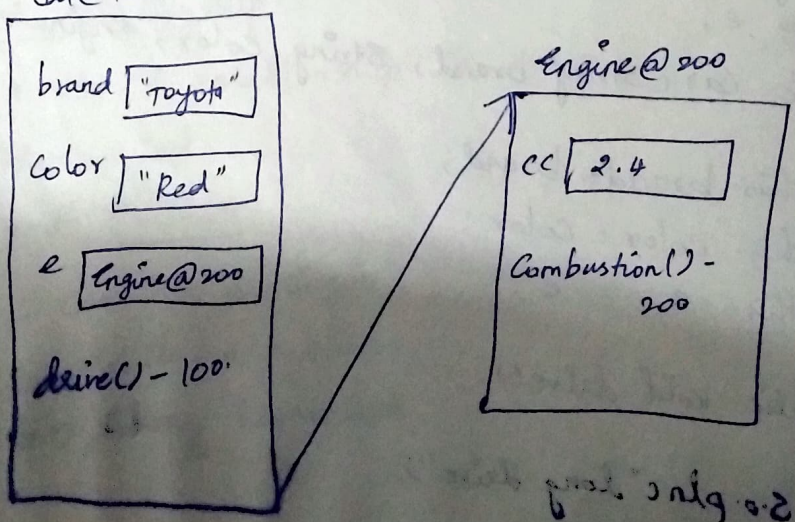
```
class Engine
{
  double cc;
  public Engine(double cc)
  {
    this.cc = cc;
  }
  public void combustion()
  {
    S.o.pln("Power is generated");
  }
}
class TestCar
{
  public static void main(String[] args)
  {
    S.o.pln(Car c = new Car("Toyota", "Red", new
                                    Engine(2.4));
    S.o.pln(c.brand);
    S.o.pln(c.color);
    S.o.pln(c.e.cc);
    S.o.pln(c.drive());
    S.o.pln(c.e.combustion());
  }
}
```

Tracing

Car c

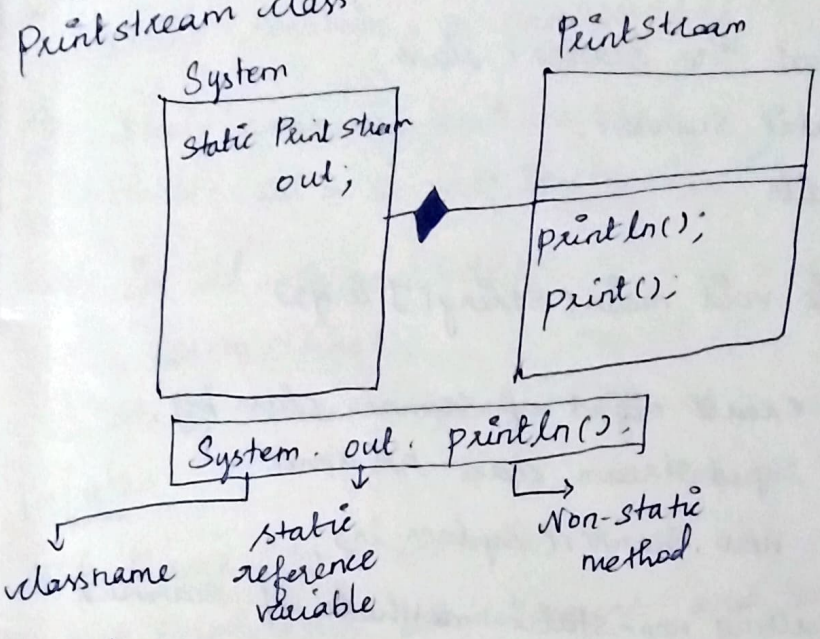
Car@100

new Car("Toyota", "Red", new Engine()) — Engine@200



1) What is System.out.println()?

System.out.println() is a composition where System class is having static reference variable of PrintStream class



12/9/23

Scanner class

* Scanner class is a built-in class present in java.util package
* Scanner class is used to read the data from the console dynamically.
* Scanner class provides following non-static methods to read the data from the console:

Datatype	Non-static method
byte	nextByte()
short	nextShort()
int	nextInt()
long	nextLong()
float	nextFloat()
double	nextDouble()
boolean	nextBoolean()
string	next()
	nextLine()
char	next().charAt(0)

Steps to read data from the console

step 1 : Import Scanner class.

Program

```
// Step 01 : Import the scanner class
import java.util.Scanner;
class UserDetails
{
  public static void main(String[] args)
  {
    // step 02 : create object of Scanner class by
        passing InputStream class reference
    Scanner sc = new Scanner(System.in);
    // Step 03 : calling non-static methods of scanner
        class
    S.o.pln("Enter your name");
    String name = sc.nextLine();
    S.o.pln("hi, " + name + " Enter your age");
    byte age = sc.nextByte();
    S.o.pln("hi," + name + "Enter your salary");
    double sal = sc.nextDouble();
    S.o.pln("hi," + name + " Enter your gender"),
    char gender = sc.next().charAt(0);
    S.o.pln(" Name:: " + name);
    S.o.pln(" Age :: " + age).
    S.o.pln(" Salary :: " + sal);
    S.o.pln(" Gender :: " + gender);
    // step 04 : closing connection
    sc.close();
  }
}
```

① int id = sc.nextInt();
long phone = sc.nextLong(); ✓

② String name = sc.nextLine();
String location = sc.nextLine(); ✓

③ String name = sc.nextLine();
double sal = sc.nextDouble(); ✓

④ int id = sc.nextInt();
sc.nextLine();
String name = sc.nextLine();

19/9/23

Inheritance (Is-A relationship)

* Relationship between parent and child is known as Is-a relationship.

* Example: Bike is a vehicle.
Dog is a animal.
Mango is fruit

* In Java, we can achieve is relationship using inheritance.

Inheritance

* Inheritance is a design technique to achieve is-a relationship where one class will acquire states and behaviours of another class.

* The class which acquires states and behaviours of another class is known as child class.

* the class which gives its states and behaviours to another class is known as parent class.

* Inheritance is used to achieve code reusability.

* In Java, we can achieve inheritance using two keywords. They are
  (i) Extends (1).
  (ii) Implements (i)

Extends

* Extend is a keyword.
* Extends is used to achieve inheritance between classes or interfaces
* Extends must be used with subclass

Syntax

```
class Parent
{
}
class child extends Parent
{
}
```

Inheritance with respect to static members:

* All the static members are inherited.
* Using child class conference, we can access both child class members and parent class members.
* Using parent class reference, we cannot access child class members

Example

Program

```
class sheela
{
    static int id=01;
```

```
}
class Mala extends sheela
{ static int age=25;
  public static void main(String [] args)
  {
    //Using child class reference
    S.o.pln("Mala class var Mala.age :: "+Mala.age);
                                          // 25.
    S.o.pln("sheela class var Mala.id :: "+Mala.id);

    //Using parent class reference
    S.o.pln("sheela class var sheela.id :: "+sheela.id);
    S.o.pln("sheela class var sheela.age:
            Mala                "sheela.ago); //CE
  }
}
```

20/9/23

Note:

+ In Java, every class must have a Parent class. If user is not extended the class to any other class, Compiler will extends to object class.

Demo.java

```
class Demo
{


}
```

javac Demo.java →

Demo.class

```
class Demo extends
{       Object
  public Demo()
  {
  }
}
```

Inheritance with respect to non-static members

* All the non-static members are inherited.
* Using child class object reference, we can access both child class members as well as parent class members.
* Using parent class object reference, we can access only parent class members.

Program

```
class Super
{
  int id = 101;
  public Super()
  {
  }
}
class Sub extends Super
{
  int age = 25;
  public Sub()
  {
  }
}
class Test
{
  public static void main(String[] args)
  {
    // Accessing Using child class object reference
    Sub child = new Sub();
    S.o.pln("Sub class var child.age :: "+child.age);
                                        //25
    S.o.pln("Super class var child.id :: "+child.id);
                                        //101
    // Accessing Using Parent class object reference
    Super parent = new Super();         //OK
    S.o.pln("super class var parent.id :: "+parent.id);
    S.o.pln("Sub class var parent.age :: "+parent.age);
                                        //CTE
  }
}
```

Note

* whenever we create child class object, the object will have
  * Instance of object class
  * Instance of parent class
  * Instance of child class
* The above is achieved because of super() statement

Super():

* Super() is a constructor call statement
* Super() is used to call parent class constructor
* Super() must be the first statement inside constructor
* Super() is a default statement that is if user doesn't define this statement, the compiler will add super().

child → Sub @ 100
new Sub();

```
Sub()
{
// super()
// load NSV & NSM
// load NSI
// UWI
}
```

```
Super()
{
// Super()
// load NSV & NSM
// NSI
// UWI
}
```

```
object()
{
// super()
// load NSV & NSM
// NSI
// UWI
}
```

Sub @ 100.

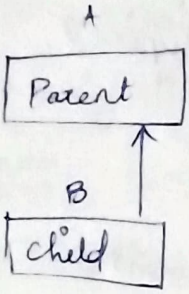
Instance of object class	Object class // Non-static methods
Instance of Parent class	Super class id 101
Instance of child class	Sub class age 25

```
Constructor()
{
// Constructor call Stmt → this() - user defined stmt
                        → super() - default stmt
// load NSV & NSM
// NSI
// UWI
}.
```

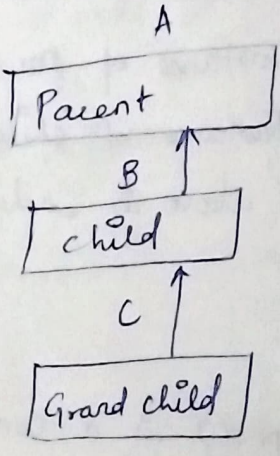
Types of inheritance

* In Java, we have 5 types of inheritance

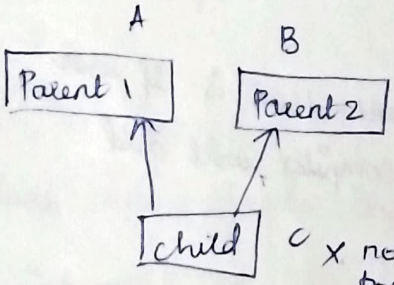
① Single level inheritance



② Multi-level inheritance

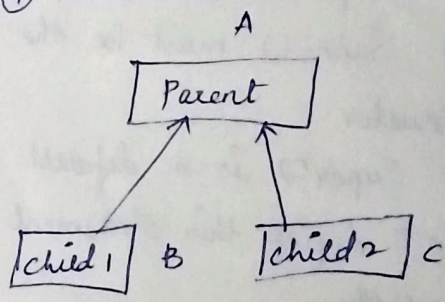


③ Multiple inheritance

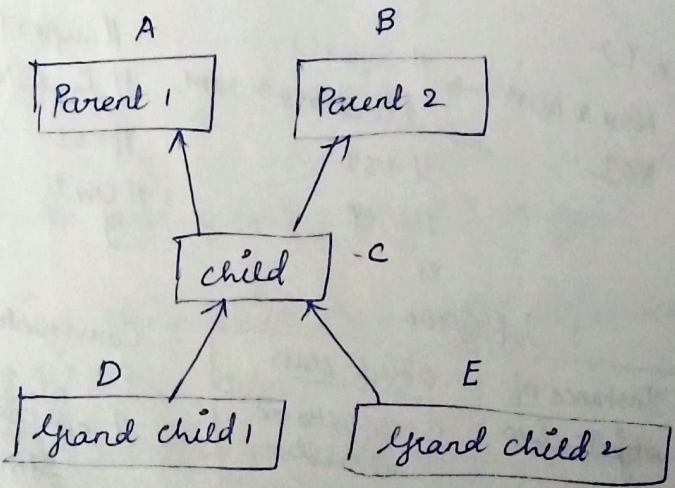


x not possible in java using classes

④ Hierarchical inheritance



⑤ Hybrid inheritance



Note

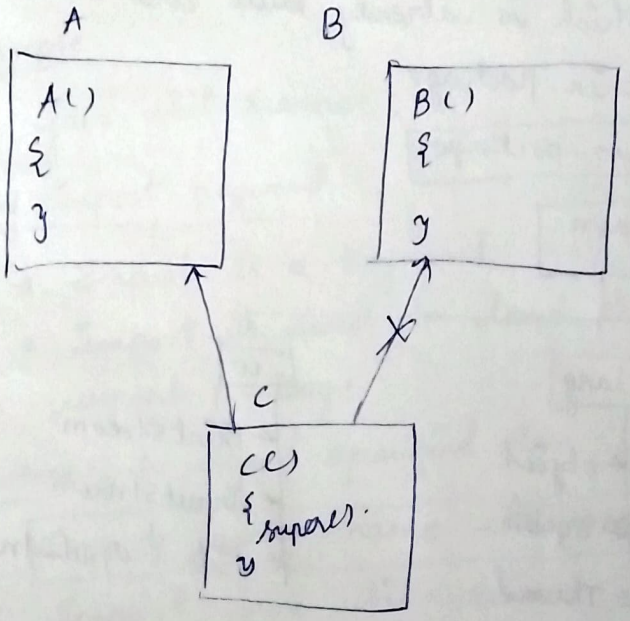
* In Java, multiple inheritance is not supported in classes because of

i) Syntax not supported

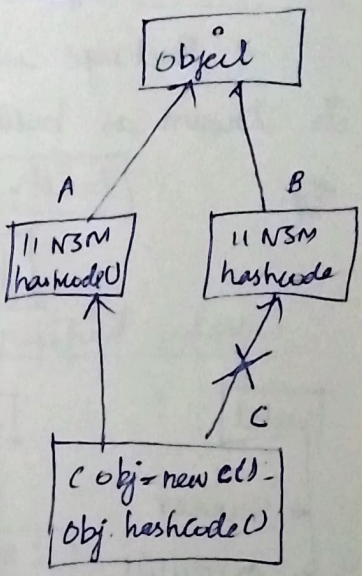
ii) Ambiguity of diamond triangle

Ambiguity of diamond triangle

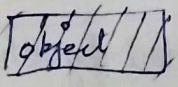
Case 1: *

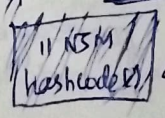


Case 2



Case 1:





* Let us consider class C extends both class A and class B, when C class object created. super() of C class will be in confusion to call the constructor of class A or to call the constructor of class B. This situation is known as ambiguity of diamond problem.

Case 2: * Let us consider class C extends both class A and class B

* The C class will be in a confusion to acquire the properties of object class from class A or class B. This situation is known as ambiguity of diamond problem.

21/9/23

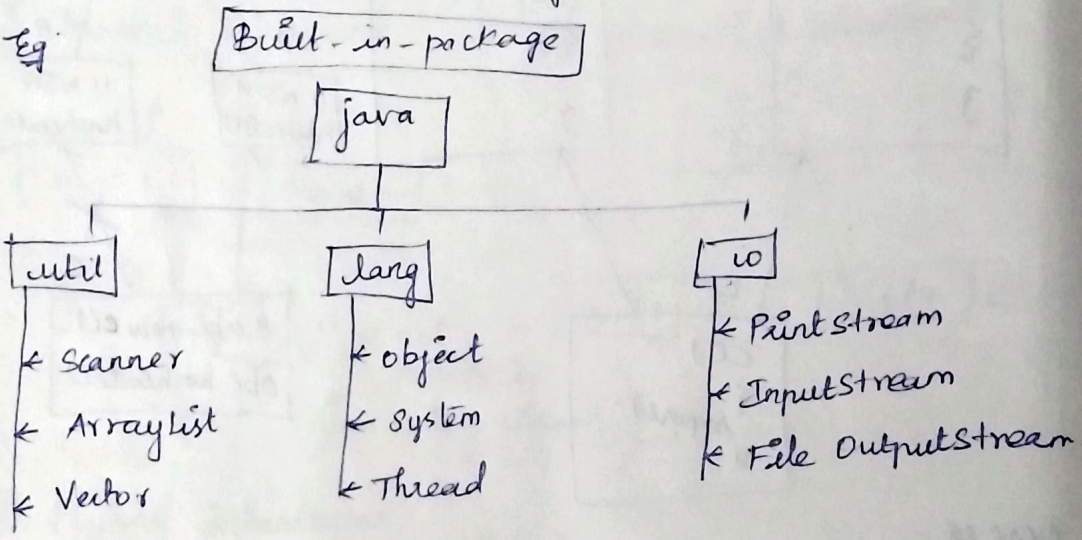
Package

* Package is a mechanism to store similar set of java file
* In java, package is classified into two types
  i) Built-in package
  ii) User-defined package

Built-in package

* Package which is already built and stored is known as built-in package

Eg



User-defined package

* Package which is created by the programmer is known as user-defined package
* User can create package using package keyword

Syntax

```
package packagename;
```

Note

* Convention for packagename is package name must be in lowercase and must follow below syntax:

com.companyname.projectname.module;

* We can use member of one package inside another package only in two ways
  (i) Fully qualified name
  (ii) Using import keyword

Using fully qualified name

* Fully qualified name is the path address of a member from the source folder

Example

java.util.Scanner;

Using import keyword

* Import is a keyword
* import is used to load specified class into current package

Syntax - to import specified class

import packagename.classname;

Eg: import java.util.Scanner;

Syntax - to import all the class of the package

import packagename.*;

Eg: import java.util.*;

* java.lang package is imported to the current package by default

Example program

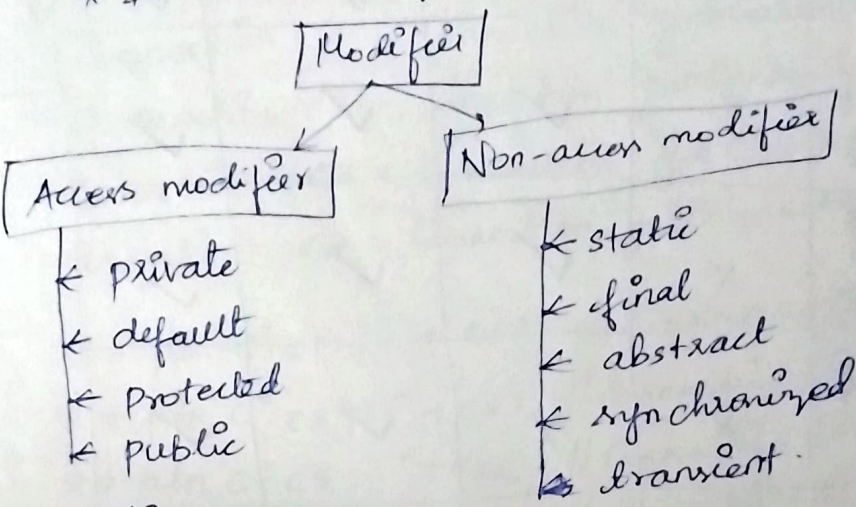
```
package com.ty.ai.modifiers;
import java.util.*;
import com.ty.ai.inheritance.Mala;
public class Demo
{
  public static void main(String[] args
  {
```

```
// Using fully qualified name
com.ty.a1.modifiers.sheela obj1 =
        inheritance
    new com.ty.a1.inheritance.sheela();
// Using import keyword
Mala obj2 = new Mala();
Vector v = new Vector();
ArrayList a1 = new ArrayList();
Thread t = new Thread();
  }
}
```

22/9/23

Modifiers

* Modifiers are keywords which are used to modify the behaviour of java members
* In Java, modifiers are classified into 2 types



Access modifiers

* Access modifiers are used to control the accessibility (or) visibility (or) scope of the java members
* Access modifiers can be prefixed with
  (i) Variable
  (ii) methods
  (iii) constructor
  (iv) class

Note

* class cannot be prefixed with private (or) protected modifiers
* Initializers cannot be prefixed with access modifiers

Visibility of java members with respect to access modifiers

	private	default	protected	public
same class	✓	✓	✓	✓
diff class of same pack	X	✓	✓	✓
sub class of same pack	X	✓	✓	✓
sub class of diff pack	X	X	✓	✓
diff class of diff pack	X	X	X	✓

Singleton:

* Singleton is a design pattern to make sure that a class can have only one object.

* Rules of singleton:

-> Constructor must be ~~protected~~ private.
-> class must have private static reference variable.
-> class must have public static getter method.

Program:

```
class Connection
{
    private static Connection connection;
    private Connection(){ }
    public static Connection getConnection()
    {
        if (Connection == null)
        {
            Connection = new Connection();
        }
```

```
    return Connection;
  }
}
public class Test {
  public static void main (String[] args)
  {
    Connection c1 = Connection.getConnection();
    Connection c2 = Connection.getConnection();
    Connection c3 = Connection.getConnection();
    Connection c4 = Connection.getConnection();

    S.o.pln("c1:: "+c1); // Connection@100
    S.o.pln("c2:: "+c2); // Connection@100
    S.o.pln("c3:: "+c3); // Connection@100
    S.o.pln("c4:: "+c4); // Connection@100
  }
```

Class casting (or) Non-primitive Typecasting

* The process of converting one class reference into another class reference is known as Non-primitive type casting (or) class casting

* In Java, class casting can be done only between parent and child classes.

* class casting is classified into two types:

(i) Upcasting

(ii) Downcasting

Upcasting

* Upcasting is the process of converting child reference into parent reference.

* Upcasting is done by the compiler implicitly.

* The advantage of upcasting is we can generalize store any subclass address reference variable

Program

```
class Cab
{
}
class Auto extends Cab
{
}
class Mini extends Cab
{
}
class luxury extends Cab
{
}
```

```
class Test
{
  public static void main(String[] args)
  {
    // Upcasting ==> Converting child reference into
       Parent reference

    Cab x = new Auto();
    S.o.pln(x); // Auto@100

    x = new Mini();
    S.o.pln(x); // Mini@100

    x = new luxury();
    S.o.pln(x); // luxury@100
  }
}
```

Now Note Disadvantage of upcasting is we cannot access child class properties (or) members using Parent class reference

* To overcome this advantage, we go for downcasting

Downcasting

* Downcasting is the process of converting Parent reference into child reference.
* Downcasting is not done by the compiler implicitly. Programmer has to do it explicitly using typecast operator.

Program

```
class Cab
{
}
class Auto extends Cab
{
  int price = 10;
}
```

```
class Mini extends Cab
{
  int price = 15;
}
class Luxury extends Cab
{
  int price = 20;
}
class Test
{
  P S V m(S[] args)
  {
    // Upcasting ==> Converting child reference
    to parent reference
    Cab x = new Auto();
    S.o.pln(x.price); //CTE
    // Downcasting ==> Converting parent reference
    into child reference
    Auto y = (Auto)x;
    S.o.pln(y.price); //10
  }
}
```

ClassCastException

* ClassCastException is a runtime problem we get during classCasting that is if the classes does not have Is-A relationship and we tried to cast them we get ClassCastException

* We can avoid ClassCastException by checking the instance of the class in the object using instanceof Keyword

instanceof

* instanceof is a keyword
* instanceof is used to check instance of specified class in the object

Syntax

Objectref instanceof classname

* instanceof operator returns true if the specified class present in the object else returns false

Program

```
class Test
{
  public static void main(String[] args)
  {
    // Upcasting ==> Converting child reference into Parent reference
    Cab cab = new Mini();
    // Downcasting ==> Converting Parent reference into child reference
    // Without Using instanceof operator
    Auto y = (Auto) cab; // class cast Exception
    // Using instanceof operator
    if (cab instanceof Auto)
    {
      Auto auto = (Auto) cab;
      S.o.pln(auto.price); //10
    }
    else if (cab instanceof Mini)
    {
      Mini mini = (Mini) cab;
      S.o.pln(mini.price); //15
    }
    else if (cab instanceof Luxury)
    {
      Luxury luxury = (Luxury) cab;
      S.o.pln(luxury.price); //20
    }
  }
```

27/9/23

Polymorphism

* Polymorphism is a process of same member showing multiple behaviours based on situation is known as polymorphism
* In Java, we have 2 types of polymorphism

They are:

i) Compile time polymorphism.
ii) Runtime polymorphism

Compile time polymorphism:

* which implementation of a member has to be executed is decided by the compiler during compilation based on arguments. This process is known as compile time polymorphism
* Compile time polymorphism is also known as static binding (or) early binding

* Examples
  (i) Method overloading
  (ii) Constructor overloading
  (iii) Method hiding / shadowing.

Runtime polymorphism

* which implementation of a member has to be executed is decided by the JVM during runtime based on objects. This process is known as runtime polymorphism.
* Runtime polymorphism is also known as dynamic binding (or) late binding

* Examples
  (i) Method overriding

Method overriding

* Method overriding is the process of inheriting parent class method and changing its implementation according to its child class requirements.

* Method Overriding rules

(i) There must be Is-A relationship between classes

(ii) Methods must be non-static

(iii) Signature must be same

(iv) Return type must be same

(v) Modifier of subclass methods must have equal (or) higher visibility than super class method modifiers

Note

* We cannot override private methods, static methods and final methods.

program

```
class Allen
{ public void party() { S.o.pln("Home"); }
}
class Smith extends Allen
{ public void party() { S.o.pln("Beach"); }
}
public class Test
{ public static void main(S[] args)
  {
    Allen obj = new Allen();
    obj.party(); // Home
    obj = new Smith();
} } obj.party(); // Beach
```

28/09/23

final

* final is a keyword
* In java, we can prefix final keyword with
  (i) variables -> To create constants
  (ii) Method -> To prevent method overriding
  (iii) class -> To prevent inheritance.

final variable

* Variable which is prefixed with final keyword is known as final variable.
* final variable is also known as constant.
* final variable cannot be reinitialized.

Note

* Global final variable must be initialized in the class block only. that is using
  (i) Declare and initialization statement
  (ii) Initialization block.

Note

Non static variable can be initialized in 3 ways:
  (i) Declare and Initialization statement
  (ii) Initialization block.
  (iii) Constructor.

* Convention of a final variable is ~~Snakecase~~. => SNAKE_CASE.

Program:

```
public class FinalVariable
{
    final static int PHONE = 5555;
    final static String NAME;
    static
    {
        NAME = "Sheela";
    }
    public static void main(String[] args)
    {
        NAME = "Mala"; //CTE
        PHONE = "6666; //CTE
        final int ID = 101;
        final int AGE;
        AGE = 25;
        ID = 102; //CTE
        AGE = 35; //CTE
    }
}
```

final method

* Method which is prefixed with final keyword is known as final method.
* final method cannot be overrided.

Example

```
public class Mala
{
    public final void eat()
    {
        S.o.println("Sambar Sadhan");
    }
}
```

```
public class sheela extends Mala
{
  @Override
  public void eat() //CTE
  {
    S.o.println("Briyani");
  }
}
```

final class

* class which is prefixed with final keyword is known as final class.
* final class cannot be inherited

Program

```
public final class Mala
{
}
class Sheela extends Mala //CTE
{
}
```

Note

* final class is also known as immutable class

Immutable class

* Immutable class is a final class that is once an object created, we cannot change or modify the content of object.

Rules of Immutable class

(i) class must be final
(ii) Data members must be private and final

(iii) Data members must have getter methods

(iv) Data members must be initialized through Constructor

Program

```
final class Immutable
{
  private final int ID;
  private final String NAME;
  public Immutable (int id, String name)
  {
    this.ID = id;
    this.NAME = name;
  }
  public int getID()
  {
    return ID;
  }
  public String getName()
  {
    return NAME;
  }
}
public class TestImmutable
{
  public static void main (String[] args)
  {
    Immutable obj = new Immutable(101, "Sheeda");
    S.o.pln(obj.getId());
    S.o.pln(obj.getName());
  }
}
```

Examples of Immutable classes

* Math classes
* string classes
* Random classes
* Wrapper classes

29/9/23

Abstraction

* Abstraction is the process of hiding unnecessary details and showing only necessary details
* In java, hiding implementation and providing only declaration of the method is known as abstraction
* In java, we can achieve abstraction using
  (i) Abstract class
  (ii) Interface
* Whenever we want to achieva partial abstraction (0-100%.) we go for abstract class
* Whenever we want to achieve 100%. abstraction, we go for interface
* Advantage of abstraction
  (i) Data security
  (ii) Easy code maintenance.
  (iii) loose coupling.
* Real time examples of abstraction,
  (i) ATM machine
  (ii) laptop.
  (iii) Coffee machine.
  (iv) Car, etc..,

Abstract class.

* class which is prefixed with abstract keyword is known as abstract class
* Abstract class is used to achieve partial abstraction.
* We cannot create object for abstract class.

* We can have constructor in the abstract class.

* In abstract class, we can have both concrete method and abstract method.

* We can provide implementation to the abstract method using implementing class by method overriding.

Program

```
// Abstract class
abstract class Addition
{
  // Abstract method
  public abstract void add(int a, int b);
  // Concrete method
  public void sub(int a, int b)
  {
    System.out.println("From addition :: "+(a-b));
  }
} // Implementing class
class Manga extends Addition
{
 @Override
 public void add(int a, int b)
 {
   S.o.pln("From Manga :: "+(a-b));
 }
}
public class Test
{
  public static void main(String[] args)
  {
    Addition add = new Manga();
    add.add(45, 25); // 20
    add.sub(50, 30); // 20
  }
}
```

Interface

* Interface is a blueprint of class.
* In java, we can create interface using interface keyword

Syntax

```
interface InterfaceName
{
  // stmt.
}
```

* Interface is used to achieve 100% abstraction.
* In interface, we can have only
  (i) public static final variable.
  (ii) Non-static abstract method
  (iii) Static method
  (iv) Default method
  (v) Private method.
* In interface, we cannot have
  (i) Constructor.
  (ii) Initializer.
* We cannot create object for interface.

Program

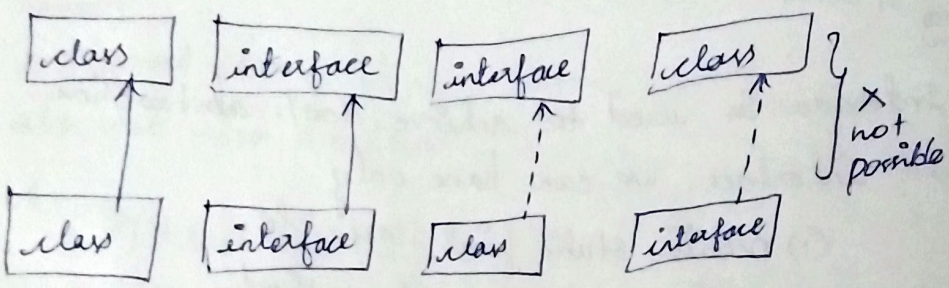
```
// Interface
interface Addition
{ int id = 101; // public static final variable
  public void add(int a, int b); // Abstract method
} // Implementing class
class Manga implements Addition
{ @Override
  public void add(int a, int b)
  { s.o.pln("from Manga:: "+(a+b)); }
};
public class test
{ p s v m(s[] args)
  { Addition add = nao Manga(); add.add(45, 25); // to }}
```

Inheritance with respect to interface

* Whenever child and parent templates are same, we go for extends
* Whenever parent and child templates are different, we go for implements

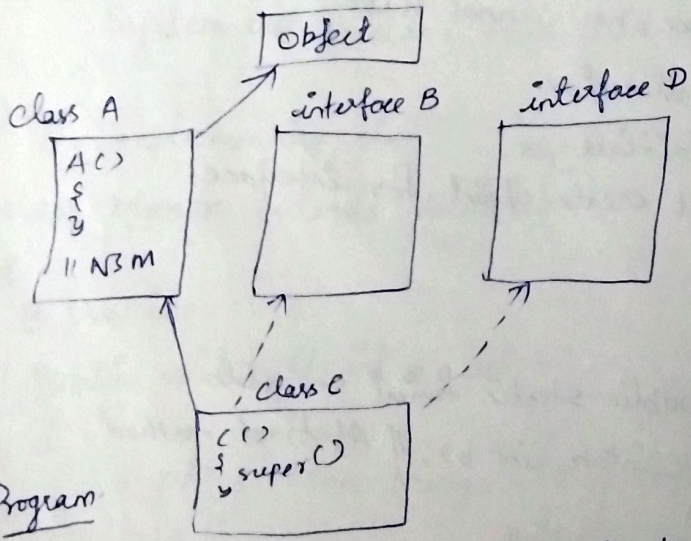
Note

* Interface cannot have class as a parent.



Multiple inheritance using interfaces

* In java, we can achieve multiple inheritance using interface.



Program

```
class A
{
}
interface B
{
}
interface D
{
}
```

```
public class C extends A
        implements B,D
{
}
```