

MY JAVA NOTES

FROM JSPIDER, HEBBAL

Prepared By :

SAGAR SUNAR

Contact No. : +91-7483666261

+977-9864453622(Whatsapp)

E-mail :- sagarsunar202@gmail.com

Address : Thirumenahalli, Yelanhanka, Bangalore-64.

*These notes are completely based on lectures given by respected sir, Mr. **Subham K.S.** , whose teachings always made new hope in learning programming concepts. Hence, it is only for my personal educational purpose and should not be used for any commercial printing or selling purpose.*

*-Thank you!
SAGAR*

Module 2

(Object Oriented Procedure)

- Object and Class
- Object Initialization
- Inheritance **

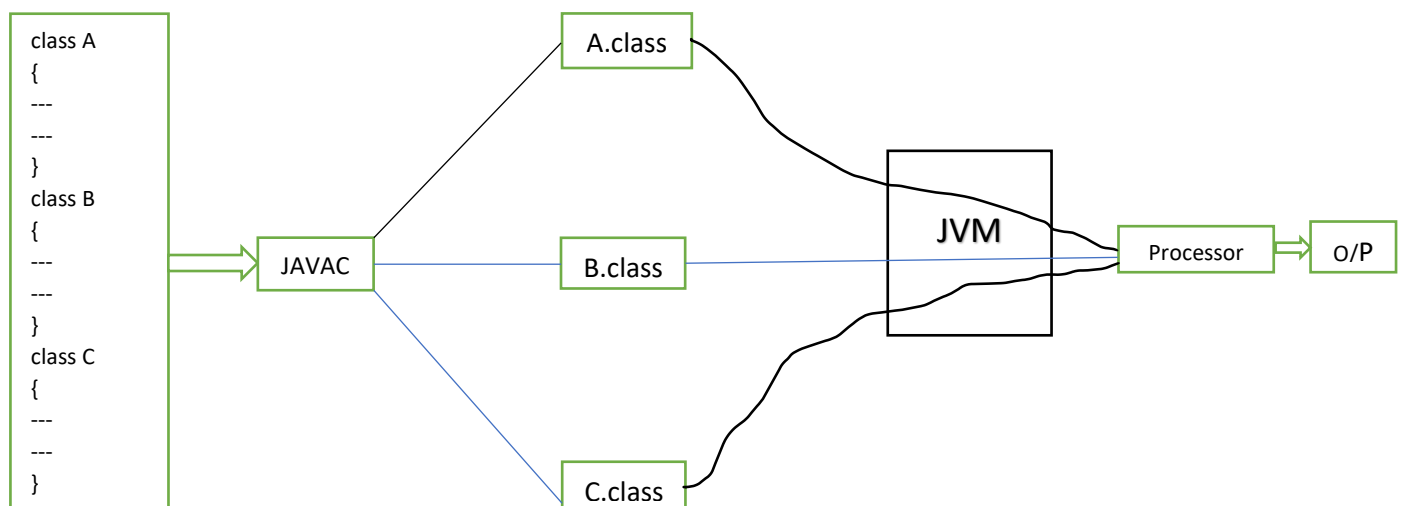
Method Overloading and Method Overriding

- Packages
- Abstract Class and Abstract Methods
- Interface
- Polymorphism**
- Encapsulation**
- Abstraction**
- JavaBean Class**
- Singleton Class
- Typecasting**

OBJECT AND CLASS

1.

In a java, a **source** file can have any number of classes in it. The number of bytecode file/class file/ executable file to be generated after successful compilation will be equal to the number of classes present in the source file. Out of all the bytecode file we can pass one at a time to the **JVM** for execution.



2.

In Java, even empty class will also be executed but it is of no use because when we pass that class to the JVM for the execution, we will throw run time error as shown below:

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

class B
{
    public static void m1()
    {
        System.out.println("Hello World! B");
    }
}

class C
{
}
```

OUTPUT :

R:\Program Practice\module2>java A

Hello World! A

R:\Program Practice\module2>java B

**Error: Main method not found in class B, please define the main method as:
public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application**

R:\Program Practice\module2>java C

**Error: Main method not found in class C, please define the main method as:
public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application**

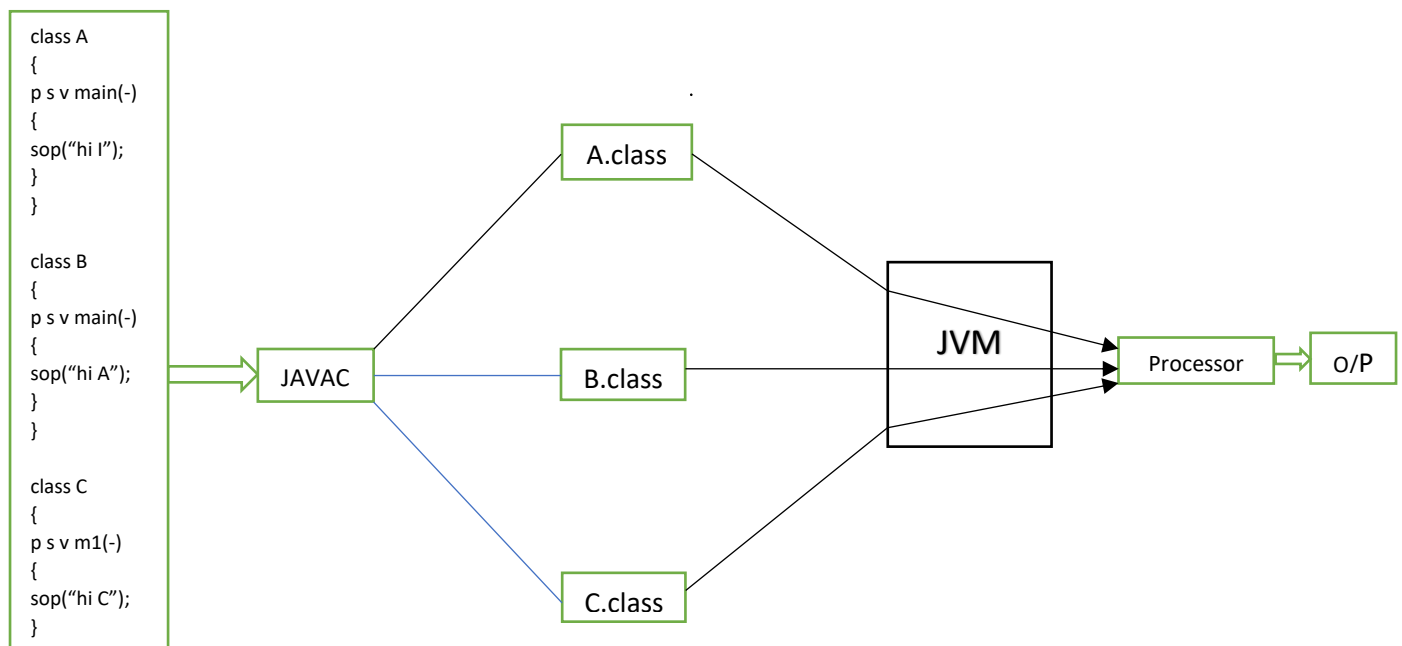
3.

The number of bytecode file or executable file or class file to be generated after successful compilation will be equal to the number of classes present in the source file. Out of all the executable file, we can pass only those to the **JVM** which contains main method in it. If at all, we pass a class file to the JVM which does not contains **main method** in it, then we will get run time error as shown below:

O/P1 : Hi A

O/P2 : Hi B

O/P3 : **RTE** , can't find main() in class C or can't find the method in below format:
public static void main(-)



→ Important Note :

1. In Java, it is possible to have multiple main method in a single source file provided if they are in different classes but as per the **IT Standards** , it is not recommended. A **source** file should have only one main method inside it.

2. Whenever a source file contains multiple classes in it, the source filename will be same as that of the class name which contains **main** method in it.


```

class A
{
    ----
    ----
}

class B
{
    p s v main(-)
    {
        ---
        ---
    }
}

```

B.java

```

class Sample
{
    ----
    ----
}

class Sample
{
    p s v main(-)
    {
        ---
        ---
    }
}
class Dummy
{
    ----
    ----
}

```

Sample.java

3. Explain the purpose of multiple classes in java?

According to the **IT standards**, the class which contains main method inside it should not have anything else; This is the reason in **JAVA**, every source file will have minimum two classes: One for the logic & another for the main method.

→ Class Templet :

<pre> class CN { Variable/DataMember / \ static non-static </pre>	<pre> class Demo { int x=10; static int y=20; public static void m1() { </pre>
---	--

```

Methods/memberFunction
├── Static
└── non static
    
```

```

}
{
}
}
    
```

```

int a =10;
-----
public void main m2()
-----
int b=20;
-----
    
```

Local Variable
Neither static nor non static

1. The members of a class which are declared using static keyword are known as **static members**.
2. The members of a class which are declared without using static keyword are known as **non-static members**.
3. Java does not support the concept of **global variable** rather it allows the following variable.
 - a. **Static variable / static data member**
 - b. **Non-static variable / non-static data member**
 - c. **Local variables**
4. The data members that are declared outside the method using static keyword are known as **static data member**.
5. The data members that are declared outside a method without static keyword are known as **non-static data member**.
6. The variables which are declared inside a method are known as **Local Variable**, they are neither **static** nor **non-static**.
7. Java allows two types of methods :
 - a. Static method or static member function
 - b. Non static method or non-static member function
8. The methods that are declared using static keyword are known as **static method** or **static member function**.
9. The methods that are declared without using **static** keyword are known as **non-static methods** or **non-static member function**.

Accessing static properties within the same class

Within the same class static properties can be accessed directly by its link.

```

class A
{
    static int x=10;
    static double y=12.12;
    public static void m1()
    {
        System.out.println("Hi m1");
    }

    public static void main(String[] args)
    {
        System.out.println("Hello Main");
        System.out.println(x);
        System.out.println(y);
        m1();
        System.out.println("Bye main");
    }
}
    
```

Static variable/static datamember

Static method/ static memberFunction

Main method is also static method which is called automatically by JVM

OUTPUT :

Hello Main
10
12.12
Hi m1
Bye main

Data : 01-02-2022

→ Accessing Static Properties of one class into another class :

1. We can access static properties of one class into another class by using the below syntax:
classname.Staticmembername;
2. During class loading time, that means when we pass the class file to the JVM for execution, all the static members present in the class will be loaded into **class memory** or **class area** which will have the same name as that of the class name. This is the reason we access static properties of one class into another class by using class name.

Example 1:

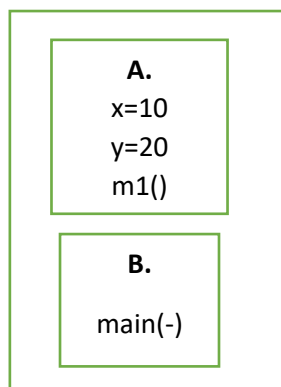
```
class A
{
    static int x=10;
    static double y=10.12;
    public static void m1()
    {
        Sop("Hello m1");
    }
}
```

```
class B
{
    p s v main(-)
    {
        sop(A.x);
        sop(A.y);
        A.m1();
    }
}
```

1. Save : ***B.java***

2. Compile : ***javac B.java***
→ A.class
→ B.class

3. Execute :
java B



Class Memory
or
Class area

```

class Demo
{
    static int a=100;
    {
        public static void m1()
        {
            -----
        }
    }
}

```

```

class Sample
{
    static boolean b=false;
    static char c= 'A';
}

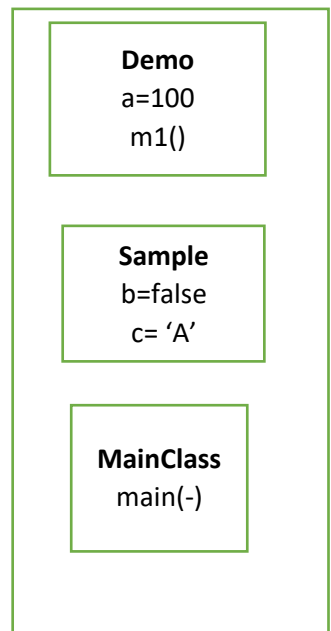
```

```

class DemoClass
{
    p s v main(-)

    sop(Sample.a);
    sop(Sample.b);
    sop(Sample.c);
    Demo.m1();
}

```



Class Memory or Class Area

(It will come into existence during class loading time.)

Interview Questions :

1. Write a program to print factors of a given number.(follow all the IT protocols.)

```

import java.util.Scanner;
class Factor
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a number:");
        int num= sc.nextInt();
        FactorsLogic.PrintFactors(num);
    }
}

class FactorsLogic
{
    public static void PrintFactors(int num)
    {
        System.out.println("=====");
        System.out.println("The factors of given number are :");
        for (int i=1;i<=num ;i++ )
        {
            if (num%i==0)
            {
                System.out.println(i);
            }
        }
    }
}

```

OUTPUT :

Enter a number:

12

=====

The factors of given number are :

1

2

3

4

6

12

2. Write a program to check if a given character is vowel or consonant.

```
import java.util.Scanner;
class VowelOrConsonant
{
    public static void main(String[] args)
    {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter a character");
        char a = sc.next().charAt(0);
        Logic.check(a);
    }
}

class Logic
{
    public static void check(char a)
    {
        if (a=='a' || a=='e' || a=='i' || a=='o' || a=='u' || a=='A' || a=='E' || a=='I' || a=='O' ||
            a=='U' )
        {
            System.out.println(a+" is a vowel character.");
        }
        else
        {
            System.out.println(a+" is a consonant character.");
        }
    }
}
```

OUTPUT :

Enter a character

i

i is a vowel character.

3. Write a program to print factorial of a given number.

```
import java.util.Scanner;
class Factorial
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a number:");
        int num= sc.nextInt();
```

```

        FactorialLogic.PrintFactorial(num);
    }
}

class FactorialLogic
{
    public static void PrintFactorial(int num)
    {
        int fact=1;
        for (int i=1;i<=num ; i++)
        {
            fact=fact*i;
        }
        System.out.println("The factorial of "+num+" is " +fact);
    }
}

```

OUTPUT :

Enter a number:

5

The factorial of 5 is 120

4. Write a program to check if a given number is prime number or not.

```

import java.util.Scanner;
class PrimeNumber
{
    public static void main(String[] args)
    {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter a number:");
        int num= sc.nextInt();
        PrimeLogic.checkPrimeNumber(num);
    }
}

class PrimeLogic
{
    public static void checkPrimeNumber(int num)
    {
        int c=0;
        for (int i=2;i<num ; i++)
        {
            if (num%i==0)
            {
                c++;
            }
        }
        if (c==0)
        {
            System.out.println(num+" is a prime number.");
        }
        else
        {
            System.out.println(num+" is not a prime number.");
        }
    }
}

```

```

    }
}
}

```

OUTPUT :

Enter a number:

7

7 is a prime number.

5. Write a program to print Fibonacci series for the given length.

```

import java.util.Scanner;
class FibonacciSeries
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n1=0, n2=1;
        System.out.println("Enter length of fibonacci series");
        System.out.println("=====");
        int length = sc.nextInt();
        FibonacciLogic.Fibonacci(length,n1,n2);
    }
}
class FibonacciLogic
{
    public static void Fibonacci(int length, int n1, int n2)
    {
        System.out.println("=====");
        System.out.println("Fibonacci series of given length are:");
        if (length<=0)
        {
            System.out.println("Oops!..Please Enter the valid integer length");
        }
        else if (length==1)
        {
            System.out.println(n1);
        }
        else if (length==2)
        {
            System.out.println(n1);
            System.out.println(n2);
        }
        else
        {
            System.out.println(n1);
            System.out.println(n2);
            for (int i=3; i<=length; i++)
            {
                int n3=n1+n2;
                System.out.println(n3);
                n1=n2;
                n2=n3;
            }
        }
    }
}

```

```

    }
}

OUTPUT :
Enter length of fibonacci series
=====
7
=====
Fibonacci series of given length are:
0
1
1
2
3
5
8

```

DATE : 02-02-2022

➔ **Accessing Non-Static properties of One class into another class.**

If we want to access the non-static properties of one class into another class , the below steps must be followed:

1. We need to create an object or instance of the class whose non-static properties must be accessed by using the below syntax:

new classname();

e.g., *new A(); //class A object is created*

new Demo(); // class Demo object is created

new Sample(); // class Sample object is created

2. Create a reference variable in order to refer to the object which is present inside **heap memory** or **heap area** by using below syntax:

classname ref_varname;

e.g., *A a1; //a1 is reference variable of class A*

Demo x; //x is reference variable of class Demo

Sample a; // a is reference variable of class Sample

3. We can combine **step 1 & step 2** into a single statement as shown below:

classname ref_varname = new classname();

e.g., *A a = new A();*

reference keyword constructor

variable

Important Note:

*Whenever the object creation statement is encountered by the **JVM** , the below steps will be followed:*

i. *An empty object will be created in the **heap memory** or **heap area**.*

ii. *All the non-static members of the class whose object is created will be loaded into the object.*

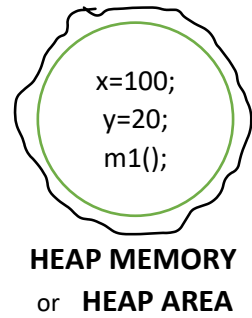
iii. *Using a reference variable, refer to the object present in the heap memory in order to access the*

non-static properties.

Example 1 :

```
class A
{
    int x=100;
    int y=20;
    public void m1()
    {
        -----
        -----
    }
}
```

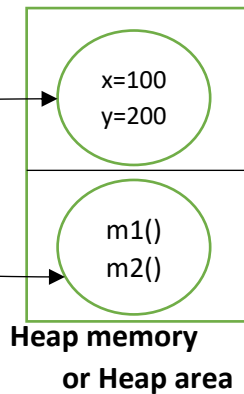
```
class B
{
    public static void main(-)
    {
        A a1 = new A();
        sop(a1.x);
        sop(a1.y);
        a1.m1();
    }
}
```



Example 2:

```
class Demo
{
    int x=100;
    int y=200;
}
class Sample
{
    public void m1()
    {
        sop("hi m1");
    }
    public double m2()
    {
        return 12.12;
    }
}
```

```
class MainClass
{
    public static void main(-- )
    {
        sop("main starts");
        Demo d1= new Demo();
        sop(d1.x);
        sop(d1.y);
        Sample s1= new Sample();
        s1.m1();
        sop(s1.m2());
        sop("main ends");
    }
}
```



OUTPUT :
main starts
100
200
hi m1
main ends

Interview Questions :

1. Write a program to find the area of triangle.

```
import java.util.Scanner;
class AreaOfTriangle
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Base");
        double base=sc.nextDouble();
        System.out.println("Enter height");
        double height=sc.nextDouble();
        AreaOfTriangleLogic d1=new AreaOfTriangleLogic();
        d1.checkArea(base,height);
    }
}
```

```

class AreaOfTriangleLogic
{
    public void checkArea(double base, double height)
    {
        double area = 0.5*base*height;
        System.out.println("Area of triangle is "+area);
    }
}

```

OUTPUT :

```

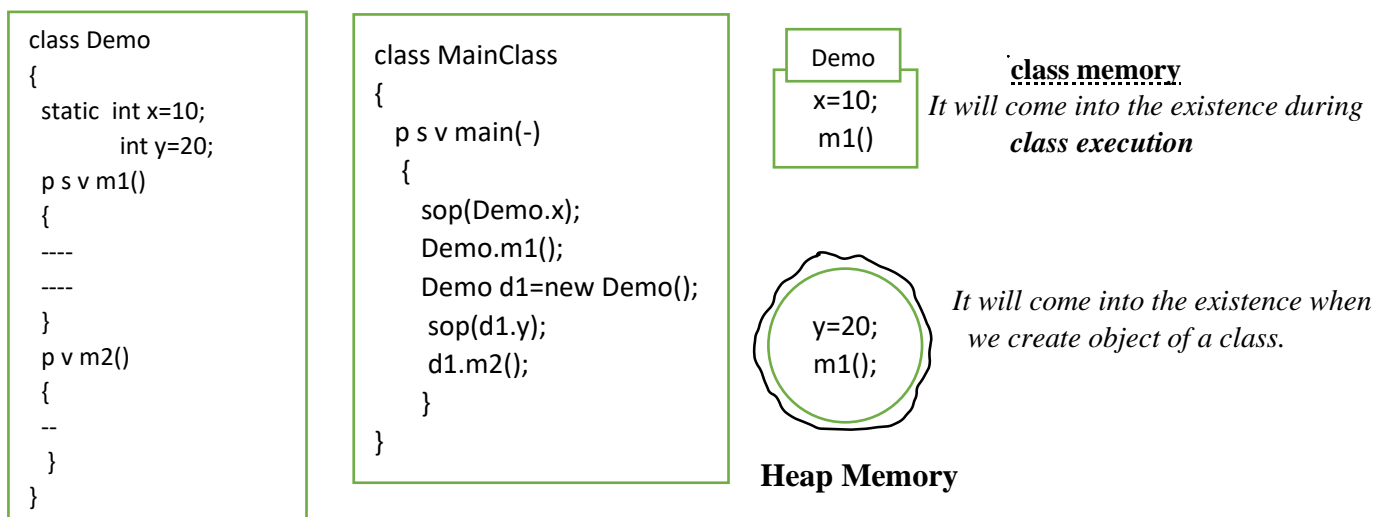
Enter Base
3.5
Enter height
4
Area of triangle is 7.0

```

Important Conclusions :

Conclusion 1:

A java class can have both **static** and **non-static members** in it, the **static** members can be accessed with the help of **classname** whereas the **non-static members** will be accessed with the help of **reference_variable** as shown in the below example:



Conclusion 2:

We cannot use a **local variable** without initializing it but it is possible to use a **static variable** & **non-static variable** without initializing or without assigning values to it. If a user is not initializing them, these variables will be given the default value depending upon their data type.

Datatype	Default Value
Int	0
Double	0.0
Boolean	False
String	Null
Char	-

class Demo

```

{
    //static variable
    static int a;
    static char c;
    static boolean e;

    //non-static variable
    double b;
    String d;
}
class MainOfDemo
{
    public static void main(String[] args)
    {
        int x=10; //local variable

        //accessing local variable
        System.out.println("x = "+x); //x=10

        //accessing static variable
        System.out.println("Demo.a = "+Demo.a);
        System.out.println("Demo.c = "+Demo.c);
        System.out.println("Demo.e = "+Demo.e);

        Demo d1=new Demo();
        //accessing non-static variable
        System.out.println("d1.b "+d1.b);
        System.out.println("d1.d "+d1.d);
    }
}

```

OUTPUT :

```

x = 10
Demo.a = 0
Demo.c = 
Demo.e = false
d1.b 0.0
d1.d null

```

DATE : 03-02-2022

Conclusion 3:

We can create any number of objects for a class which will be referred by different reference variables. If we do changes in the content of object, it will not affect the content of another object as shown below:

<pre> class A { int x=10; </pre>	<pre> class B { p s v main(-) </pre>
--------------------------------------	--

```

int y=20;
}

{
    A a1= new A();
    sop(a1.x); //10
    sop(a1.y); //20

    A a2= new A();
    sop(a2.x); //10
    sop(a2.y); //20

    a1.x=100; //re-initialization
    a2.y=200; //re-initialization

    sop(a1.x); //100
    sop(a1.y); //20
    sop(a2.x); //10
    sop(a2.y); //200
}
}

```

Conclusion 4:

One object can be referred using multiple reference variable. If we do any changes to content of the object using one reference variable, it will affect all the other reference variables pointing to that object as shown in the below example:

```

class A
{
    int x=10;
    int y=20;
}

class B
{
    p s v main(-)
    {
        A a1, a2, a3;
        a1=new A();
        a2=a1;
        a3=a1;
        sop(a1.x); //10
        sop(a2.x); //10
        sop(a3.x); //10
        sop(a1.y); //20
        sop(a2.y); //20
        sop(a3.y); //20
        a1.x=11; //re-initialization
        a3.y=22; //re-initialization
        sop(a1.x); //11
        sop(a2.x); //11
        sop(a3.x); //11
        sop(a1.y); //22
        sop(a2.y); //22
        sop(a3.y); //22
    }
}

```

Conclusion 5:

If we declare a **static variable** or **non-static variable** as **final**, we cannot **re-initialize** or **re-assign** values to them because a **final variable** can be assigned value only once through its lifetime. If at all; we try to **re-initialize** a final variable, we will get **Compile Time Error** as shown below :

```
class Sample
{
    int a=10;
    static int b=20;
}
class MainOfSample
{
    public static void main(String[] args)
    {
        System.out.println(Sample.b); //20

        Sample s1= new Sample();
        System.out.println(s1.a);

        Sample.b=200; //re-initialization of static variable
        s1.a=100; //re-initialization of non-static variable

        System.out.println(Sample.b);
        System.out.println(s1.a);
    }
}

/*
CTE at line no 15 and 17
error message: can't reassign or reinitialize a final variable
*/
```

Conclusion 6:

a. What is **Instantiation**?

The process of creating an object or instant of a class is known as **Instantiation**.

Syntax:

```
new classname();
```

e.g.:

```
new A();
new Demo();
```

b. What is **Reference Variable**?

Reference variables are **identifiers** using which we can refer to the object present in the **heap memory**.

In a variable, we can store only **primitive** type value whereas reference variable will accept only objects.

E.g., :

```
int a; // a is variable
Demo d1; //d1 is reference variable
double d; //d is variable
Sample s; // s is reference variable
```

```
a = 1234;
d1 = new Demo();
```

```
d = 12.12;
s = new Sample();
```

Conclusion 7 :

a. What is an object?

→ An object is real world entity which will have its own **states** and **behavior**. State refers to characteristics and behavior refers to functionality.

Example:

Object: Student

states/characteristics : name, age, height, weight.....

behavior/functionality : run, read, jump, write.....

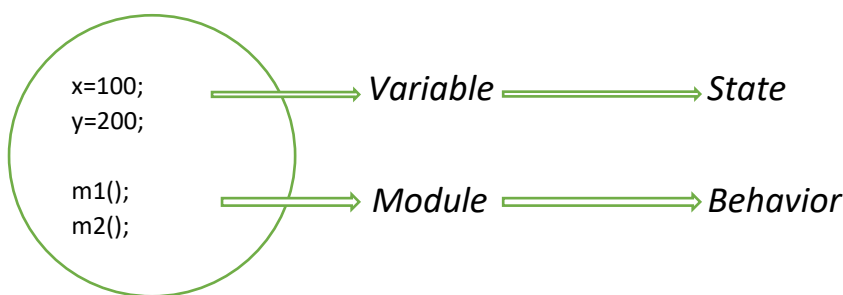
Object: car

states/characteristics : color, name, model, reg no.,.....

behavior/functionality : move →, move ← ,.....

Object With Respect To JAVA

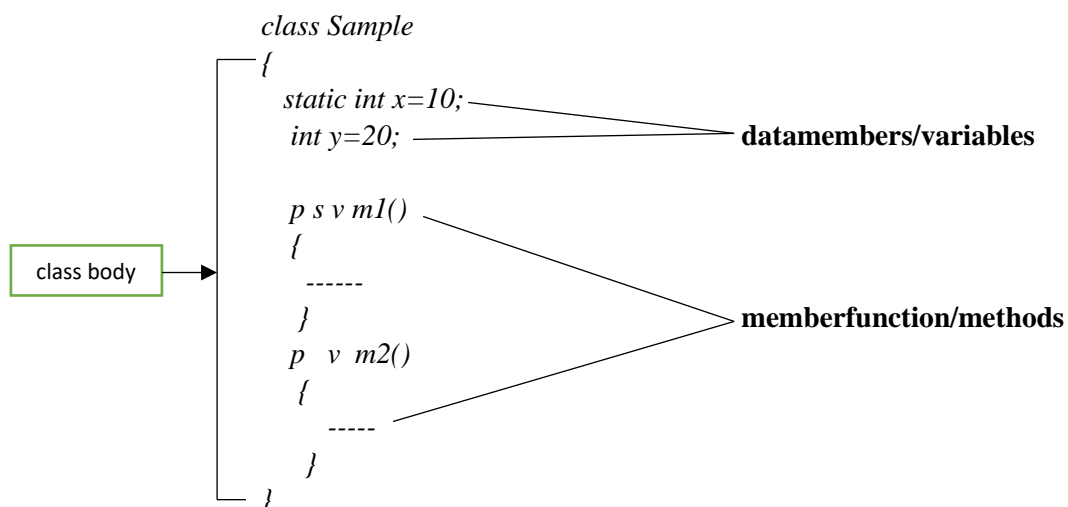
An object is an entity which will have its own **states** and **behavior**. State refers to **variables** or **data_member** and a **behavior** refers to methods or member_functions.



b. What is **class**?

A class is a java definition block or a java blueprint which will contain states and behavior. State refers to data members & behavior refers to member function.

Example:



Conclusion 8 :

→ Can we access the static properties of a class using reference variable?

Whenever we create an object of a class; a copy of its **class memory** will be given to every object. This is the reason we are able to access static properties using a reference variable but according to the IT Standards, it is highly **not recommended**. **Static members** must be accessed using the class name only.

```
class A
{
```

```
    int x=10;
```

```
    static int y=20;
```

```
}
```

```
class B
{
```

```
    p s v main(-)
    {
```

```
        sop(A.y); //20
```

```
        A a1 = new A();
        sop(a1.x); // 10
        sop(a1.y); //20
```

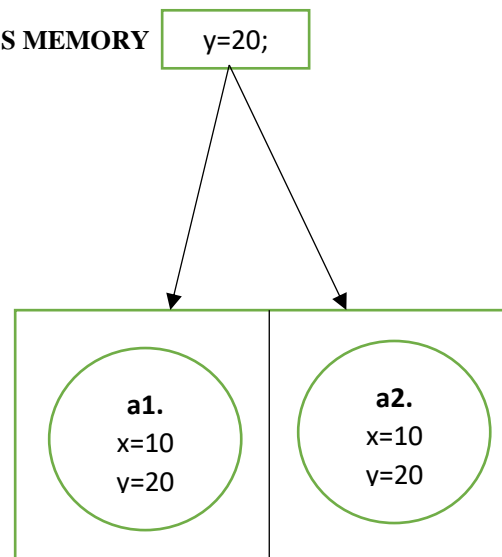
```
        A a2 = new A();
        sop(a2.x); // 10
        sop(a2.y); //20
```

```
    }
}
```

CLASS MEMORY

y=20;

Heap
Memory



Conclusion 9:

→ Deciding when to declare a member as static or non-static:

The common properties will be declared as static. E.g., College_name, College_address, Company_name, etc.

The individual properties or the property which differ from person to person will be declared as non-static.

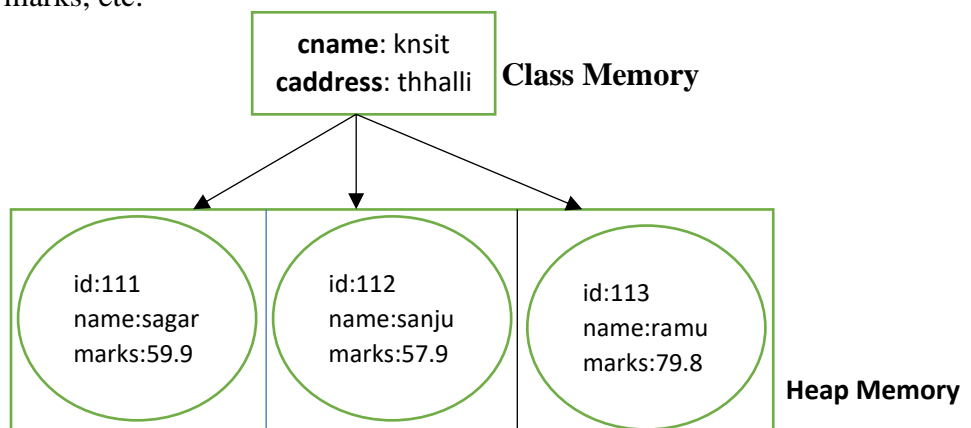
E.g., ID, name, salary, marks, etc.

Object: Student

id
name
marks
cname
caddress

cname: knsit
caddress: thhalli

Class Memory



```

class Student
{
    int id;
    String name;
    double marks;
    static String cname;
}

class MainOfClass
{
    public static void main(String[] args)
    {
        Student.cname="knsit";

        Student s1= new Student();
        s1.id=111;
        s1.name="Sagar";
        s1.marks=89.2;
        System.out.println("Student 1 details are");
        System.out.println("=====");
        System.out.println(s1.id);
        System.out.println(s1.name);
        System.out.println(s1.marks);
        System.out.println(Student.cname);

        Student s2= new Student();
        s2.id=112;
        s2.name="Sandhya";
        s2.marks=45.2;
        System.out.println("Student 2 details are");
        System.out.println("=====");
        System.out.println(s2.id);
        System.out.println(s2.name);
        System.out.println(s2.marks);
        System.out.println(Student.cname);

    }
}

```

OUTPUT :

Student 1 details are

=====

111

Sagar

89.2

knsit

Student 2 details are

=====

112

Sandhya

45.2

knsit

OBJECT INITIALIZATION

The process of initializing the non-static variables present in the object is known as **object initialization**.

According to the IT standards, the non-static variables should not be initialized inside the main method. If we want to do so, we should make use of any one of the followings :

- a) Non-static block
- b) Constructors

Block :

Blocks are set of instructions written by a user in order to perform some special task.

Types of blocks:

- i. Static Block
- ii. Non-static Block

Static block :

The blocks that are declared using **static** keyword are known as static block.

The static block will be executed automatically during class loading time that means when we load the class file to the JVM for execution, the JVM will first execute static block(if present) and then the main method.

```
class Demo
{
    static
    {
        System.out.println("static block executed successfully");
    }
    public static void main(String[] args)
    {
        System.out.println("hello world");
    }
}
```

OUTPUT :

```
static block executed successfully
hello world
```

In a class, we can have multiple static blocks, they will be executed in the same order in which they are defined during class loading time.

```
class Demo
{
    static
    {
        System.out.println("static block 1 executed successfully");
    }
    public static void main(String[] args)
    {
        System.out.println("hello world");
    }
    static
    {
        System.out.println("static block 2 executed successfully");
    }
}
```

```

        static
        {
            System.out.println("static block 3 executed successfully");
        }

    }

```

OUTPUT:

```

static block 1 executed successfully
static block 2 executed successfully
static block 3 executed successfully
hello world

```

Non-static block :

The blocks that are declared without using **static** keyword are known as non-static block. The non-static blocks will be executed automatically when we create an object or instance of a class. Whenever the object creation statement is encountered by the JVM, the below steps will be followed:

1. An empty object will be created in the heap memory.
2. All the non-static properties of the class whose object is created will be loaded into the object.
3. The class whose an object is created, its non-static block will be executed(if present).
4. Refer the object using reference variable in order to access the non-static properties.

```

class Demo
{
    {
        System.out.println("non-static block executed");
    }
    public static void main(String[] args)
    {
        System.out.println("hello main");
        new Demo();
        new Demo();
        System.out.println("bye main");
    }
}

```

OUTPUT :

```

hello main
non-static block executed
non-static block executed
bye main

```

A java class can have multiple non-static blocks in it, they will be executed in the same order in which they are defined during object creation.

```

class Demo
{
    {
        System.out.println("non-static block 1 executed");
    }
    public static void main(String[] args)
    {
        System.out.println("hello main");
        new Demo();
        System.out.println("bye main");
    }
    {
        System.out.println("non-static block 2 executed");
    }
}

```

```

    }
    {
        System.out.println("non-static block 3 executed");
    }
}

```

OUTPUT :

```

hello main
non-static block 1 executed
non-static block 2 executed
non-static block 3 executed
bye main

```

Important Conclusions on BLOCKS :

CONCLUSION 1 :

- ✓ Class can have both static block and non-static block
- ✓ The static block will be executed only once during class loading time where as the non-static block will be executed anytime when we create object of the class.

```

class Demo
{
    static
    {
        System.out.println("hey");
    }
    //non-static block
    {
        System.out.println("yo");
    }
    public static void main(String[] args)
    {
        System.out.println("hello main");
        new Demo();
        new Demo();
        System.out.println("bye main");
    }
}

```

OUTPUT :

```

hey
hello main
yo
yo
bye main

```

CONCLUSION 2 :

Though the static block is executed before the main method, we cannot load a class file to the JVM without the main method. It at all we pass a class file to the JVM which does not contains main method in it and is having only static block, the JVM will throw run time error as shown below:

```

class Demo
{
    //static block
    static

```

```

    {
        System.out.println("hey");
    }
}

```

OUTPUT:

main method not found in class Demo

CONCLUSION 3 :

Explain the purpose of static blocks in JAVA?

- ➔ Since the static block is executed before the main method, we can write those lines of code inside the static block which is required by the main method in order to start the execution of the program.

Example :

- a) Loading the in-built libraries.
- b) Opening a file
- c) Establishing connections to a database server.

CONCLUSION 4 :

Explain the purpose of non-static blocks in JAVA?

- ➔ Using non-static block, we can initialize the non-static variables present in the object.

```

class A
{
    int x;
    int y;
    {
        x=10;
        y=20;
    }
}

class B
{
    public static void main(String[] args)
    {
        A a1 = new A();
        System.out.println(a1.x); //10
        System.out.println(a1.y); //20
        A a2 = new A();
        System.out.println(a1.x); //10
        System.out.println(a1.y); //20
    }
}

```

- ➔ Whenever we initialize the non-static variables using non-static blocks, variables present across different objects will be initialized with same value. In order to overcome this issue, we implement constructors.

CONSTRUCTOR

It is a JAVA defined block or special methods which has same name as that of class name.

<access modifier> classname()

```
{  
---  
----  
---  
}
```

- To initialize a non-static variable, we use constructor.
- To achieve object initialization.

A class can have both non-static block and constructor, whenever we create object the non-static block will be executed first & then the constructor

Important NOTE :

Whenever the object creation statement is encountered by the JVM, the below steps will be followed:

1. *An empty object will be created in the heap memory.*
2. *All the non-static properties of the class whose object is created will be loaded into the heap memory.*
3. *Non-static block will be executed.*
4. *Constructor will be executed.*
5. *Create a reference variable in order to access the non-static properties present inside the object.*

We have two types of constructors :

i. Default/Compiler Defined Constructor

When a user is not declaring any constructor in the class, the JVM will write a default constructor.

They are no argument constructor

e.g.,

```
class Demo  
{  
    Demo()  
    {  
        ➔ user defined constructor  
    }  
}
```

ii. User-Defined Constructor

If a user is defining a constructor, JVM will not declare any default constructor.

Also, user-defined constructor can have parameterized and non-parameterized constructor.

Non-Parameterized(no argument) constructor

```
class Student  
{  
    String name;  
    long cont_num;  
    Student()  
    {  
        name = "Sagar";  
        cont_num = 986993473;  
    }  
}
```

```

    }
    public static void main(String[] args)
    {
        Student s1 = new Student();
        Student s2 = new Student();
        System.out.println(s1.name);
        System.out.println(s1.cont_num);
        System.out.println(s2.name);
        System.out.println(s2.cont_num);
    }
}

```

OUTPUT:

```

Sagar
986993473
Sagar
986993473

```

Parameterized(argument) constructor:

The use of parameterized constructor is to initialize the object with different values:

```

class Student
{
    String name;
    long cont_num;
    Student(String s, long ph_num)
    {
        name = s;
        cont_num = ph_num;
    }
    public static void main(String[] args)
    {
        Student s1 = new Student("Sagar",748362261);
        Student s2 = new Student("Sunar",986993974);
        System.out.println(s1.name);
        System.out.println(s1.cont_num);
        System.out.println(s2.name);
        System.out.println(s2.cont_num);
    }
}

```

OUTPUT :

```

Sagar
748362261
Sunar
986993974

```

Explain the purpose of constructors in JAVA.

Whenever we initialize the non-static variables using non-static blocks, the variables present in different object will be initialized with the same value. In order to overcome this issue, if we want to initialize the non-static variables present inside the object, we will make use of constructors.

Example program:

```

class Demo
{
    int x;
    int y;
    public Demo()

```

```

    {
        x=100;
        y=200;
    }
}

class MainClass
{
    public static void main(String[] args)
    {
        Demo d1 = new Demo();
        System.out.println(d1.x); //100
        System.out.println(d1.y); //200

        Demo d2 = new Demo();
        System.out.println(d2.x); //100
        System.out.println(d2.y); //200
    }
}

```

Difference between constructor and method.

Method	Constructor
1. Method name can be anything.	1. Constructor name must be same class name.
2. It can accept & return values.	2. It has no return type.
3. It has some access modifier(optional).	3. It has no modifier.
4. We need to explicitly call the method to execute.	4. It will execute implicitly at the time object creation.
5. Method is used to define the object and writing Logics.	5. Constructor is used to initialize the object.

CONCLUSION 1:

If a user is going to give a return type to a constructor, the JVM will treat that constructor as **non-static method**.

```

class Demo
{
    public void Demo()
    {
        System.out.println("Hi-method");
    }
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.Demo();
    }
}

```

Since, in above program, Demo() has return type, hence it becomes non-static method and we will have to extract in main method by non-static member syntax as above.

Since, the following program has **return type**, hence it is treated as **non-static method** rather than as a constructor.

```
class Demo
{
    public void Demo() → non-static method
    {
        System.out.println("Hi");
    }
}
```

CONCLUSION 2:

If we declare a modifier to a constructor, we will end up with compile time error.
Modifier static is not allowed here.

```
class Demo
{
    public static Demo() → Compile Time Error
    {
        System.out.println("Hi");
    }
}
```

OUTPUT :

```
main of Demo: java 3: error
modifier static not allowed here
public static Demo()
```


CONSTRUCTOR OVERLOADING

The process of having multiple constructors in the same class which differ in their argument, parameter or signature is known as **constructor overloading**.

The arguments of the constructor should differ in their signature i.e.,

Argument length,

Argument type,

Order of occurrence of the argument.

```
class A
{
    public A()
    {
        System.out.println("No argument");
    }
    public A(int x)
    {
        System.out.println("int argument "+x);
    }
    public A(float f, char ch)
    {
        System.out.println("float "+f+" char "+ch);
    }
    public A(char ch, float f)
    {
        System.out.println("float "+f+" char "+ch);
    }
}

class B
{
    public static void main(String[] args)
    {
        new A();
        new A(100);
        new A(10.1f, 'a');
        new A('x', 100.1f);
    }
}
```

OUTPUT :

No argument

int argument 100

float 10.1 char a

float 100.1 char x

METHOD OVERLOADING

The process of having multiple methods with same name but with different argument, parameter or signature within the same class is known as **method overloading**.

The arguments of the constructor should differ in their signature i.e.,

- Argument length,
- Argument type,
- Order of occurrence of the argument.

In JAVA, both static and non-static methods can be overloaded :

Overloading of non-static method

```
class Demo()  
{  
    public void m1()  
    {  
        System.out.println("No argument");  
    }  
    public void m1()  
    {  
        System.out.println("int argument "+x);  
    }  
    public void m1(char a,double b)  
    {  
        System.out.println("char "+a+" double "+b);  
    }  
}  
  
class MainOfDemo  
{  
    public static void main(String[] args)  
    {  
        Demo d1 = new Demo(); //compiler defined constructor will be executed  
        d1.m1();  
        d1.m1('w',10.20);  
        d1.m1(123);  
    }  
}
```

OUTPUT :

No argument
int argument 123
char w double 10.20

Overloading of static method

```
class Demo()  
{  
    public static void xyz(boolean b)  
    {  
        System.out.println("boolean argument "+b);  
    }  
    public static void xyz(int x)  
    {  
        System.out.println("int argument "+x);  
    }  
    public static void xyz(String a, char b)
```

```

        {
            System.out.println("String "+a+" char "+b);
        }
    }

    class MainOfDemo
    {
        public static void main(String[] args)
        {
            Demo.xyz(100);
            Demo.xyz(false);
            Demo.xyz("Hello", 'A');
        }
    }

```

OUTPUT :

boolean argument false
int argument 100
String Hello char A

Important Conclusions on BLOCKS :

CONCLUSION 1 :

Can we overload main method? Explain.

Yes, it is possible to overload the main method in JAVA. That means in a class, we can have multiple methods with the name **main** which differ in the arguments.

These methods will be treated like any other user defined static methods which can be accessed with the help of class name.

Example:

```

class Sample
{
    public static void main(String[] args)
    {
        System.out.println("Hello Main");
        main();
        System.out.println("Bye Main");
    }
    public static void main()
    {
        System.out.println("hey Main");
        main("Dinga");
    }
    public static void main(String S)
    {
        System.out.println("YO "+S);
    }
}

```

OUTPUT :

Hello Main
hey Main
YO Dinga
Bye Main

CONCLUSION 2 :

While performing method overloading, we can change the return type of the methods.

```
class Sample
{
    public static char m1()
    {
        return 'A';
    }
    public static String m1(int x)
    {
        System.out.println("X = "+x);
        return "Sagar";
    }
    public static int m1(double d)
    {
        System.out.println("d = "+d);
        return 100;
    }
}
class MainOfSample
{
    public static void main(String[] args)
    {
        System.out.println(Sample.m1());
        System.out.println(Sample.m1(10.12));
        System.out.println(Sample.m1(100));
    }
}
```

OUTPUT :

```
A
d = 10.12
100
X = 100
Sagar
```

CONCLUSION 3 :

Explain the purpose of method overloading in JAVA?

Using method overloading concept, we can reduce the user complexity that means: Without implementing overloading concept, user will have to remember multiple method names as shown in **case 1**: But if we implement method overloading concept, the user will have to remember only one method name as shown in **case 2**.

Case 1:

```
class Sample
{
    public void addint(int a, int b)
    {
        System.out.println(a+b);
    }
    public void adddouble(double a, double b)
    {
        System.out.println(a+b);
    }
    public void addfloat(float a, float b)
    {

```

```

        System.out.println(a+b);
    }

}

class MainOfSample
{
    public static void main(String[] args)
    {
        Sample d1 = new Sample();
        d1.addint(10,20);
        d1.adddouble(10.1,10.1);
        d1.addfloat(10.1f,10.2f);
        d1.addint(100,200);
    }
}

```

Case 2 :

```

class Sample
{
    public void add(int a, int b)
    {
        System.out.println(a+b);
    }
    public void add(double a, double b)
    {
        System.out.println(a+b);
    }
    public void add(float a, float b)
    {
        System.out.println(a+b);
    }
}

class MainOfSample
{
    public static void main(String[] args)
    {
        Sample d1 = new Sample();
        d1.add(10,20);
        d1.add(10.1,10.1);
        d1.add(10.1f,10.2f);
        d1.add(100,200);
    }
}

```

CONCLUSION 4 :

Explain the purpose of constructor overloading in JAVA?

Using constructor overloading concept, we can create multiple objects of the same class which differ in their length of the argument and type of argument.

```

class JSPEnquiry
{
    int id;
    String name;
    String mailid;
    long phno;
}

```

```

    public JSPEnquiry(int a, String b, String c, long d)
    {
        id=a;
        name=b;
        mailid=c;
        phno=d;
    }
    public JSPEnquiry(int a, String b)
    {
        id=a;
        name=b;
    }
    public JSPEnquiry(int a, String b, String c)
    {
        id=a;
        name=b;
        mailid=c;
    }
}
class MainOfJSPEnquiry
{
    public static void main(String[] args)
    {
        JSPEnquiry j1 = new JSPEnquiry(111,"Dinga", "Dingadingi.com",74825132);
        JSPEnquiry j2 = new JSPEnquiry(112,"Dingi", "Dingidinga.com",69974354);

        JSPEnquiry j3 = new JSPEnquiry(113,"Singa");
        JSPEnquiry j4 = new JSPEnquiry(44,"Jinga","sagar210.com");
    }
}

```

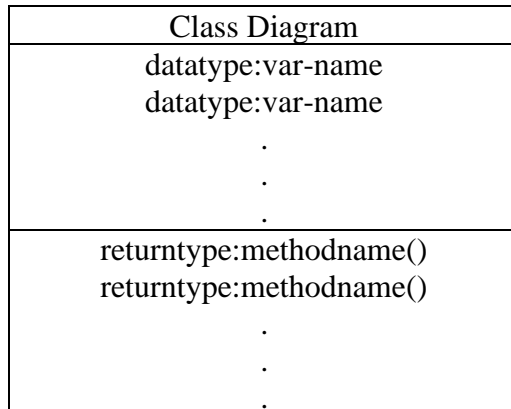
Important Note:

- In **static block**, we can access only the **static properties**.
- In **non-static block**, we can access both **static member** and **non-static member** but according to IT standards, in **non-static block**, we should initialize only **non-static member**.
- In a **constructor**, we can access both **static properties** and **non-static properties** but recommended to access only **non-static member**.

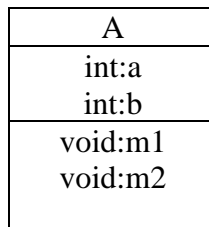
CLASS DIAGRAM

The diagram representation of a class is known as **class diagram**.

Syntax:



Example:



```
class A
{
    int a=10;
    int b=20;
    p v m1()
    {
        --
        --
    }
    p s v m2()
    {
        ---
        ---
    }
}
```

>> Different memories associated with Java :

- a) **Class Memory or class area** : All the static members of the class will be loaded into the class memory during class loading time.
- b) **Heap Memory or heap area** : All the static properties or members of the class will be loaded into the heap memory during when we create object of a class.
- c) **Method area** : All the method implementation will be loaded into the method area.
- d) **Stack Memory** : The program execution by the JVM will take place in the stack memory. The stack follows the rule of **LIFO**(Last In First Out).

INHERITANCE

The process of deriving one class property into another class is known as **inheritance**.


The class from which we derive the properties is known as **parent** class or **super** class or **base** class and the class to which we derive the properties is known as **child** class or **sub-class** or **derived** class.

In JAVA, inheritance is also known as **IS-A relationship**.

In order to inherit the properties of one class into another class, we will make use of **extends** keyword.

Syntax:

class A(<i>parent class</i>)		class B extends A
{		{
----		---- <i>child class/</i>
----		---- <i>sub class/</i>
----		---- <i>derived class</i>
}		}

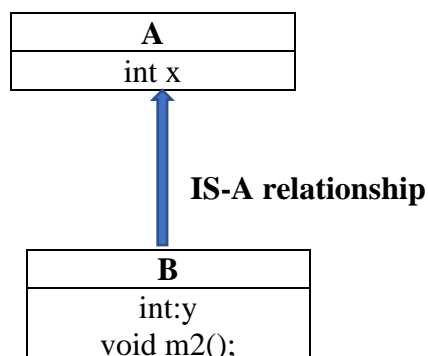


The different types of inheritance are :

- a) Single-level inheritance
- b) Multi-level inheritance
- c) Hierarchical inheritance
- d) Multiple inheritance
- e) Hybrid inheritance

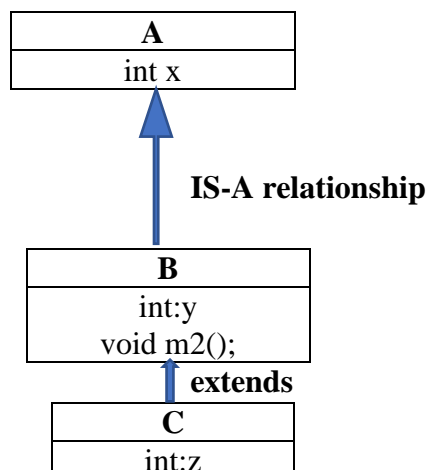
Single Level Inheritance :

A super class having a single subclass is known as single level inheritance.



Multi-level Inheritance :

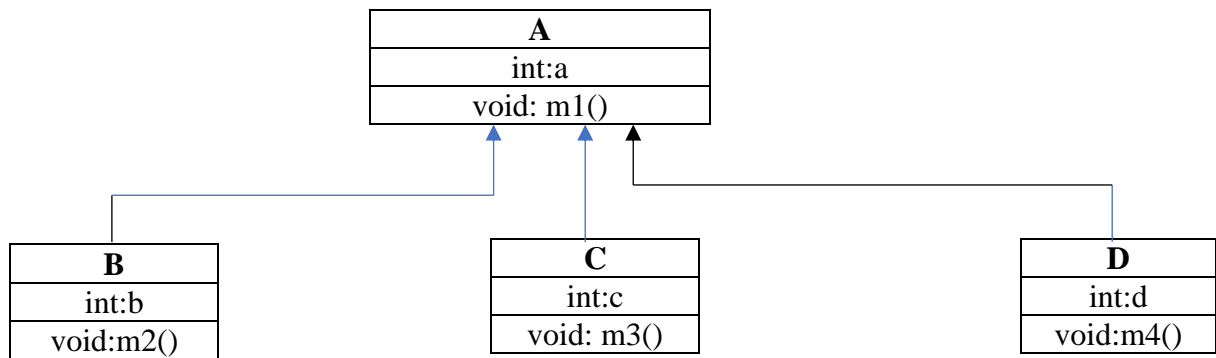
A class having a **sub class** and that sub class is acting as **super class** for its **subclass** is known as multi-level inheritance.



Hierarchical Inheritance :

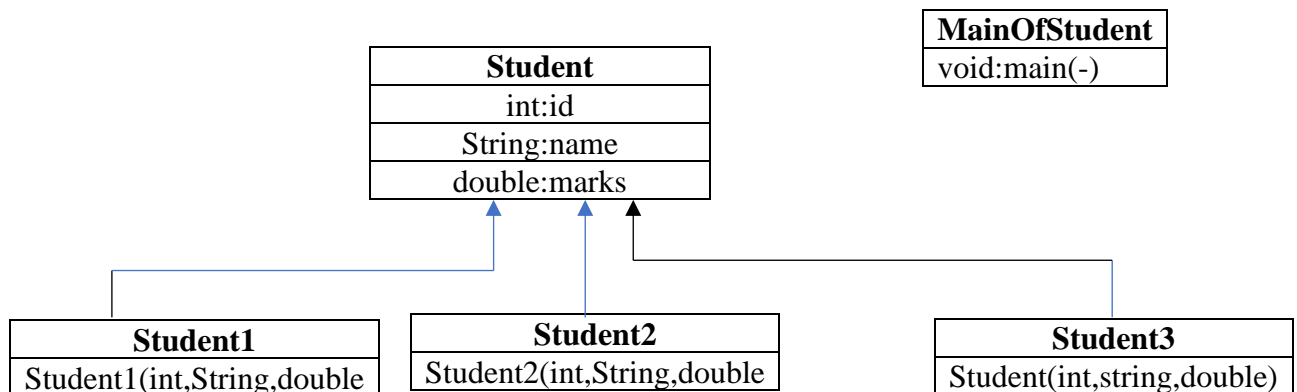
A super class having more than one sub-class is known as hierarchical inheritance.

➔ Hierarchical inheritance is the most widely used inheritance in industry.



Interview Questions:

Develop a menu-based program for the given class diagram.



Program :

```
import java.util.Scanner;
class Student
{
    int id;
    String name;
    double mark;
}
class Student1 extends Student
{
    public Student1(int a, String b, double c)
    {
        id=a;
        name=b;
        mark=c;
    }
}
class Student2 extends Student
{
    public Student2(int a, String b, double c)
    {
```

```

        id=a;
        name=b;
        mark=c;
    }
}
class Student3 extends Student
{
    public Student3(int a, String b, double c)
    {
        id=a;
        name=b;
        mark=c;
    }
}
class MainOfStudent
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("1.Student1\n2.Student2\n3.Student3");
        System.out.println("Enter your choice");
        int choice = sc.nextInt();
        switch(choice)
        {
            case 1: Student1 s1 = new Student1(111,"Dinga",53.21);
                     System.out.println(s1.id+"\t\t"+s1.name+"\t\t" + s1.mark);
                     break;
            case 2: Student2 s2 = new Student2(121,"Yinga",73.41);
                     System.out.println(s2.id+"\t\t"+s2.name+"\t\t" + s2.mark);
                     break;
            case 3: Student3 s3 = new Student3(113,"Pinga",66.92);
                     System.out.println(s3.id+"\t\t"+s3.name+"\t\t" + s3.mark);
                     break;
            default : System.out.println("Invalid Choice");
        }
    }
}

```

OUTPUT :

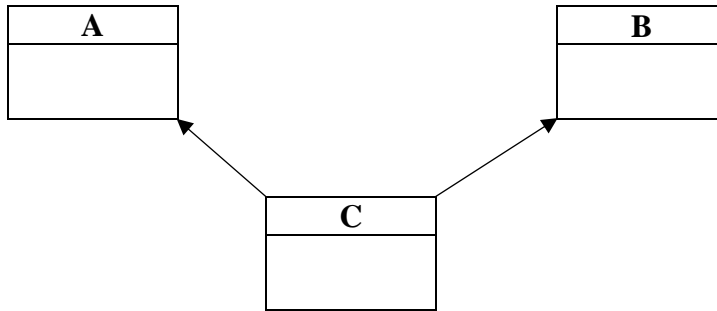
```

1.Student1
2.Student2
3.Student3
Enter your choice
1
111      Dinga      53.21

```

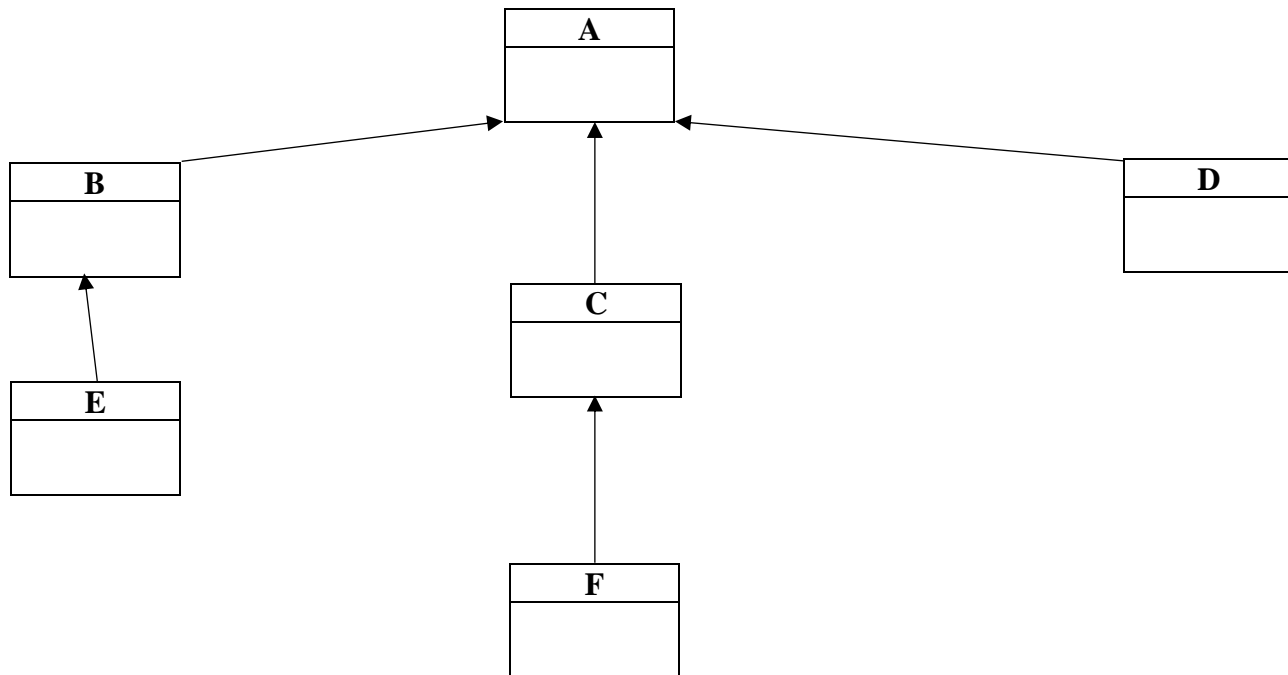
Multiple Inheritance :

Many super classes with only one sub-class. It is not allowed in Java.



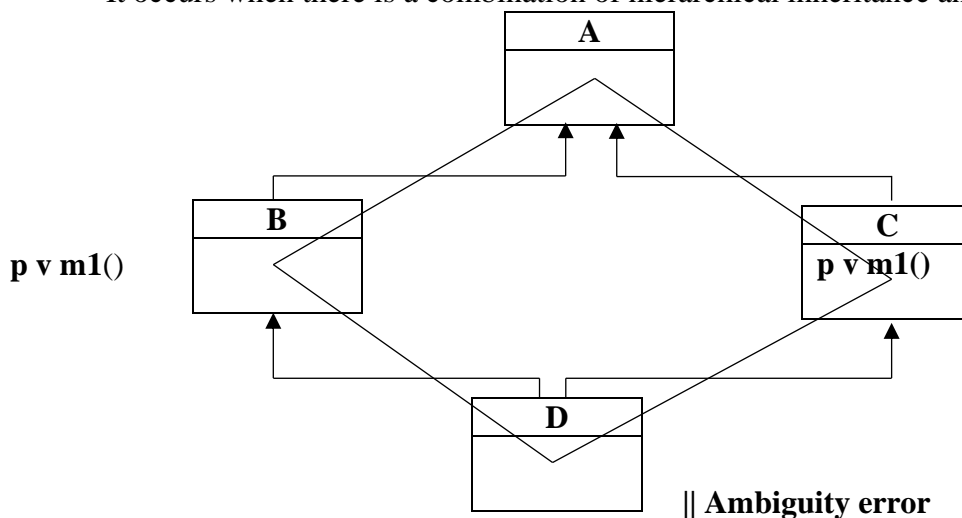
Hybrid Inheritance :

It is the combination of both multi-level and hierarchical inheritance.



Diamond-Ring Problem :

It occurs when there is a combination of hierarchical inheritance and multiple inheritance.



Reasons :

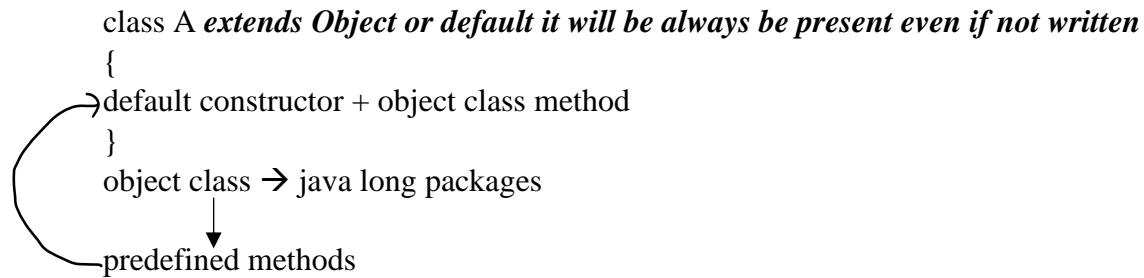
- * Because it contains multiple inheritance.
- * In **DRP**(Diamond Ring Problem) in class B & in class C contains same method with same signature. So the class A or JVM will get confused which class method to access & it will give error as ambiguity error.

CONCLUSION ON INHERITANCE

CONCLUSION 1 :

Object class is the super most class.

All the classes in the source file are all sub-classes.



CONCLUSION 2 :

final int x *//we cannot re-initialize*

final class A *//we cannot extends if we make a class or final*

```
{
  ----
  ----
}
```

Example for pre-defined final class

String class

String buffer

String Builder

*** For this, we can have **super** class but not **sub class**.

CONCLUSION 3 :

Inheritance is inheriting the properties of one class to another class.

But there are two types of properties that cannot be inherited:

→ Private Members

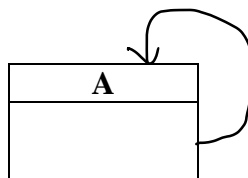
→ Constructor *//rule is constructor name should be same as class name*

CONCLUSION 4 :

A cyclic inheritance is not possible.

A class extends itself is known as cyclic inheritance.

E.g. 1 ; class A extends A



2. class A extends B

```
{
  ----
}
```

class B extends A

```
{
  ----
}
```

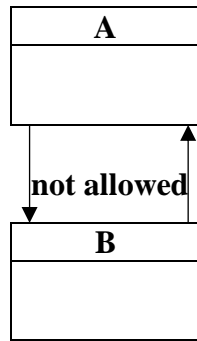


Fig. : Cyclic Inheritance

this keyword :-

It is a keyword which is used to refer to the **current object**.

The object using which we are calling something is known as **current object**.

Usually, we make use of this keyword in order to differentiate between the local variables and non-static variables. Since this keyword is associated with the object, we can make use of it only inside a constructor or non-static method.

It is optional to use this keyword when the local variable names and the non-static variable are different.

It is mandatory to use this keyword when the local variable names and non-static variables name are same.

Example Program :

```
class A
{
    public int x;
    public int y;
    public A(int x)
    {
        this.x = 30;
        System.out.println(x);
        System.out.println(this.x);
    }
    public void m1(int y)
    {
        System.out.println(y);
        this.y = y;
    }
}
class main
{
    public static void main(String[] args)
    {
        A a = new A(10);
        a.m1(20);
    }
}
```

OUTPUT :

```
10
30
20
```

INTERVIEW QUESTIONS

Create a faculty class with the attributes id, name and salary. Initialize these attributes with the help of constructor and this keyword. And then display the details of 3 faculties.

```
class Faculty
{
    int id;
    String name;
    double salary;
    public Faculty(int id,String name, double salary)
    {
        this.id=id;
        this.name=name;
        this.salary=salary;
    }
    public void DisplayDetails()
    {
        System.out.println(this.id+"\t\t"+this.name+"\t\t"+this.salary);
    }
}
class MainOfFaculty
{
    public static void main(String[] args)
    {
        System.out.println("Id\t\tName\t\tSalary");
        System.out.println("=====");
        Faculty F1 = new Faculty(112,"Dinga",42.20);
        F1.DisplayDetails();
        Faculty F2 = new Faculty(145,"Cinga",45.20);
        F2.DisplayDetails();
        Faculty F3 = new Faculty(132,"Tinga",46.15);
        F3.DisplayDetails();
    }
}
```

OUTPUT :

Id	Name	Salary
112	Dinga	42.2
145	Cinga	45.2
132	Tinga	46.15

Super keyword :

It is a keyword which is used to access the super class properties in the sub class. Using **super** keyword, we can differentiate between the sub-class & non-static variable with its super class: non static variable.

It is optional to use **super** keyword when the subclass variable/member names and **super class** member names are different. However, it is mandatory to use **super** keyword when sub class member names and super class member names are same.

Using super keyword, we can access the properties of immediate super class only.

Example :

```
class A
{
    public A()
    {
```

```

        this(10);
        System.out.println("A-class constructor");
    }
    public A(int x)
    {
        System.out.println("Int-argument constructor");
    }
}
class B extends A
{
    public B()
    {
        this(10);
        System.out.println("B-class Constructor");
    }
    public B(int y)
    {
        super();
        System.out.println("Hi");
    }
}
class Main
{
    public static void main(String[] args)
    {
        B b = new B();
    }
}

```

OUTPUT :

Int-argument constructor
 A-class constructor
 Hi
 B-class Constructor

NOTE :

Just like **this** keyword, even **super** keyword can be used only inside a constructor & non-static method. Using **super** keyword, we can access the properties of the immediate super class only.

Example Program :

```

class Demo1
{
    int a=1;
    int b=2;
}
class Demo2 extends Demo1
{
    int a=10;
    int b=20;
    public void display()
    {
        System.out.println("In display method of Demo2 class");
        System.out.println("this.a "+this.a);
        System.out.println("this.b "+this.b);
    }
}

```

```

        System.out.println("Super.a "+super.a);
        System.out.println("Super.b "+super.b);
    }
}
class Demo3 extends Demo2
{
    int a=100;
    int b=200;
    public void display()
    {
        System.out.println("In display method of Demo3 class");
        System.out.println("this.a "+this.a);
        System.out.println("this.b "+this.b);
        System.out.println("Super.a "+super.a);
        System.out.println("Super.b "+super.b);
        super.display(); //calling method of super class
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Demo3 d1 = new Demo3();
        d1.display();
        System.out.println("Main Ends");
    }
}

```

OUTPUT :

```

Main Starts
In display method of Demo3 class
this.a 100
this.b 200
Super.a 10
Super.b 20
In display method of Demo2 class
this.a 10
this.b 20
Super.a 1
Super.b 2
Main Ends

```


CONSTRUCTOR CALLING :

Calling one constructor from another constructor within the same class is known as **constructor calling**.

We can perform constructor calling by making use of **this()** method.

this() method can be used only inside constructor and it must always be the first statement of constructor body.

```
class A
{
    public A()
    {
        this(10);
        System.out.println("No argument Constructor");
    }
    public A(int x)
    {
        this(10,'c');
        System.out.println(x+" = Int-argument Constructor");
    }
    public A(int x, char c)
    {
        System.out.println(x+" Integer value " + c + " character constructor");
    }
}
class main
{
    public static void main(String[] args)
    {
        A a = new A();
    }
}
```

OUTPUT :

```
10 Integer value c character constructor
10 = Int-argument Constructor
No argument Constructor
```

CONSTRUCTOR CHAINING :

Definition:

The process of calling **super class** constructor from the **sub class** is known as **constructor chaining**.

- We can perform constructor chaining by making use of “**super()**” method.
- Super method can be used only inside a constructor and it must always be the first statement of constructor body.
- Using **super** method, we can execute the constructor of immediate **super** class only.

Example :

```
class A
{
    public A(char A)
    {
        -----
        -----
        -----
    }
}
class B extends A
{
    public B(int x)
    {
        super('A');
        -----
        -----
    }
}
class C
{
    public static void main(String[] args)
    {
        return B(10);
    }
}
```

The constructor chaining process is mandatory that means whenever we create object of sub class, its super class constructor has to get executed in order to complete the constructor chaining process.

That is why in Java, the constructor chaining process happens in 2 ways:

- Implicitly by the JVM
- Explicitly by a user.

The constructor chaining process will take place **implicitly** by the JVM when the super class contains **no-argument constructor(default constructor)**.

The constructor chaining process should take place explicitly by the user when the **super** class contains **argument** constructor.

Example for Implicit Constructor Chaining

```
class A
{
    public A()
    {
        -----
    }
}
class B extends A
{
    public B(int x)
    {
        super(); //JVM
        -----
    }
}
class C
{
    public static void main(String[] args)
    {
        return B(10);
    }
}
```

The diagram illustrates implicit constructor chaining. In class B, the constructor `public B(int x)` calls `super();` which is automatically inserted by the JVM. In class C, the `main` method calls `return B(10);`. Arrows show the flow of execution from the `main` method to the `B` constructor and then to the `super()` call.

Example for explicit Constructor chaining

```
class A
{
    public A(double d)
    {
        -----
    }
}
class B extends A
{
    public B()
    {
        super(12.12); //User
        -----
    }
}
class C
{
    public static void main(String[] args)
    {
        new B();
    }
}
```

The diagram illustrates explicit constructor chaining. In class B, the constructor `public B()` explicitly calls `super(12.12);`. In class C, the `main` method calls `new B();`. Arrows show the flow of execution from the `main` method to the `B` constructor and then to the `super(12.12)` call.

Important Conclusions

CONCLUSION 1 :

this	this()	super	super()
It is keyword.	It is a special method.	It is a keyword.	It is a special method.
It is used to refer to the current object.	It is used for performing constructor calling.	It is used to access super class properties in the sub-class .	It is used to perform constructor chaining.
It can be used inside non-static method and constructor.	It can be used only inside a constructor.	It can be used inside non-static method and constructor.	It can be used inside a constructor only.
It can be used anywhere within the constructor or non-static method body.	It should be the first statement of constructor body.	It can be used anywhere within the constructor or non-static super body.	It should be the first statement of constructor body.
IS-A relationship is optional.	IS-A relationship is optional.	IS-A relationship is mandatory.	IS-A relationship is mandatory.

CONCLUSION 2 :

Explain why is multiple inheritance not allowed in Java?

As per Java rule, whenever we create object of a sub class , its super class constructor has to get executed in order to complete the constructor chaining process.

In multiple inheritance, one sub-class will have multiple superclass. When we create object of the sub-class, it will lead to an ambiguity to which super class constructor should be executed first and due to this reason, the constructor chaining process will remain incomplete.

That's why, multiple inheritance is not allowed in Java.

CONCLUSION 3 :

Can we achieve both constructor calling and constructor chaining in a single constructor?

Since both **this()** and **super()** must be the first statement of constructor body, hence we can either achieve constructor calling or constructor chaining at a time but not both in a single constructor.

CONCLUSION 4 :

Demonstrate constructor calling and constructor chaining in a single program.

```
class A
{
    public A(int x)
    {
        this(10.1,true);
        System.out.println(x);
    }
    public A(char y)
    {
        System.out.println(y);
    }
}
```

```

        public A(double a, boolean b)
        {
            this('A');
            System.out.println(a);
            System.out.println(b);
        }
    }
    class B
    {
        public B(float f)
        {
            this(100);
            System.out.println(f);
        }
        public B(int a)
        {
            super(200);
            System.out.println(a);
        }
        public B(char c, String s)
        {
            this(10.1f);
            System.out.println(c);
            System.out.println(s);
        }
    }
    class C
    {
        public static void main(String[] args)
        {
            System.out.println("Main Starts");
            new B('a', "hi");
            System.out.println("Main Ends");
        }
    }
}

```

CONCLUSION 5 :

Explain the purpose of constructor calling in Java.

With the help of **this()**, we can invoke all the constructors of a class just by creating a single object.

CONCLUSION 6 :

Explain the purpose of constructor chaining in Java.

With the help of constructor chaining concept, JVM will get to know who is the super class of the current class and along with this using **super()**, we can also initialize the super class properties from the sub class.

Interview Question :

Write a program to initialize the super class members in super class constructor without creating super class object.

```

class Parent
{
    int id;
}

```

```

        String name;
        int age;
        public Parent(int id,String name,int age)
        {
            this.id=id;
            this.name = name;
            this.age = age;
        }
    }
    class Child extends Parent
    {
        public Child(int id,String name, int age)
        {
            super(id,name,age); //as per the question calling super class,
                                constructor is initialized the properties
        }
    }
    class MainClass
    {
        public static void main(String[] args)
        {
            Child cl = new Child(111,"Sagar",23);
            System.out.println("cl.id = "+cl.id);
            System.out.println("cl.name = "+cl.name);
            System.out.println("cl.age = "+cl.age);
        }
    }
}

```

OUTPUT :

```

cl.id = 111
cl.name = Sagar
cl.age = 23

```

Packages :

A java package is **a group of similar types of classes, interfaces and sub-packages**.

Package in java can be categorized in two form, **built-in** package and **user-defined** package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Packages are used for:

- Preventing naming conflicts. For example, there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

Data Hiding :

The process of hiding data members or variables of a class from being accessed from another class directly is known as data hiding.

We achieve the highest level of data hiding by making use of **private access modifier**.

In all object-oriented programming(**OOP**) language, the primary focus will be on data member rather than member function that means we need to restrict direct accessing of data members. If any outside user wants to access these private data members, then they can access only via helper methods.(**getter & setters**)

Get() methods are used to get the **private** data member value.

Set() methods are used to re-initialize or initialize the private data member.

For every private dataMember, we will have separate **get()** & **set()**.

Example:

```
class A
{
    private int X=10;
    private int Y=20;

    public int getX()
    {
        return X;
    }
    public int getY()
    {
        return Y;
    }
    public void setX(int a)
    {
        X=a;
    }
    public void setY(int b)
    {
        Y=b;
    }
}

class B
{
    public static void main(String[] args)
    {
        A a1 = new A();
        System.out.println(a1.X); //CTE
        System.out.println(a1.Y); //CTE

        System.out.println(a1.getX());
        System.out.println(a1.getY());

        a1.X=100; //CTE
        a1.Y=200; //CTE
        a1.setX(100);
        a1.setY(200);

        System.out.println(a1.getX());
        System.out.println(a1.getY());
    }
}
```

JAVA Bean Class :

A public class with private data members, public constructor and public get() & set() is known as JAVA bean class.

Using Java bean class, we **can achieve data hiding**. Java Bean class is a very good example of **encapsulation**.

Create a student class with the private attributes, id, name and marks. Initialize these attributes with the help of constructor for 3 students and perform the below operation:

a. Display all the students' details.

```
class Student
{
    private int id;
    private String name;
    private double marks;

    public Student(int a, String b, double c)
    {
        id=a;
        name=b;
        marks=c;
    }

    public int getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public double getMarks()
    {
        return marks;
    }
}

class MainClass
{
    public static void main(String[] args)
    {
        Student s1 = new Student(111,"Dinga",45.12);
        Student s2 = new Student(112,"Sagar",51.32);
        Student s3 = new Student(113,"Sunar",52.32);

        System.out.println("Id\t\tName\t\tMarks");
        System.out.println("=====");
        System.out.println(s1.getId()+"\t\t"+s1.getName()+"\t\t"+s1.getMarks());
        System.out.println(s2.getId()+"\t\t"+s2.getName()+"\t\t"+s2.getMarks());
        System.out.println(s3.getId()+"\t\t"+s3.getName()+"\t\t"+s3.getMarks());
    }
}
```


OUTPUT :

Id	Name	Marks
=====	=====	=====
111	Dinga	45.12
112	Sagar	51.32
113	Sunar	52.32

Create a Mobile class with the private attributes name, price and color. Initialize these attributes with the help of constructors and display the details of 4 Mobile.

```

class Mobile
{
    private String name;
    private double price;
    private String color;

    public Mobile(String a, double b, String c)
    {
        name=a;
        price=b;
        color=c;
    }

    public String getName()
    {
        return name;
    }

    public double getPrice()
    {
        return price;
    }

    public String getColor()
    {
        return color;
    }
}

class MainClassOfMobile
{
    public static void main(String[] args)
    {
        Mobile m1 = new Mobile("Redmi",15999,"Blue");
        Mobile m2 = new Mobile("Vivo",19999,"Dark Green");
        Mobile m3 = new Mobile("Samsung",33999,"Golden Brown");
        Mobile m4 = new Mobile("Iphone",66999,"White");

        System.out.println("Name\t\tPrice(in INR)\t\tColour");
        System.out.println("=====");
        System.out.println(m1.getName()+"\t\t"+m1.getPrice()+"\t\t"+m1.getColor());
        System.out.println(m2.getName()+"\t\t"+m2.getPrice()+"\t\t"+m2.getColor());
        System.out.println(m3.getName()+"\t\t"+m3.getPrice()+"\t\t"+m3.getColor());
        System.out.println(m4.getName()+"\t\t"+m4.getPrice()+"\t\t"+m4.getColor());
    }
}

```

OUTPUT :

Name	Price(in INR)	Colour
Redmi	15999.0	Blue
Vivo	19999.0	Dark Green
Samsung	33999.0	Golden Brown
Iphone	66999.0	White

ENCAPSULATION :

Binding of related data together is known as **encapsulation**.

Real-time example:

- A classroom is encapsulation of chairs, system, students, faculty, etc.
- A school bag is encapsulation of books, pen, water bottle, etc.

Examples with respect to JAVA :

A class is an encapsulation of data members and member function.

A package is an encapsulation of classes and interfaces.

rt.jar file is an encapsulation of JAVA executable files.

Method Overriding :

Whenever two classes are having **IS-A** relationship, super class properties will be inherited to the sub class. If the sub-class is not happy with any of its super-class method's implementation, the sub-class can change the implementation of that method as per its requirement. This property of changing super class implementation in the sub-class is known as **method overriding**.

The method's name and signature must be **same** while overriding.

Is-a relationship is mandatory between the classes while overriding the method.

Example 1 :

```
class Parent
{
    public void marriage()
    {
        System.out.println("Anuska");
    }
}
class Child extends Parent
{
    @Override //annotation which specifies that this method originally.
    public void marriage() belongs to its super class. It's overridden here
    {
        System.out.println("Deepika");
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Parent P = new Parent();
        P.marriage(); //Anuska

        Child C = new Child();
        C.marriage(); //Deepika
    }
}
```

Example 2:

```
class Parent
{
    public void place()
    {
        System.out.println("Kashi");
    }
    public void travel()
    {
        System.out.println("train");
    }
    public void stay()
    {
        System.out.println("ashram");
    }
}
class Child extends Parent
{
    @Override
    public void place()
    {
        System.out.println("Goa");
    }
    @Override
    public void stay()
    {
        System.out.println("resort");
    }
}
class MainOfTrip
{
    public static void main(String[] args)
    {
        Parent P = new Parent();
        P.place();
        P.travel();
        P.stay();
        Child C = new Child();
        C.place();
        C.travel();
        C.stay();
    }
}
```

OUTPUT :

Kashi
train
ashram
Goa
train
resort

Important Conclusions on Method Overriding

CONCLUSION 1 :

Overriding concept is applicable only for methods not for variables. Changing the value of a variable is known as **re-initialization** , not overriding.

CONCLUSION 2 :

If we declare a method as **final**, the method cannot be overridden in the **sub-class**.

Example :

```
class Parent
{
    final public void place()
    {
        System.out.println("Kashi");
    }
}
class Child extends Parent
{
    @Override
    public void place()
    {
        System.out.println("Goa");
    }
}
class MainOfTrip
{
    public static void main(String[] args)
    {
        Parent P = new Parent();
        P.place();

        System.out.println();

        Child C = new Child();
        C.place();
    }
}
```

OUTPUT window :

```
error: place() in Child cannot override place() in Parent
    public void place()
           ^
    overridden method is final
```

Important Note :

1. Final variable cannot be re-initialized.
2. Final method cannot be overridden.
3. Final class will not support inheritance or not have any sub-class.

CONCLUSION 3 :

While overriding super class method in the sub-class, either we should retain the same visibility of the method or we can increase the visibility but we should not decrease the visibility of the method by overriding.

Private methods cannot be overridden because private members will not get inherited to the sub-class.

1. <pre>class A { public v m1() { ---- ---- } } class B extends A { public v m1() { ---- ---- } }</pre>	2. <pre>class A { protected v m1() { ---- ---- } } class B extends A { protected/public v m1() { ---- ---- } }</pre>	3. <pre>class A { public v m1() { ---- ---- } } class B extends A { default/protected/public v m1() { ---- ---- } }</pre>	4. <pre>class A { private v m1() { ---- ---- } } class B extends A { private members will not get inherited. }</pre>
---	--	---	--

CONCLUSION 4 :

While overriding the super class method in the sub-class, we cannot change the return type of the method.(with respect to primitive time)

```
class A
{
    public void m1()
    {
        ----
        ----
    }
}
class B extends A
{
    public int m1()
    {
        CTE, cannot change return type
    }
}
```

CONCLUSION 5 :

Overriding concept is applicable only for object-level methods. That means, we cannot override static methods.

Only non-static methods can be overridden, the same concept with respect to static method is known as **method hiding**.

Difference between Overloading and Overriding.

Overloading	Overriding
1. Performing one task in multiple ways is known as overloading.	1. Changing the task itself is known as overriding.
2. It can happen within the same class.	2. It can happen only across different classes.
3. Is-a relationship is not mandatory.	3. Is-a relationship is mandatory.
4. Method name should be same but arguments must be different.	4. Both method name and signature must be same.
5. Constructors can be overloaded.	5. Constructors cannot be overridden.
6. Private method and final method can be overloaded.	6. Final methods and private method cannot be overridden.
7. Main method can be overloaded.	7. Main method cannot be overloaded.

Eclipse Software :

It is an IDE(Integrated Development Environment) which is used to fasten the coding process. The programs written by a developer on eclipse will be compiled automatically and the errors will be fixed on the spot.

Using eclipse, the development process can happen faster.

* **Creating a project under eclipse :**

File → new → select Java Project

* **Creating a package under a project**

Right click on src → new → select Package

Standard Syntax for creating a package :

domainname.companyname.projectname.modulename

domain- **com** for commercial, **edu** for educational and **org** for organization.

e.g., edu.jspiders.myproject.Basics

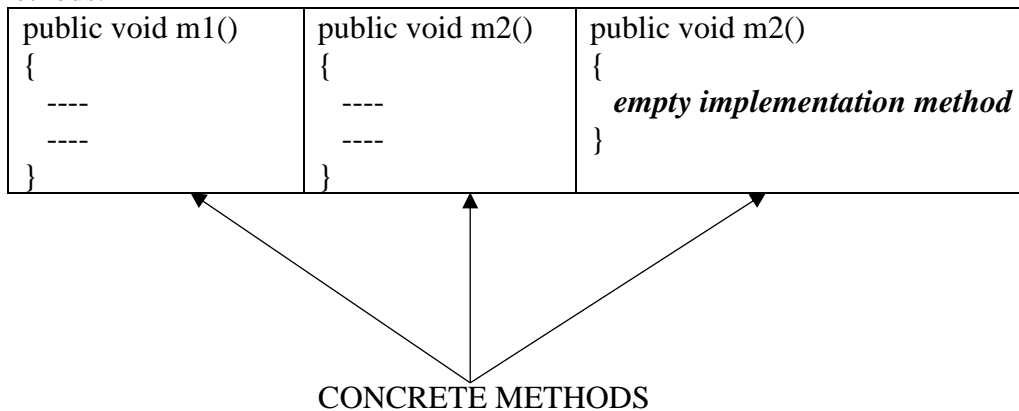
* **Create a class under a package**

Right click on package → new → class

ABSTRACT CLASS & ABSTRACT METHOD

Concrete methods :

The methods which will have some implementation or the methods which will have body are known as concrete methods.



Abstract methods :

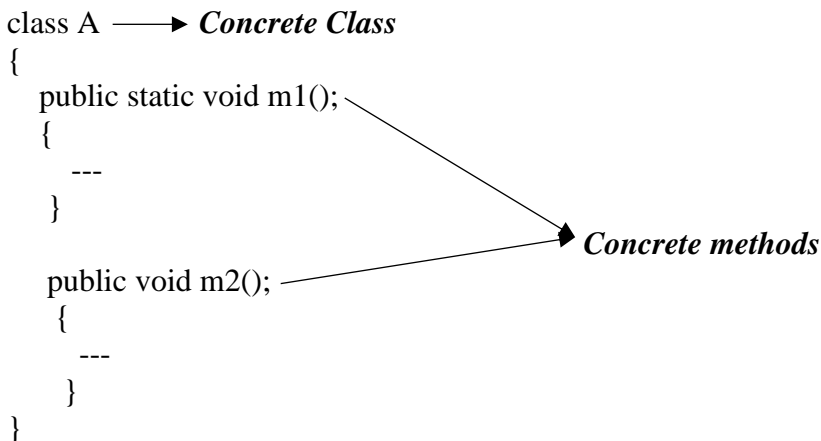
Methods without any implementation are known as abstract method and such method should be mandatory declared as **abstract** as shown in the below example :

abstract public void m1();
or
public **abstract** void m1();

abstract public void display();
or
public **abstract** void display();

Concrete class :

The class which contains only concrete methods inside it are known as concrete class.



Abstract Class :

Define Abstract.

When we want to do something but we don't know how to do it at that particular moment, such scenarios are known as **abstract**.

The class which are declared using **abstract** keyword are known as abstract class.

If a class contains at least one abstract method in it, it must be declared as **abstract** mandatorily.

We will provide for abstract method of an abstract class in its sub-class.

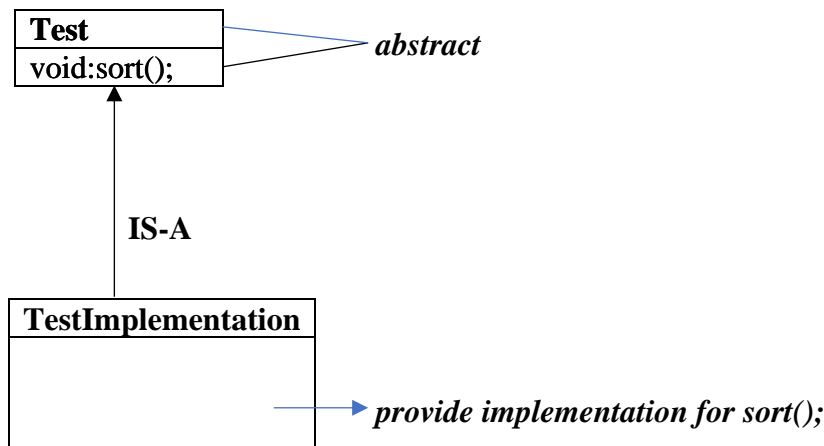
As per Java rule, we cannot create an object or instance of an abstract class.


```

abstract class A → abstract class
{
    public void m2()
    {
        ----- → Concrete method
    }
    abstract public void m1(); → abstract method
}

```

Example :



```

package edu.jspiders.myfirstproject.abstractlearning1;

```

```

public abstract class Test
{
    abstract public void sort();
}

```

```

package edu.jspiders.myfirstproject.abstractlearning1;

```

```

public class TestImplementation extends Test
{
    public void sort()
    {
        System.out.println("sorting done");
    }
}

```

```

package edu.jspiders.myfirstproject.abstractlearning1;

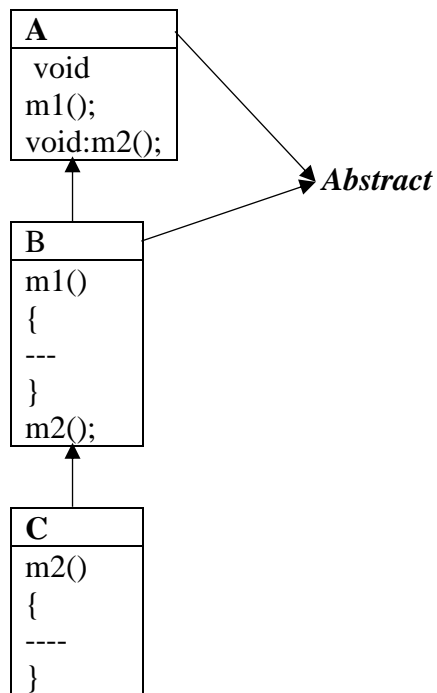
```

```

public class MainClass
{
    public static void main(String[] args)
    {
        TestImplementation tm = new TestImplementation();
        tm.sort();
    }
}

```

If the sub-class fails to provide implementation for all the abstract methods of its super class, the sub-class also must be declared as abstract and the implementation should be provided in further sub-class as shown in the below scenario :



```
package edu.jspiders.myfirstproject.abstractlearning2;
```

```
abstract public class A
```

```
{
    abstract public void m1();
    abstract public void m2();
}
```

```
package edu.jspiders.myfirstproject.abstractlearning2;
```

```
abstract public class B extends A
```

```
{
    public void m1()
    {
        System.out.println("m1 implemented");
    }
}
```

```
package edu.jspiders.myfirstproject.abstractlearning2;
```

```
public class C extends B
```

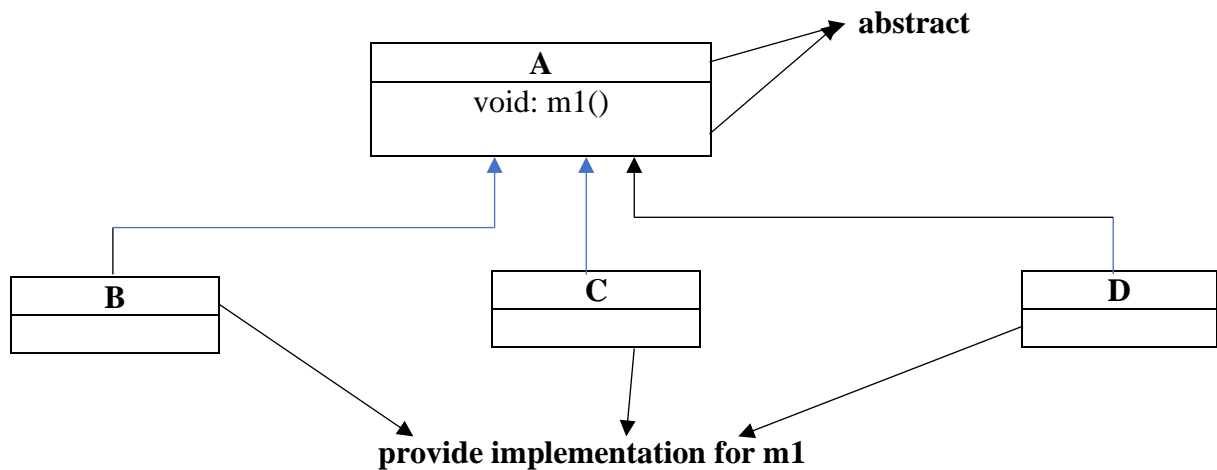
```
{
    public void m2()
    {
        System.out.println("m2 implemented");
    }
}
```

```
package edu.jspiders.myfirstproject.abstractlearning2;
```

```
public class MainClass
```

```
{
    public static void main(String[] args)
    {
        C c1 = new C();
        c1.m1();
        c1.m2();
    }
}
```

For an abstract method of an abstract class, we can provide different implementation in its different sub classes as shown in the below scenario:



```
package edu.jspiders.myfirstproject.abstractlearning3;
```

```
public abstract class A
{
    abstract public void m1();
}
```

```
package edu.jspiders.myfirstproject.abstractlearning3;
```

```
public class B extends A
{
    public void m1()
    {
        System.out.println("m1 implement in B");
    }
}
```

```
package edu.jspiders.myfirstproject.abstractlearning3;
```

```
public class C extends A
{
    public void m1()
    {
        System.out.println("m1 implement in C");
    }
}
```

```
package edu.jspiders.myfirstproject.abstractlearning3;
```

```
public class D extends A
{
    public void m1()
    {
        System.out.println("m1 implement in D");
    }
}
```

```
package edu.jspiders.myfirstproject.abstractlearning3;
```

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

```

    {
        B b = new B();
        b.m1();

        C c = new C();
        c.m1();

        D d = new D();
        d.m1();
    }
}

```

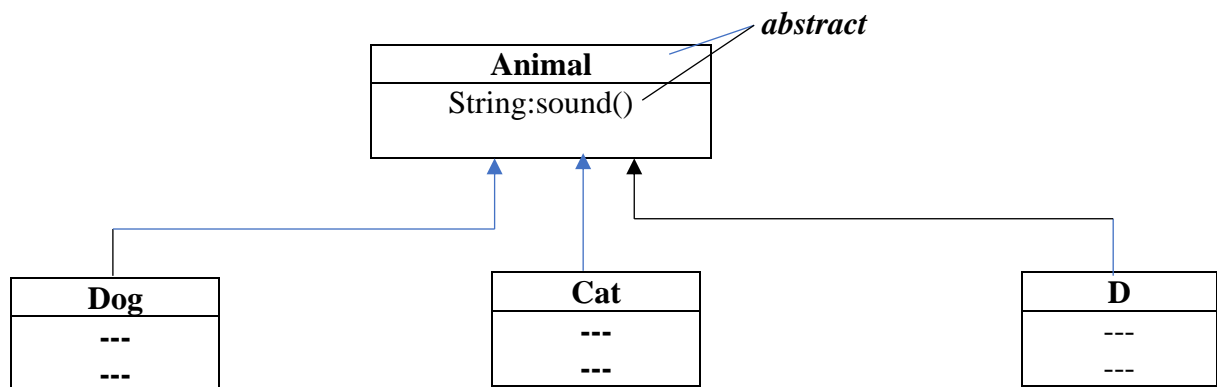
OUTPUT :

```

m1 implement in B
m1 implement in C
m1 implement in D

```

Interview Question :



Program :

```

package edu.jspiders.myfirstproject.AnimalAbstract;

public abstract class Animal
{
    abstract public void sound();
}

package edu.jspiders.myfirstproject.AnimalAbstract;

public class Dog extends Animal
{
    public void sound()
    {
        System.out.println("Bow Bow!!");
    }
}

package edu.jspiders.myfirstproject.AnimalAbstract;

public class Cat extends Animal
{
    public void sound()
    {
        System.out.println("Meow Meow!!");
    }
}

```

```

}
package edu.jspiders.myfirstproject.AnimalAbstract;

public class Lion extends Animal
{
    public void sound()
    {
        System.out.println("Roar!!");
    }
}

package edu.jspiders.myfirstproject.AnimalAbstract;

import java.util.Scanner;

public class MainOfAnimal
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        while(true)
        {
            System.out.println("1.Dog\n2.Cat\n3.Lion");
            System.out.println("Please select your favorite animal");
            int choice = sc.nextInt();
            System.out.println();

            switch(choice)
            {
                case 1: Dog d=new Dog();
                        d.sound();
                        break;
                case 2: Cat c=new Cat();
                        c.sound();
                        break;
                case 3: Lion l=new Lion();
                        l.sound();
                        break;
                default : System.exit(0);
            }
        }
    }
}

```

OUTPUT :

```

1.Dog
2.Cat
3.Lion
Please select your favorite animal
1

```

Bow Bow!!

```

1.Dog
2.Cat
3.Lion
Please select your favorite animal
2

```

Meow Meow!!

1.Dog

2.Cat

3.Lion

Please select your favorite animal

3

Roar!!

1.Dog

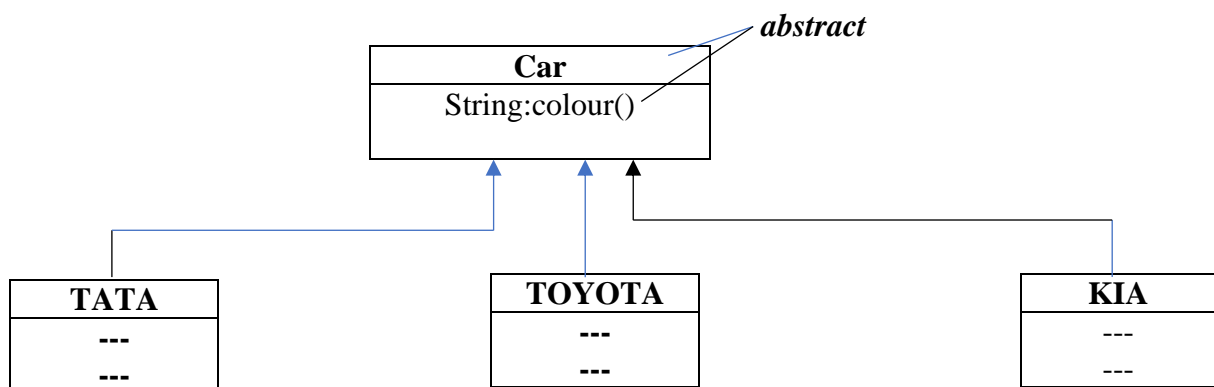
2.Cat

3.Lion

Please select your favorite animal

5

Write a program based on following Class Diagram.



Program :

```
package edu.jspiders.myfirstproject.Colorabstract;

public abstract class Car
{
    abstract public String colour();
}

package edu.jspiders.myfirstproject.Colorabstract;

public class Tata extends Car
{
    public String colour()
    {
        return "Black colour";
    }
}

package edu.jspiders.myfirstproject.Colorabstract;

public class Toyota extends Car
{
    public String colour()
    {
        return "Blue Colour";
    }
}

package edu.jspiders.myfirstproject.Colorabstract;

public class Kia extends Car
{

```

```

        public String colour()
        {
            return "Red Colours";
        }
    }
package edu.jspiders.myfirstproject.Colorabstract;

import java.util.Scanner;

public class MainOfCar
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        while(true)
        {
            System.out.println("1.TATA\n2.TOYOTA\n3.KIA");
            System.out.println("Please select your car");
            int choice = sc.nextInt();
            System.out.println();

            switch(choice)
            {
                case 1:Tata t = new Tata();
                        System.out.println(t.colour());
                        break;
                case 2:Toyota ty = new Toyota();
                        System.out.println(ty.colour());
                        break;
                case 3:Kia k = new Kia();
                        System.out.println(k.colour());
                        break;
                default:System.exit(0);
            }
        }
    }
}

```

OUTPUT :

```

1.TATA
2.TOYOTA
3.KIA
Please select your car
1

```

Black colour

```

1.TATA
2.TOYOTA
3.KIA
Please select your car
2

```

Blue Colour

```

1.TATA
2.TOYOTA
3.KIA
Please select your car
3

```

Red Colours

1.TATA

2.TOYOTA

3.KIA

Please select your car

6

ASSIGNMENT :

1.

Create a project with the name->**JavaBeanProject1**

Create a package with the name

-> LaptopJavaBean

-> LaptopMainJavaBean

under the package LaptopJavaBean

Create a javabean class with the name Laptop with attributes of your choice.

under the package LaptopMainJavaBean

create a MainClass

create 3 objects of the laptop and display their details with help of helper methods.

```
package edu.jspiders.javabeanproject1.LaptopJavaBean;
```

```
public class Laptop
{
    private String name;
    private String model;
    private double price;
    public Laptop(String a, String b, double c)
    {
        name=a;
        model=b;
        price=c;
    }
    public String getName()
    {
        return name;
    }
    public String getModel()
    {
        return model;
    }
    public double getPrice()
    {
        return price;
    }
}
```

```
package edu.jspiders.JavaBeanProject.LaptopMainJavaBean;
```

```
import edu.jspiders.javabeanproject1.LaptopJavaBean.*;
```

```
public class MainClassOfLaptop
```

```
{
    public static void main(String[] args)
    {
        Laptop l1 = new Laptop("Macbook", "Macbook Air-18", 83999);
        Laptop l2 = new Laptop("Dell", "Inspiron 3500", 57499);
        Laptop l3 = new Laptop("Asus", "Notebook", 49999);
    }
}
```



```

        System.out.println("Name\tModel\tPrice");

        System.out.println("=====");
        System.out.println(l1.getName()+"\t"+l1.getModel()+"\t"+l1.getPrice());
        System.out.println(l2.getName()+"\t"+l2.getModel()+"\t"+l2.getPrice());
        System.out.println(l3.getName()+"\t"+l3.getModel()+"\t"+l3.getPrice());
    }

}

```

OUTPUT :

SuperHero Name	Age	No. of Enemy Killed
Iron Man	33	7
Captain America	31	5

Important Conclusion on Abstract Class & Method :

CONCLUSION 1 :

We cannot declare abstract class as **final** because implementation for abstract method of an abstract class will be provided in its sub-class where as a **final** class will not have only sub-class. If at all, we declare an abstract class as final, we will get Compile Time Error as shown below:

```

final abstract class A
{
    abstract public void m1();
}
class B extends A
{
    public void m1()
    {
        ----
    }
}

```

CONCLUSION 2 :

Static methods cannot be declared as abstract because abstract keyword is applicable only for object level methods not class level methods.

```

abstract public void m1()

abstract public static void m2()

```

error

CONCLUSION 3 :

An abstract class can have both concrete methods and abstract methods, the concrete properties present inside abstract class must be static so that they can be accessed with the help of class name.

e.g.

```

package edu.jspiders.myfirstproject.con3;

```

```

public abstract class A
{
    static int x=100;
    static double y = 20.0;
    public static void m1()

```

```

        {
            System.out.println("Hello World!");
        }
        abstract public void m2();
    }
    package edu.jspiders.myfirstproject.con3;

    public class B
    {
        public static void main(String[] args)
        {
            System.out.println(A.x);
            System.out.println(A.y);
            A.m1();
        }
    }
}

```

OUTPUT :

```

100
20.0
Hello World!

```

CONCLUSION 4 :

While providing implementation for abstract method of an abstract class in its sub-class, we should either retain the same visibility of the method or we can increase the visibility but we should not decrease the visibility of the method.

e.g.,

Super class	Super class
-public	-default
sub class	sub class:
-public	-default/protected/public
Super class	Super class:
-protected/public	-private(error, cannot declare abstract method as private)
	Sub class:

CONCLUSION 5 :

Abstract methods can be overridden to concrete method and concrete method can be overridden to abstract method.

```

abstract public class Demo1
{
    abstract public void m1();
}
public class Demo2 extends Demo1
{
    @Override
    public void m1()
    {
        -----
    }
} // abstract method overridden to concrete method.

```

```

public class Demo1
{

```

```

    public void m1()
    {
        -----
    }
}
abstract public class Demo2 extends Demo1
{
    @Override
    abstract public void m1();
} // concrete method overridden to abstract method.

```

CONCLUSION 6 :

When to go for abstract class?

→ Whenever we know the partial implementation of an application, we can make use of abstract class.

When to declare the method as abstract?

→ If we want to perform some tasks but we don't know the implementation of it at that particular time, such method can be declared as abstract. In future, the method will be implemented in sub-class.

CONCLUSION 7 :

Will abstract class allow constructor? Explain.

→ For the abstract method of an abstract class, we will provide the implementation in its sub-class. As per Java rule, every time when we create an object of sub class, its super class constructor has to be called in complete constructor chaining process. Because of this reason, abstract class will have constructor.

Important Note :

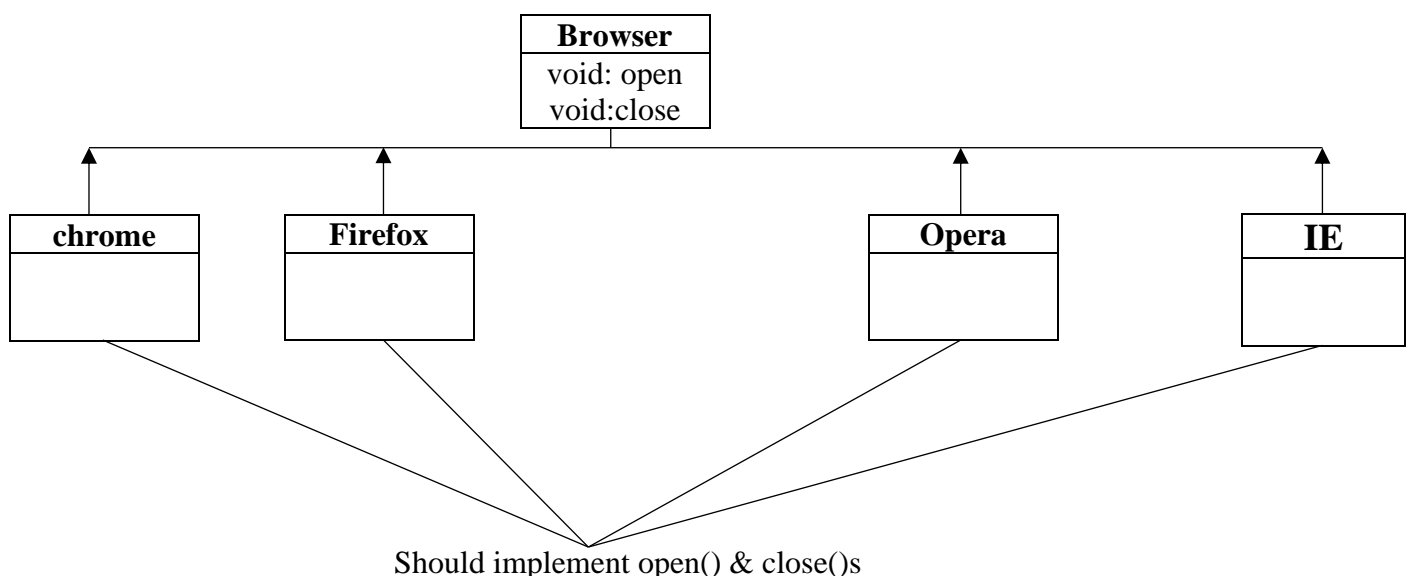
→ A class which contains concrete property can also be declared as abstract class because of:

1. To avoid object creation of class.
2. To ensure all the concrete properties static.

Advantage

- i. To achieve Abstraction

Interview Question:



Passing and Returning of Objects:

Write a method that accepts two integer value and returns nothing.

```
class A
{
    public static void main(String[] args)
    {
        m1(10,20);
    }
    public static void m1(int a, int b)
    {
        System.out.println(a);
        System.out.println(b);
    }
}
```

Write a method that accepts 1 double, 1 boolean value and returns a string value.

```
class A
{
    public static void main(String[] args)
    {
        String S = m1(10.12,true);
        System.out.println(S);
    }
    public static String m1(double x, boolean y)
    {
        System.out.println(x);
        System.out.println(y);
        return "Hello";
    }
}
```

Write a method that accepts 1 char, 1 int value and returns a Boolean value.

```
class A
{
    public static void main(String[] args)
    {
        boolean c = m1('a',10);
        System.out.println(S);
    }
    public static boolean m1(char a, boolean b)
    {
        System.out.println(a);
        System.out.println(b);
        return true;
    }
}
```

Write a method that accepts 1 string, 1 float, 1 string value and returns int value.

```
class A
{
    public static void main(String[] args)
    {
        String x = "Hello";
        float y = 10.2f;
        String z = "World";
        int a = m1(x,y,z);
        System.out.println(a);
    }
}
```

```

public static int m1(String x, float y, String z)
{
    System.out.println(x);
    System.out.println(y);
    System.out.println(z);
    return 10;
}
}

```

Write a method that accepts demo class object and returns nothing.

```

class A
{
    public static void main(String[] args)
    {
        m1(new Demo()); or Demo d = new Demo();
                        m1(d);

        {
            -----
            -----
        }
    }
    public static void m1(Demo d1)
    {
        -----
        -----
    }
}

```

Write a method that accepts 1 sample class object and returns nothing.

```

class A
{
    public static void main(String[] args)
    {
        m1(new Sample()); or Sample s = new Sample();
                        m1(s);

        {
            -----
            -----
        }
    }
    public static void m1(Sample s1)
    {
        -----
        -----
    }
}

```

Write a method that accepts demo class, sample class object and returns nothing.

```

class A
{
    public static void main(String[] args)
    {
        m1(new Demo(),new Sample()); or Demo d = new Demo();
                                     Sample s = new Sample();
                                     m1(d,s);

        {
            -----
            -----
        }
    }
}

```

```

    }
    public static void m1(Demo d, Sample s)
    {
        -----
        -----
    }
}

```

Write a method that accepts A class object and returns B class object.

```

class Demo
{
    public static void main(String[] args)
    {
        B b1=m1(new A()); or A a = new A();
                           B b1=m1(a);

        {
            -----
            -----
        }
    }
    public static B m1(A a1)
    {
        -----
        -----
        return new B();
    }
}

```

Write a method that accepts Sample1,Sample2 class object and returns nothing.

```

class A
{
    public static void main(String[] args)
    {
        m1(new Sample1(),new Sample2()); or Sample1 s1 = new Sample1();
                                           Sample2 s2 = new Sample2();
                                           m1(s1,s2);

        {
            -----
            -----
        }
    }
    public static void m1(Sample s1, Sample s2)
    {
        -----
        -----
    }
}

```

Write a method that accepts X class object and returns Y class object.

```

class Demo
{
    public static void main(String[] args)
    {
        Y y1=m1(new X()); or X x1 = new X();
                           Y y1=m1(x);

        {
            -----
            -----
        }
    }
}

```

```

public static Y m1(X x1)
{
    -----
    -----
    return new Y();
}

```

Write a method that accepts Suresh, Ramesh class object and returns Mahesh class object.

```

class A
{
    public static void main(String[] args)
    {
        Mahesh m=m1(new Suresh(),new Ramesh()); or Suresh s1 = new Suresh();
                                                Ramesh r1 = new Ramesh();
                                                Mahesh m=m1(s1,r1);

        {
            -----
            -----
        }
    }
    public static Mahesh m1(Suresh s1,Ramesh r1)
    {
        -----
        -----
        return new Mahesh();
    }
}

```

Write a method that accepts Dinga, Linga, Singa class object and returns Malinga class object.

```

class A
{
    public static void main(String[] args)
    {
        Malinga m=m1(new Dinga(),new Linga(),new Singa()); or Dinga d = new Dinga();
                                                            Linga l = new Linga();
                                                            Singa s=new Singa();
                                                            Malinga m= m1(d,l,s);

        {
            -----
            -----
        }
    }
    public static Malinga m1(Dinga d, Linga L, Singa s)
    {
        -----
        -----
        return new Malinga();
    }
}

```

Important Note :

1. A method can accept any number of object but it can return only one object.
2. Whenever a method is returning an object, the return type of the method will change to the class type whose object the method is returning.

Interface

An interface is a Java definition block which will contain states and behavior where state refers to data members and behavior refers to member function.

Every variable declared inside a interface is by default **static** & **final** whereas every method declared inside an interface is by default **public** & **abstract**.

Syntax & Example :

interface interface_name

```
{  
    datamembers/variable → static & final  
    member function/method → public & abstract  
}
```

interface A

```
{  
    int a = 10; // by default static & final  
    void m1(); // by default public & abstract  
}
```

SAME

interface A

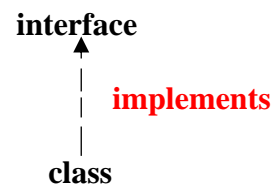
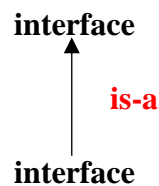
```
{  
    final static int a=10;  
    public abstract v m1();  
}
```

Important Note :

1. A class can extend from another class.

An interface can extend from another interface.

An interface and a class cannot have **Is-a** relationship. They will only have **implements** relationship.



For the abstract properties of an interface, the implementation will be provided in its **implementation class**. As per Java rule, we cannot create an object or instance of an interface.

Example Program :

```
package edu.jspiders.JavaBeanProject.Interface;
```

```
public interface A
```

```
{  
    int a = 10;  
    void m1();  
    void m2();  
}
```

```
public class B implements A
```

```
{  
    public void m1()  
    {  
        System.out.println("in m1");  
    }  
    public void m2()  
    {  
        System.out.println("in m2");  
    }  
}
```



```

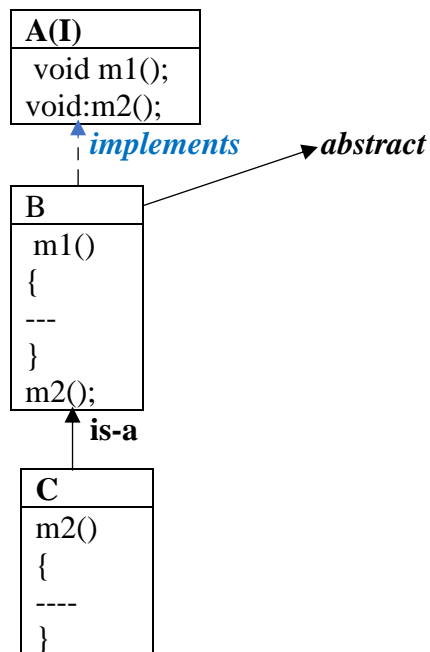
}
public class C
{
    public static void main(String[] args)
    {
        System.out.println(B.a);
        B b1 = new B();
        b1.m1();
        b1.m2();
    }
}

```

OUTPUT :
10
in m1
in m2

If the implementation class fails to provide implementation for all the abstract method of an interface, the implementation class must be declared as **abstract** and the left out implementation should be provided in the further sub-classes.

Example :



```

package edu.jspiders.JavaBeanProject.Interface2;

public interface A
{
    void m1();
    void m2();
}

abstract public class B implements A
{
    public void m1()
    {
        System.out.println("hello!");
    }
}

public class C extends B
{
    public void m2()

```

```

    {
        System.out.println("World!");
    }
}
public class MainClass
{
    public static void main(String[] args)
    {
        C c=new C();
        c.m1();
        c.m2();
    }
}

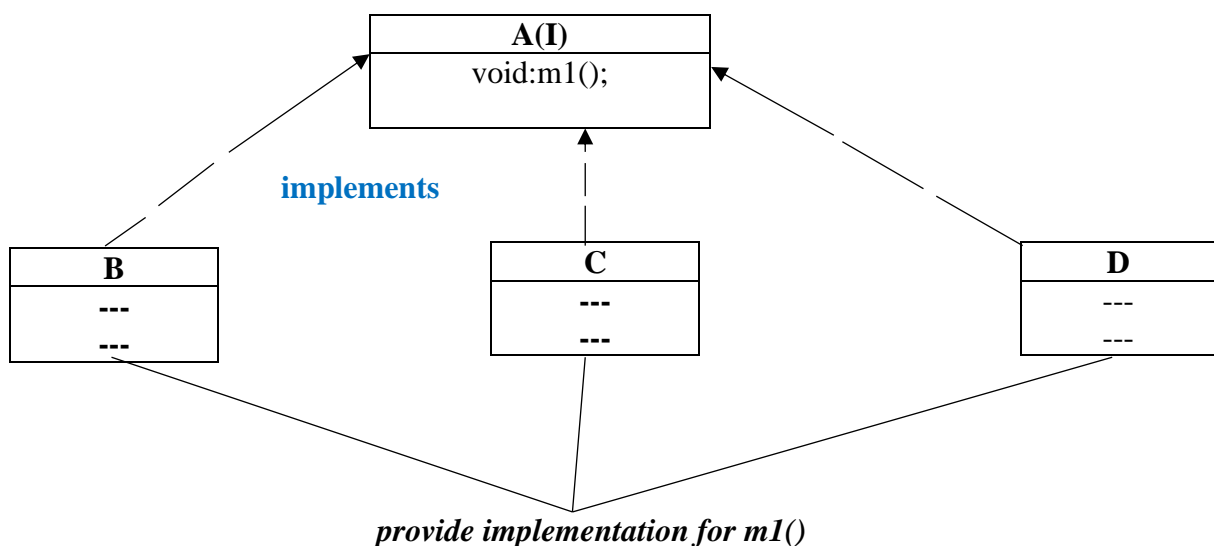
```

OUTPUT :

hello!
World!

For an abstract of an interface, we can provide different implementations in its different implementation classes.

Example:



Example Program:

```

package edu.jspiders.JavanBeanProject.Interface3;

public interface A
{
    void m1();
}
public class B implements A
{
    public void m1()
    {
        System.out.println("Implemented in B");
    }
}
public class C implements A
{
    public void m1()
    {

```

```

        System.out.println("Implemented In C");
    }
}
public class D implements A
{
    public void m1()
    {
        System.out.println("Implemented in D");
    }
}
package edu.jspiders.JavaBeanProject.Interface3;

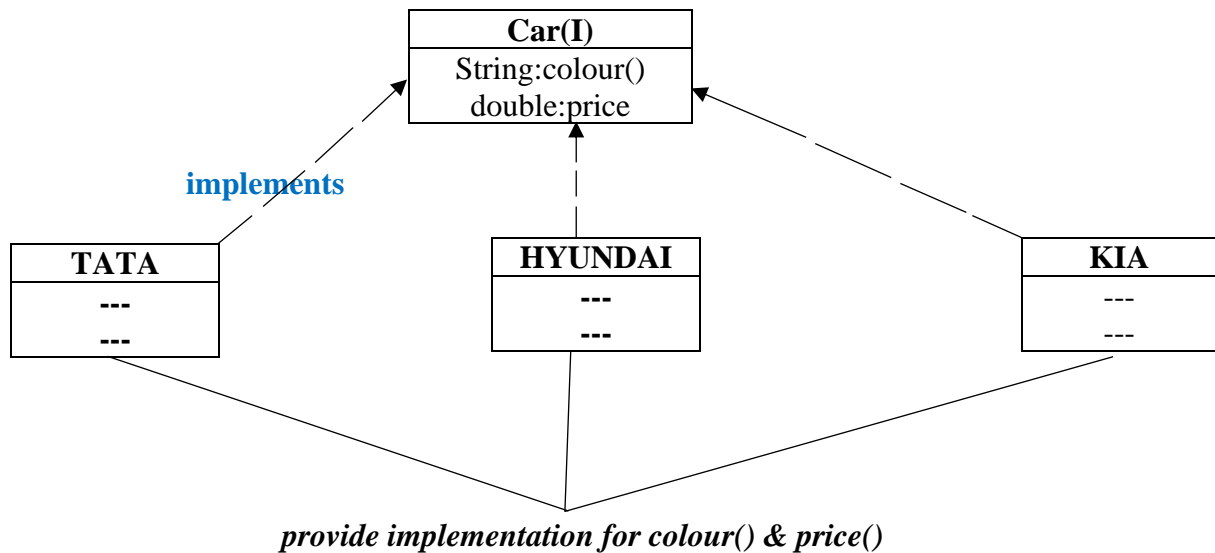
public class MainClass
{
    public static void main(String[] args)
    {
        B b = new B();
        b.m1();
        C c = new C();
        c.m1();
        D d = new D();
        d.m1();
    }
}

```

OUTPUT:

Implemented in B
Implemented In C
Implemented in D

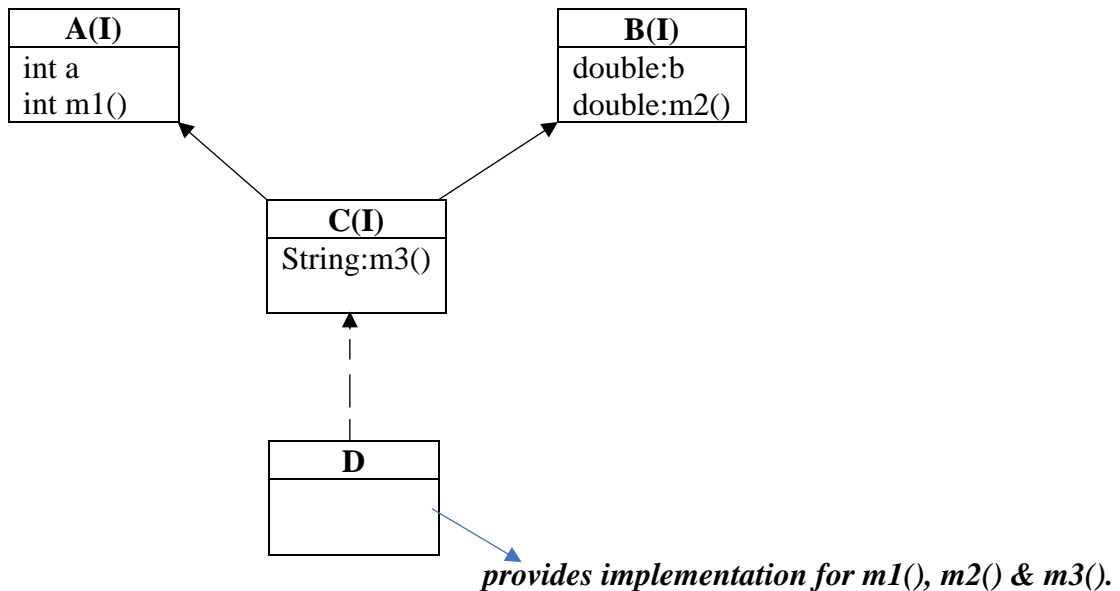
Interview Question :



Important Conclusions on Interface

CONCLUSION 1 :

With respect to classes, multiple inheritance is not allowed in Java but we can achieve multiple inheritance using interfaces. That implies, one sub-interface can have multiple super interfaces. Constructors are not allowed in interface, since constructors are not allowed there is no question of constructor chaining. Therefore, we can achieve multiple inheritance using interfaces.



```
package edu.jspiders.JavaBeanProject.Interface5;
```

```
public interface A
{
```

```
    int a=10;
    int m1();
```

```
}
```

```
public interface B
{
```

```
    double b=10.19;
    double m2();
```

```
}
```

```
public interface C extends A,B
{
```

```
    String m3();
```

```
}
```

```
public class D implements C
{
```

```
    public int m1()
```

```
    {
```

```
        return 20;
```

```
    }
```

```
    public double m2()
```

```
    {
```

```
        return 52.21;
```

```
    }
```

```
    public String m3()
```

```
    {
```

```
        return "Hello Interface";
```

```
    }
```

```
}
```

```

public class MainClass
{
    public static void main(String[] args)
    {
        D d = new D();
        System.out.println(d.m1());
        System.out.println(d.m2());
        System.out.println(d.m3());
        System.out.println(A.a);
    }
}

```

OUTPUT :

```

20
52.21
Hello Interface
10

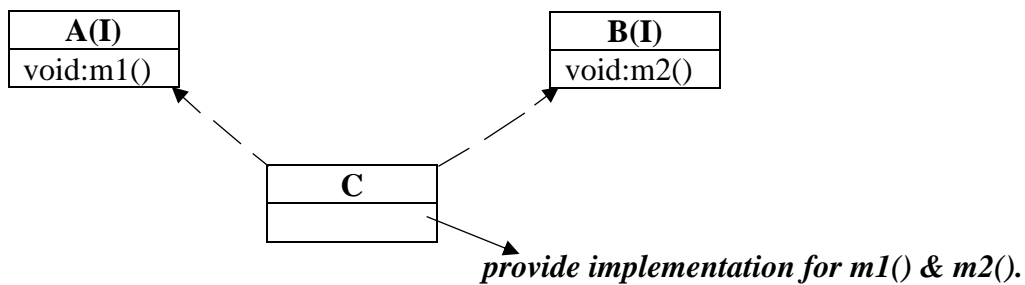
```

CONCLUSION 2 :

A class cannot extend from more than one class but a class can implement more than one interfaces at a time.

While providing implementation for more than one interface at a time, if interfaces contain method with same name and same signature: the implementation class should provide implementation only once as shown in **example 2**.

Example 1 :



C implements A,B

```

class C implements A,B
{
    public void m1()
    {
        -----
    }
    public void m2()
    {
        -----
    }
}

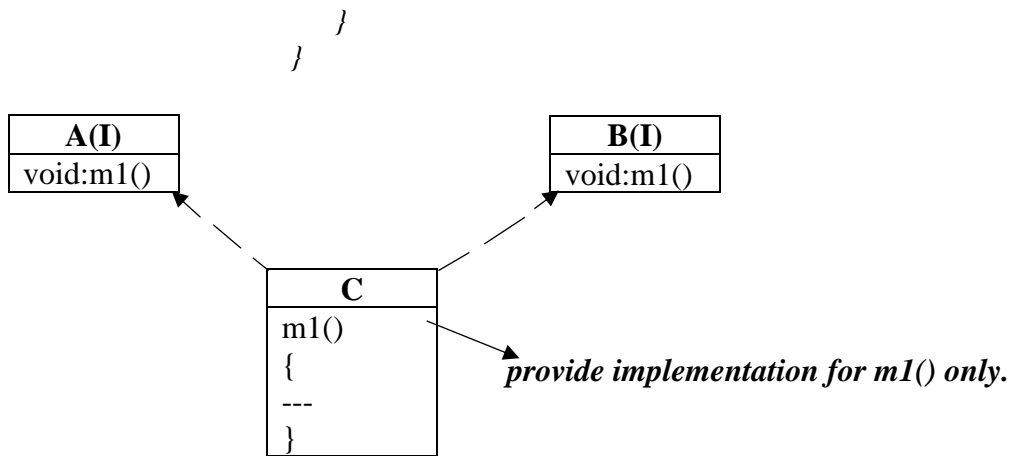
```

Example 2 :

```

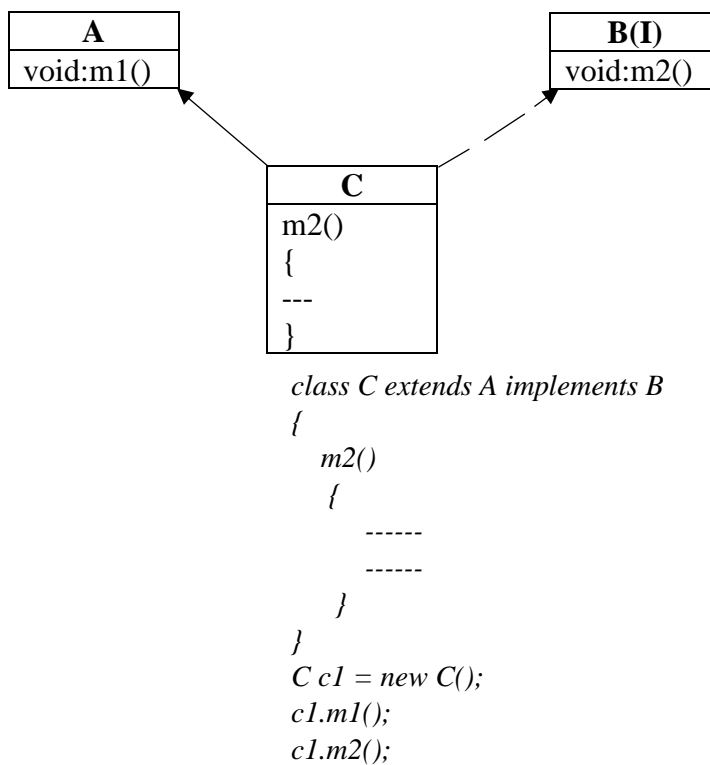
class C implements A,B
{
    public void m1()
    {
        -----
    }
}

```



CONCLUSION 3 :

A class can have both **is-a** relationship and **implements** relationship at the same time.



CONCLUSION 4 :

What are marker interfaces?

→ The interfaces which contain nothing are known as **empty** interfaces. These empty interfaces are technically known as marker interface.

Marker interfaces are not used by the programmer; they are used by the JVM in order to perform some run time validation.

```

interface A
{

}

interface Demo
{

}

E.g.
RandomAccess(I)
Serializable(I)
Cloneable(I)
  
```

used by the JVM

CONCLUSION 5 :

What is functional interface?

→ The interface which contains only one abstract method in it is known as functional interface.

interface A

```
{  
  
    void m1();  
}
```



interface Demo

```
{  
  
    void sort();  
}
```



interface B

```
{  
  
    void m1();  
    void m2();  
}
```



e.g.,

Runnable(I)
Comparable(I)
Callable(I)



**it is in-built interface
written by Java developers.**

CONCLUSION 6 :

Difference between abstract class & interface.

Abstract Class	Interface
1. It is not pure abstract.	1. It is pure abstract.
2. Recommended to use when we know partial implementation of application.	2. Recommended to use when we don't know any implementation for an application.
3. Constructors are allowed.	3. Constructors are not allowed.
4. Static & non-static data members are allowed.	4. By default, all data members are static & final.
5. Both static and non-static member functions are allowed.	5. By default, all member functions are public & abstract(only non-static).
6. A class cannot extend from more than one class.	6. An interface can extend from more than one interface.
7. Implementation is provided in sub-class.	7. Implementation is provided in its implementation class.

Important Note :

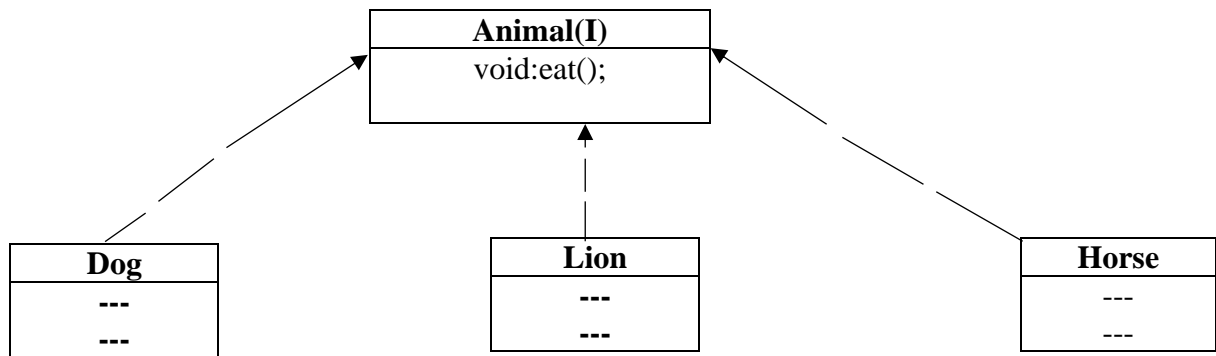
Interface was a pure abstract till version **1.7**, from **1.8** version of Java: Interface is no longer pure abstract. That means from **1.8** version, a new concept was introduced in Java: That is called as Lambda 'λ' expression. As a part of Lambda expression, concrete methods are allowed inside interface.

Advantages of Interface

1. To achieve abstraction.
2. To achieve loose coupling.

INTERVIEW QUESTION :

1.



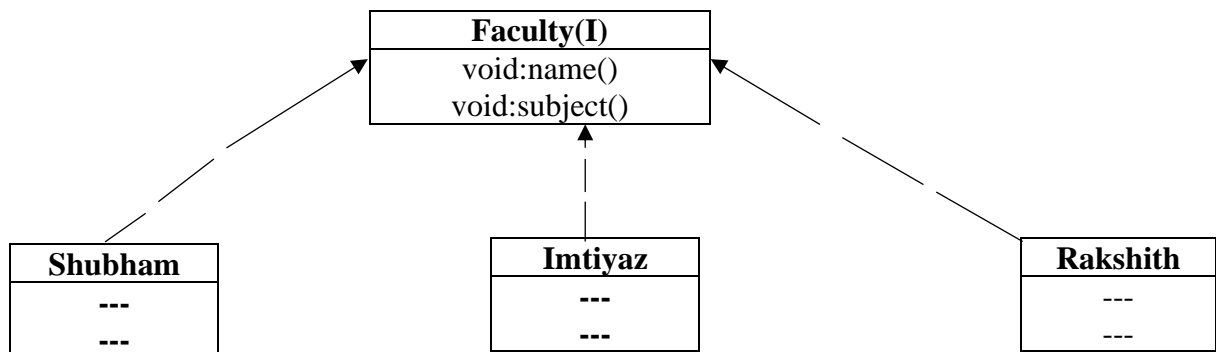
Operations

`void:display(String s)`

MainOfAnimal

`void:main(-)`

2.



Operations

`void:getDetails(String s)`

MainOfFaculty

`void:main(-)`

Program:

```
package edu.jspiders.JavaInterface.Interface8;
public interface Faculty
{
    void name();
    void subject();
}
```



```

}
public class Shubham implements Faculty
{
    public void name()
    {
        System.out.println("Shubham KS");
    }
    public void subject()
    {
        System.out.println("JAVA Core");
    }
}
public class Imtiyaz implements Faculty
{
    public void name()
    {
        System.out.println("Imtiyaz Ali");
    }
    public void subject()
    {
        System.out.println("Web Development");
    }
}
public class Rakshith implements Faculty
{
    public void name()
    {
        System.out.println("Rakshith Sharma");
    }
    public void subject()
    {
        System.out.println("SQL");
    }
}
public class Operation
{
    public void getDetails(String s)
    {
        if(s.equalsIgnoreCase("shubham"))
        {
            Shubham sh = new Shubham();
            sh.name();
            sh.subject();
        }
        else if(s.equalsIgnoreCase("imtiyaz"))
        {
            Imtiyaz i = new Imtiyaz();
            i.name();
            i.subject();
        }
        else if(s.equalsIgnoreCase("rakshith"))
        {
            Rakshith r = new Rakshith();
            r.name();
            r.subject();
        }
        else
        {
            //do nothing
        }
    }
}

```

```

    }
}
import java.util.Scanner;
public class MainOfFaculty
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        Operation o = new Operation();
        while(true)
        {
            System.out.println("1.Shubham\n2.Imtiyaz\n3.Rakshith\nAny other key to EXIT");
            System.out.println("Please select");
            int choice = sc.nextInt();
            System.out.println();
            switch(choice)
            {
                case 1:o.getDetails("Shubham");
                    break;
                case 2:o.getDetails("Imtiyaz");
                    break;
                case 3:o.getDetails("Rakshith");
                    break;
                default: System.out.println("Execution Exited");
                    System.exit(0);
            }
        }
    }
}

```

OUTPUT :

```

Shubham KS
JAVA Core
1.Shubham
2.Imtiyaz
3.Rakshith
Any other key to EXIT
Please select
2

```

```

Imtiyaz Ali
Web Development
1.Shubham
2.Imtiyaz
3.Rakshith
Any other key to EXIT
Please select
3

```

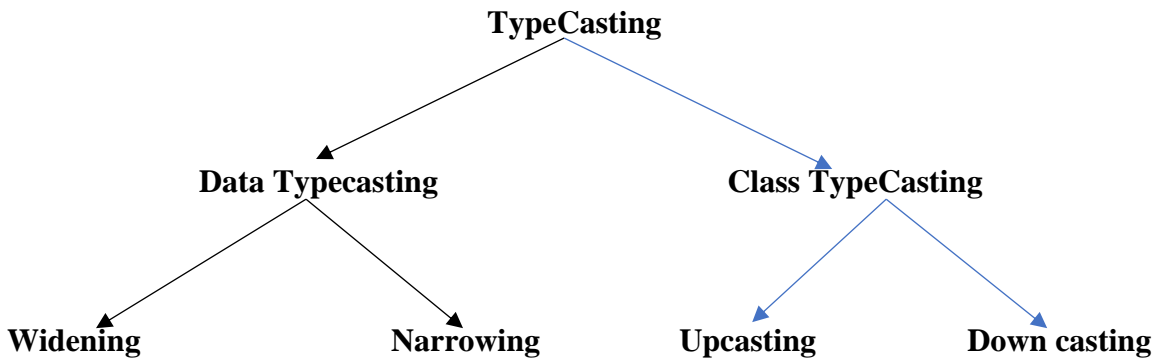
```

Rakshith Sharma
SQL
1.Shubham
2.Imtiyaz
3.Rakshith
Any other key to EXIT
Please select
5
Execution Exited

```

TYPECASTING

The process of converting one type of information to another type is known as **typecasting**. Typecasting can happen only when there is type miss-matching.



Data Typecasting :

The process of converting one type of data to another type is called data typecasting.

Widening : The process of converting smaller type of data to larger type is called widening. Widening can happen both implicitly by JVM and explicitly by a user.

Narrowing : The process of converting larger data type to smaller type is called narrowing. Narrowing only happens when explicitly done by a user. Since narrowing involves data loss, it is not executed by JVM.

type matching

```
=====
int a =123;
double d = 12.12;
char c = 'A';
Boolean b = false;
```

type mismatching

```
=====
double d1=100; //implicit widening
sop(d1); //100.0
```

```
double d2=(double)100; //explicit widening
sop(d2); //100.0
```

narrowing

```
-----
int a1=(int)123.123;
sop(a1); //123
```

```
int a2=(int)213.1;
sop(a2); 213
```



**it should be done
explicitly by
a user only.**

Example program :

```
package edu.jspiders.MyFirstProject.Typecasting;

public class Demo1
{
    public static void main(String[] args)
    {
        //implicit widening
        double d1=100;
        System.out.println("d1 = "+d1);
        int a1 = 'A';
        System.out.println("a1 = "+a1);
        double p1 = 'A';
        System.out.println("p1 = "+p1);

        //explicit widening
        double d2 =(double)123;
        System.out.println("d2 = "+d2);
        int a2 = (int)'a';
        System.out.println("a2 = "+a2);

        //explicit narrowing
        int x1 = (int)123.123;
        System.out.println("x1 = "+x1);
        int x2 = (int)12.12f;
        System.out.println("x2 = "+x2);
        char ch=(char)65.12;
        System.out.println("ch = "+ch);
    }
}
```

OUTPUT :

```
d1 = 100.0
a1 = 65
p1 = 65.0
d2 = 123.0
a2 = 97
x1 = 123
x2 = 12
ch = A
```

Class Typecasting :

The process of converting one type of class information to another class type is known as **class typecasting**.

RULES TO BE FOLLOWED TO ACHIEVE CLASS TYPECASTING :

1. **Is-a** relationship is mandatory between the classes.
2. The class which we are trying to convert should contain the properties of the class to which we are trying to convert it.

Up casting : The process of converting sub-class information type to super-class type is known as upcasting. Upcasting can be done both implicitly by JVM and explicitly by a user.

Down casting : The process of converting super-class type information to sub-class type is known as down casting. Since super class will not contain the properties of sub-class, down casting can only be done explicitly by a user.

Example :

```
class A
{
    -----
    -----
}
class B extends A
{
    -----
    -----
}

class C
{
    p s v m(-)
    {
        A a1 = new A(); //type matching
        B b1 = new B();
        //upcasting
        A a2 = new B(); //implicit
        A a3 = (A)new B(); //explicit
        //down casting
        B b2 = (B) new A(); //explicit
    }
}
```

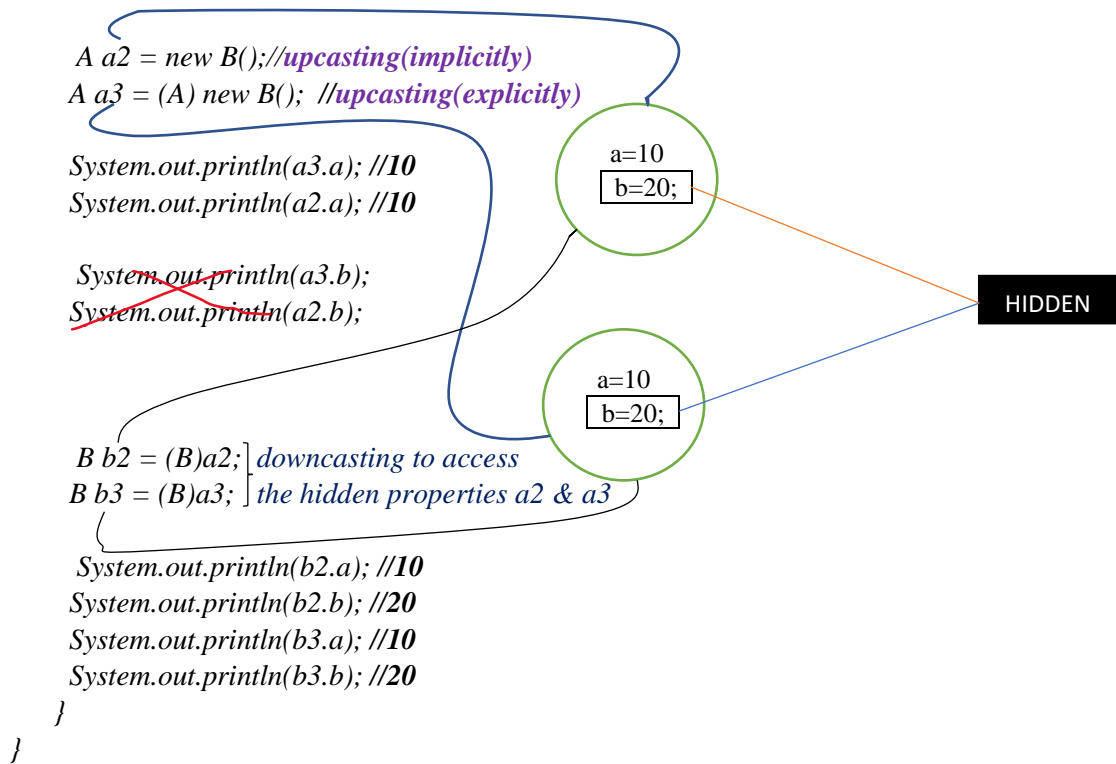
Important Note :

1. If a sub-class object is referred by super-class reference variable, it is known as **upcasting**.
2. If a super-class object is referred by sub-class reference variable, it is known as **downcasting**.
3. Whenever we perform upcasting i.e., if sub-class object is referred by super-class reference variable, the sub-class properties present in the object will be hidden, we cannot access the sub-class properties using a super class reference variable from an upcasted object. If we want to access the hidden properties from the object, the upcasted object must be downcasted as shown in the below example :

```
class A
{
    int a = 10;
}
class B extends A
{
    int b = 20;
}
class C
{
    p s v m(-)
    {
        A a1 = new A();
        System.out.println(a1.a); //10

        B b1 = new B();
        System.out.println(b1.a); //10
        System.out.println(b1.b); //20
    }
}
```

//TYPE MATCHING



4. Downcasting should be done only on upcasted object. If at all we try to perform downcasting without performing upcasting, we will get exception during runtime. (**ClassCastException**)

package edu.jspiders.MyFirstProject.Typecasting1;

public class A

```
{
    int a=10;
}
```

public class B extends A

```
{
    int b=20;
}
```

public class MainClass

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

```
        A a1 = new B(); //upcasting
        System.out.println(a1.a);
```

```
        B b1 =(B)a1; //downcasting for the upcasted object
        System.out.println(b1.a);
        System.out.println(b1.b);
```

```
        B b2=(B)new A(); //direct downcasting without performing any upcasting
```

```
    }
}
```

OUTPUT :

```
10
10
20
```

Exception in thread "main" [java.lang.ClassCastException:](#)

edu.jspiders.MyFirstProject.Typecasting1.A cannot be cast to
edu.jspiders.MyFirstProject.Typecasting1.B

at

edu.jspiders.MyFirstProject.Typecasting1.MainClass.main([MainClass.java:14](#))

Advantage of Typecasting

1. To achieve generalization.

instanceof keyword :

It is a keyword which is used to check if an object contains the properties of the specified class or not. It will return either **true** or **false**; if the object contains the properties of specified class, it will return true else it will return false.

```
class A
{
    -----
    -----
}
class B extends A
{
    -----
    -----
}
class C extends B
{
    -----
    -----
}
class D
{
    p s v m(-)
    {
        A a1 = new A();
        System.out.println(a1 instanceof object); //true
        System.out.println(a1 instanceof A); //true
        System.out.println(a1 instanceof B); //false
        System.out.println(a1 instanceof C); //false

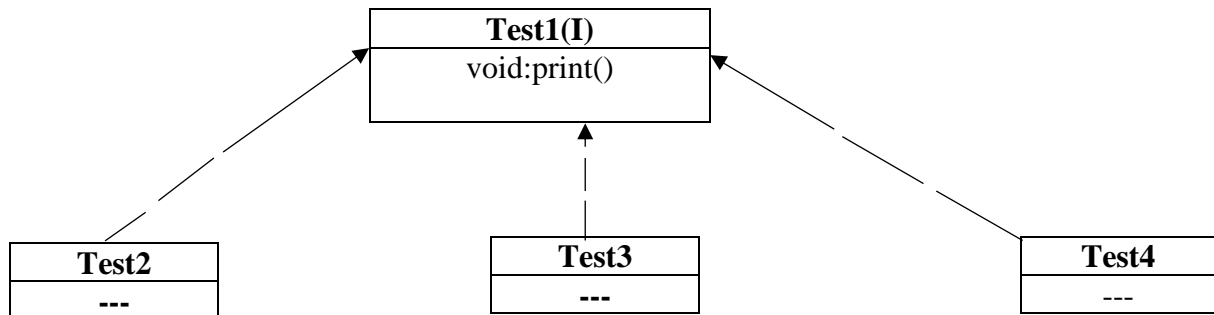
        B b1 = new B();
        System.out.println(b1 instanceof object); //true
        System.out.println(b1 instanceof A); //true
        System.out.println(b1 instanceof B); //true
        System.out.println(b1 instanceof C); //false

        C c1 = new C();
        System.out.println(c1 instanceof object); //true
        System.out.println(c1 instanceof A); //true
        System.out.println(c1 instanceof B); //true
        System.out.println(c1 instanceof C); //true
    }
}
```

Without implementing class typecasting in the program

If we do not use class typecasting concept in the program, we will end up creating multiple overloaded specialized method as shown in the below scenario. Such way of programming approach is known as **specialization** which is highly **not recommended** as per **IT Standard**.

The method which accepts only specific class objects is known as **specialized method**.

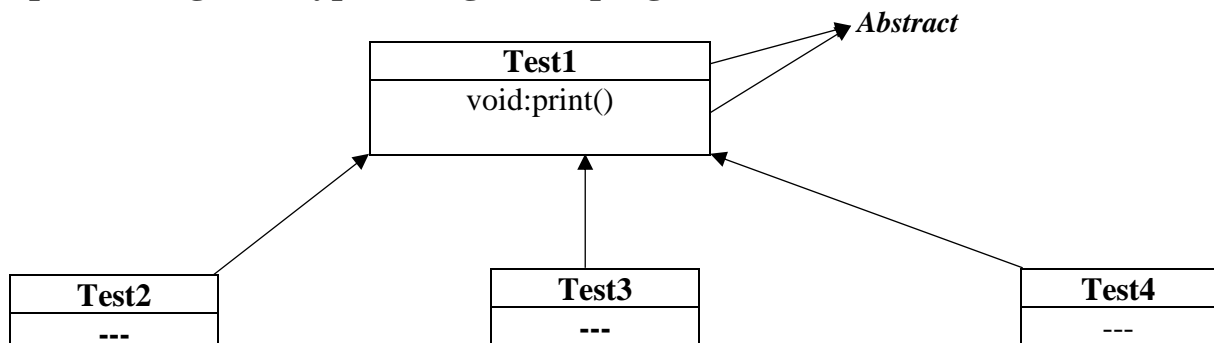


```
class TestOperation
{
    public void display(Test2 t2)
    {
        t2.print();
    }
    public void display(Test3 t3)
    {
        t3.print();
    }
    public void display(Test4 t4)
    {
        t4.print();
    }
}

class MainOfTest
{
    p s v m(-)
    {
        TestOperation to = new TestOperation();
        to.display(new Test2());
        to.display(new Test3());
        to.display(new Test4());
    }
}
```

A bracket on the right side of the `display` methods in `TestOperation` is labeled **Specialization method**.

With implementing class typecasting in the program



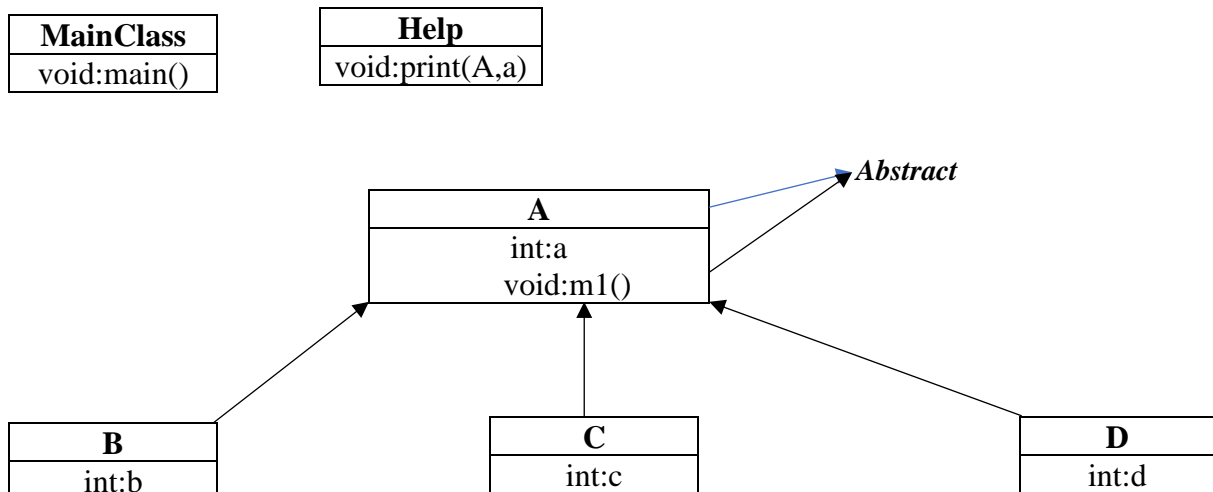
Using class typecasting concept, instead of writing multiple specialized methods, we can write a single generalized method. Such programming approach is known as **generalization**. It is recommended over **specialization** as per **IT Standard**.

A method which can accept any type of class object is known as **generalized method**.

```
class TestOperation
{
    public void display(Test1 t1)
    {
        if(t1 instanceof test2)
        {
            Test t2 = (Test2) t1; //Downcasting
            t2.print();
        }
        else if(t1 instanceof test3)
        {
            Test t3 = (Test3) t1; //Downcasting
            t3.print();
        }
        else if(t1 instanceof test4)
        {
            Test t4 = (Test4) t1; //Downcasting
            t4.print();
        }
    }
}

class MainOfTest
{
    p s v m(-)
    {
        TestOperation to = new TestOperation();
        to.display(new Test2()); //upcasting with test1 when true
        to.display(new Test3()); //upcasting with test1 when true
        to.display(new Test4()); //upcasting with test1 when true
    }
}
```

Example 2 :



Program :

```
package edu.jspiders.JavaBeanProject.typecasting1;

public abstract class A
{
    int a=30;
    abstract void m1();
}

public class B extends A
{
    int b=10;
    @Override
    public void m1()
    {
        System.out.println("in m1 of B");
    }
}

public class C extends A
{
    int c=20;
    @Override
    public void m1()
    {
        System.out.println("in m1 of C");
    }
}

public class D extends A
{
    int d=40;
    @Override
    public void m1()
    {
        System.out.println("in m1 of D");
    }
}

public class Help
{
    public void print(A a1)
    {
        if(a1 instanceof B)
        {
            B b1=(B)a1;
            System.out.println(b1.a);
            System.out.println(b1.b);
            b1.m1();
        }
        else if(a1 instanceof C)
        {
            C c1=(C)a1;
            System.out.println(c1.a);
            System.out.println(c1.c);
            c1.m1();
        }
        else if(a1 instanceof D)
        {
            D d1=(D)a1;
            System.out.println(d1.a);
            System.out.println(d1.d);
            d1.m1();
        }
    }
}
```

```

        else
        {
            //do nothing
        }
    }
}
import java.util.Scanner;

public class MainClass
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        Help h = new Help();
        while(true)
        {
            System.out.println("\n1.B\n2.C\n3.D\n");
            System.out.println("Please select");
            int choice = sc.nextInt();
            switch(choice)
            {
                case 1 : h.print(new B());
                        break;
                case 2 : h.print(new C());
                        break;
                case 3 : h.print(new D());
                        break;
                default: System.out.println("Execution aborted!!");
                        System.exit(0);
            }
        }
    }
}

```

OUTPUT :

1.B
2.C
3.D

Please select

1
30
10
in m1 of B

1.B
2.C
3.D

Please select

2
30
20
in m1 of C

1.B
2.C
3.D

Please select

3

30

40

in m1 of D

1.B

2.C

3.D

Please select

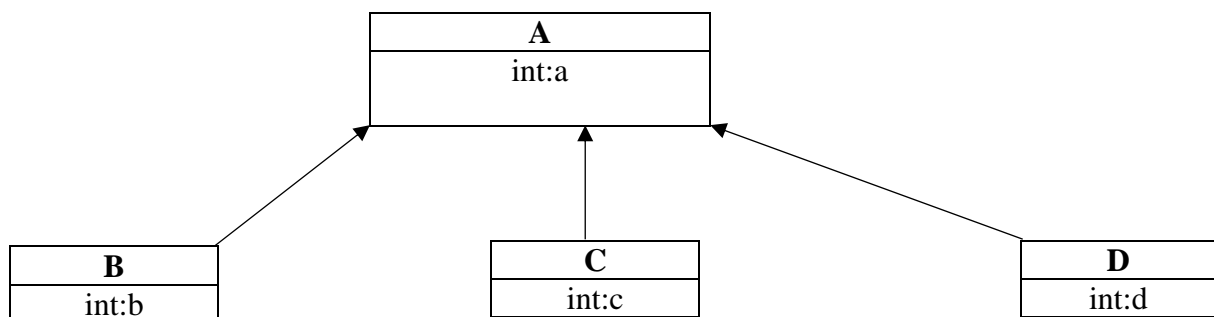
4

Execution aborted!!

Tight Coupling VS Loose Coupling

1. If an implementation class object is referred using class type reference variable, it is known as **tight coupling**.

Example:



```
class Operations
{
    public void print(A a1)
    {
        if(a1 instanceof B)
        {
            B b1 = (B)a1;
            System.out.println(b1.a);
            System.out.println(b1.b);
        }
        else if(a1 instanceof C)
        {
            C c1 = (C)a1;
            System.out.println(c1.a);
            System.out.println(c1.b);
        }
        else if(a1 instanceof D)
        {
            D d1 = (D)a1;
            System.out.println(d1.a);
        }
    }
}
```

```

        System.out.println(d1.b);
    }
}
}
class MainClass
{
    public static void main(String[] args)
    {
        Operations o = new Operations();
        o.print(new B());
        o.print(new C());
        o.print(new D());
    }
}

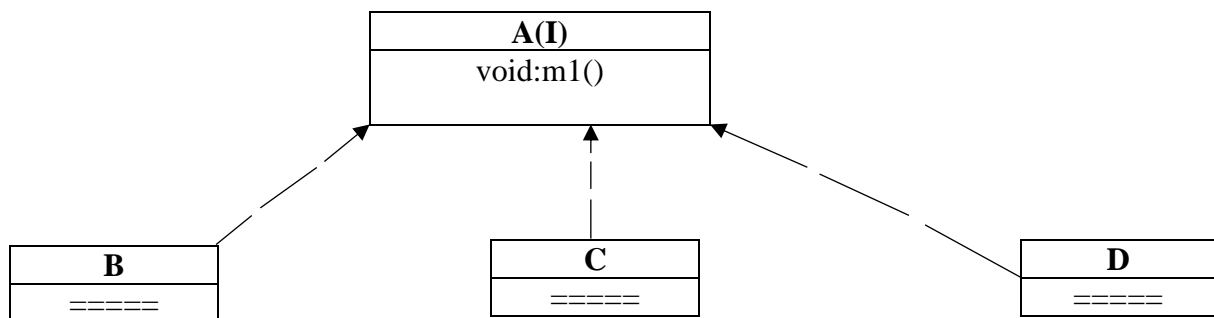
```

As per the IT rules, allowing tight coupling concept in the program is not recommended.

2. An implementation class object being referred by interface type reference variable is known as **loose coupling**.

Loose coupling approach is recommended for coding.

Example



```

class Operations
{
    public void print(A a1)
    {
        if(a1 instanceof B)
        {
            a1.m1();
        }
        else if(a1 instanceof C)
        {
            a1.m1();
        }
        else if(a1 instanceof D)
        {
            a1.m1();
        }
    }
}
class MainClass
{

```

```

public static void main(String[] args)
{
    Operations o = new Operations();
    o.print(new B());
    o.print(new C());
    o.print(new D());
}
}

```

ABSTRACTION :

The process of hiding the implementation and giving only functionality to the user is known as **abstraction**.

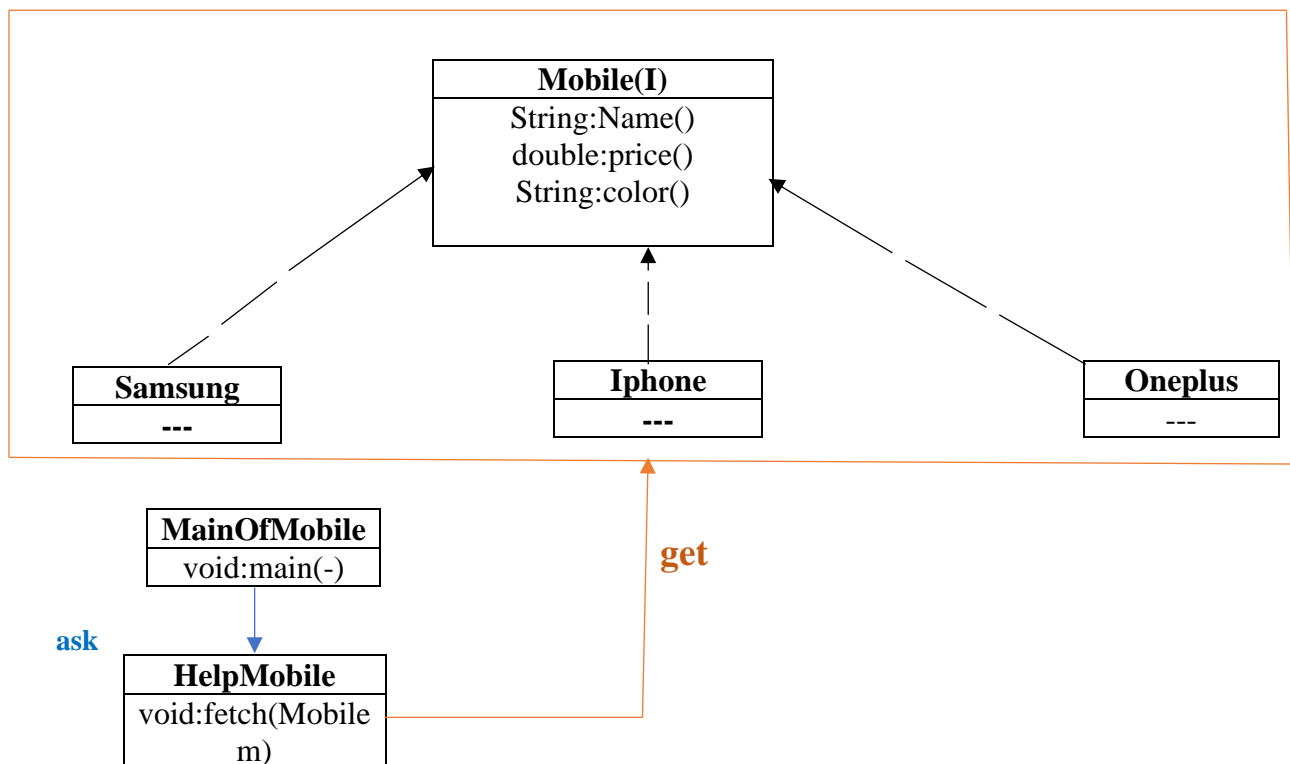
In Java, abstraction can be achieved with the help of **abstract class** and **interface** but interface is more recommended when compared to abstract class because using interface, we can achieve loose coupling.

Real World Examples for Abstraction :

1. Working of a switch board.
2. Working of a mobile.
3. Working of human organs.

Steps to achieve abstraction in Java :

- i. Identify all the common properties and declare them inside an interface.
- ii. Provide different implementations in different implementation classes for all the abstract properties of the interface.
- iii. With the help of interface type reference variable, access the appropriate implementation as requested by the user.



Program :

```
package edu.jspiders.abstraction.mobile;

public interface Mobile
{
    String Name();
    double Price();
    String Color();
}

public class Samsung implements Mobile
{
    public String Name()
    {
        return "Samsung m50";
    }
    public double Price()
    {
        return 24999;
    }
    public String Color()
    {
        return "Sky Blue";
    }
}

public class Oneplus implements Mobile
{
    public String Name()
    {
        return "Oneplus 7T pro";
    }
    public double Price()
    {
        return 38499;
    }
    public String Color()
    {
        return "Black";
    }
}

public class Iphone implements Mobile
{
    public String Name()
    {
        return "Iphone 12 pro";
    }
    public double Price()
    {
        return 124999;
    }
    public String Color()
    {
        return "Dark Green";
    }
}

public class MobileHelp
{
    public void fetch(Mobile m)
    {
        if(m instanceof Samsung)
        {

```

```

        System.out.println("\nMobile Details are");
        System.out.println("=====");
        System.out.println("Name : "+m.Name());
        System.out.println("Price : "+m.Price());
        System.out.println("Color : " +m.Color());
    }
    else if(m instanceof Iphone)
    {
        System.out.println("\nMobile Details are");
        System.out.println("=====");
        System.out.println("Name : "+m.Name());
        System.out.println("Price : "+m.Price());
        System.out.println("Color : " +m.Color());
    }
    else if(m instanceof Oneplus)
    {
        System.out.println("\nMobile Details are");
        System.out.println("=====");
        System.out.println("Name : "+m.Name());
        System.out.println("Price : "+m.Price());
        System.out.println("Color : " +m.Color());
    }
    else
    {
        //do nothing
    }
}

import java.util.Scanner;

public class MainOfMobile
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        MobileHelp h = new MobileHelp();
        while(true)
        {
            System.out.println("\nPlease select any one.");
            System.out.println("=====");
            System.out.println("1.Samsung\n2.Iphone\n3.Oneplus");
            int choice = sc.nextInt();
            switch(choice)
            {
                case 1:h.fetch(new Samsung());
                        break;
                case 2:h.fetch(new Iphone());
                        break;
                case 3:h.fetch(new Oneplus());
                        break;
                default: System.out.println("Execution aborted");
                        System.exit(0);
            }
        }
    }
}

```


OUTPUT :

Please select any one.

=====

- 1.Samsung
- 2.Iphone
- 3.Oneplus

1

Mobile Details are

=====

Name : Samsung m50

Price : 24999.0

Color : Sky Blue

Please select any one.

=====

- 1.Samsung
- 2.Iphone
- 3.Oneplus

2

Mobile Details are

=====

Name : Iphone 12 pro

Price : 124999.0

Color : Dark Green

Please select any one.

=====

- 1.Samsung
- 2.Iphone
- 3.Oneplus

3

Mobile Details are

=====

Name : Oneplus 7T pro

Price : 38499.0

Color : Black

Please select any one.

=====

- 1.Samsung
- 2.Iphone
- 3.Oneplus

4

Execution aborted

Private Constructor:

A constructor which is declared using **private** access modifier is known as private constructor.

If a class contains private constructor, the object creation of that class will be restricted from anywhere outside the class. If we want to get object of a class that contains private constructor, it is possible using **helper** method.

The helper method must be static so that it can be called anywhere by using class name.

Example 1 :

```
public class A
{
    private A()
    {
        System.out.println("Hello");
    }
    public static A get()
    {
        return new A();
    }
}
public class B
{
    public static void main(String[] args)
    {
        A a1 = A.get();
        A a2 = A.get();
        A a3 = A.get();
    }
}
```

OUTPUT:

Hello
Hello
Hello

Example 2:

```
public class Test
{
    int x;
    int y;
    private Test(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public static Test getObject(int x, int y)
    {
        return new Test(x, y);
    }
}
public class MainOfTest
{
    public static void main(String[] args)
    {
        Test t1 = Test.getObject(10, 20);
        System.out.println(t1.x);
        System.out.println(t1.y);
    }
}
```

OUTPUT :

10
20

Important Note :

If we want to restrict the object creation of a class anywhere outside the class, we can make use of **private constructor**.

SINGLETON CLASS :

A class where only one object can be created throughout the program is known as **singleton class**. Using singleton class, we can avoid unnecessary object creation & use the heap memory efficiently. A singleton class can be created by using a **private constructor** & a **helper** method. Along with Java Bean Class, singleton class is a Java Design Pattern.

Example 1:

```
public class A
{
    static A a;
    private A()
    {
        System.out.print("Hello");
    }
    public static A get()
    {
        if(a==null)
        {
            a=new A();
        }
        return a;
    }
}

public class B
{
    public static void main(String[] args)
    {
        A a1 = A.get();
        A a2 = A.get();
        A a3 = A.get();
    }
}
```

OUTPUT :

Hello

Example 2:

```
public class Test
{
    static Test t1;
    private Test()
    {
        System.out.println("hi");
    }
    public static Test getObject()
    {
        if(t1==null)
        {
            t1=new Test();
        }
    }
}
```

```

    }
    return t1;
}
}
public class MainOfTest
{
    public static void main(String[] args)
    {
        Test t1 = Test.getObject();
        Test t2 = Test.getObject();
        Test t3 = Test.getObject();
        Object obj = new Object();
        System.out.println(t1==t2);
        System.out.println(obj==t2);
        System.out.println(t1==t3);
    }
}

```

OUTPUT :

```

hi
true
false
true

```

POLYMORPHISM

An object showing different behaviour in its different stages of life is known as polymorphism.

Real World Example :

Life cycle of a butterfly : egg→caterpillar→Lava→Butterfly

Behaviour of a student in different places on different occasion.

Types of Polymorphism in Java

- a. Static Polymorphism
- b. Dynamic Polymorphism

a. Static Polymorphism :-

It is also known as **Compile Time Polymorphism**.

The binding of method declaration with method definition will take place by the same compiler during compilation time. Since it takes place during compilation, hence called as compile time polymorphism.

Example :-

MethodOverLoading & Constructor OverLoading

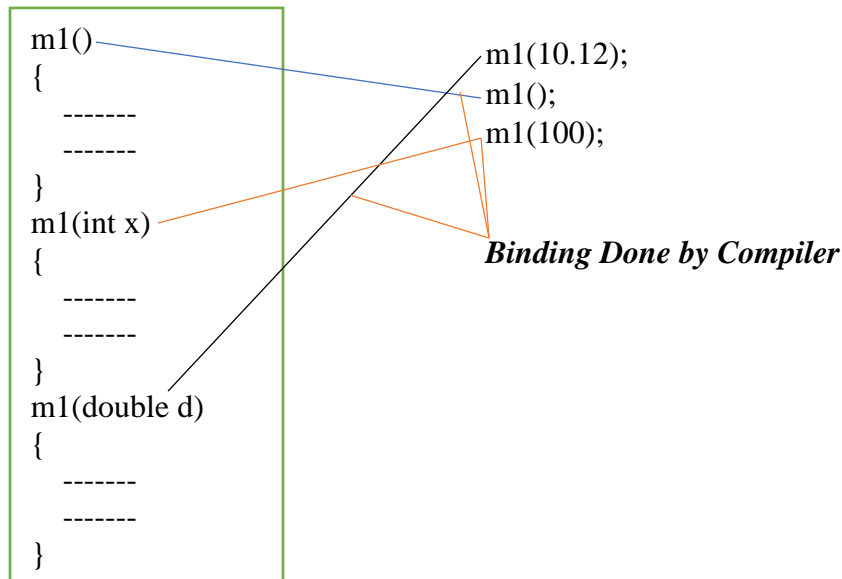
b. Dynamic Polymorphism :-

The binding of method declaration with method definition will take place by JVM during the execution based on the **run time object**. Since the binding is happen during execution, it is known as **run time polymorphism**.

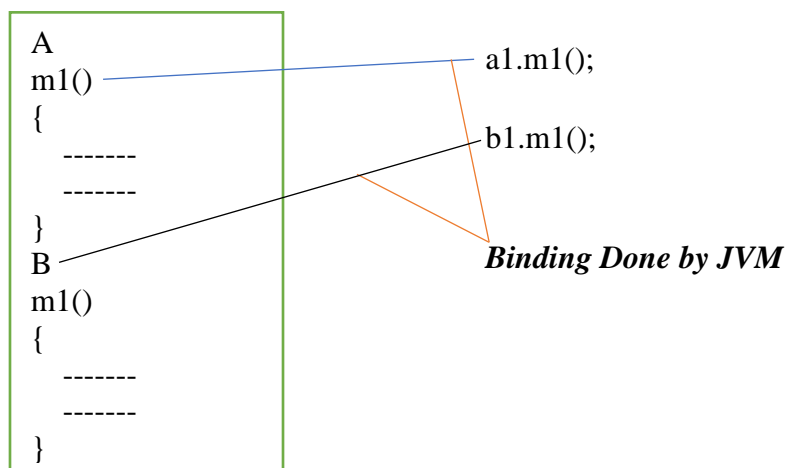
Example :

Method Overriding

Static/Compile Time Polymorphism



Run Time Polymorphism



ASSIGNMENT

Q. Create a Vehicle super class with the following attributes.

- String : Vehicle_Name
- double : Vehicle_Distance_Covered
- double : Fuel_Capacity

Create Bike, Car and Truck sub classes of Vehicle class.

Create a Transport class that contains only the main().

Initialize the attributes for the Bike , Car and Truck using augmented Constructor.

The classes Bike, Car & Truck must have the following

1.Argumented constructor which will initialize the attributes.

2.double vehicleMileage(double Vehicle_Distance_Covered,double Fuel_Capacity)

It will calculate the mileage of the vehicle using the formula
Vehicle_Distance_Covered/Fuel_Capacity.
And return the result back to the caller

3.display()

It will display the details of the Bike in the following format

Vehicle_Name

Vehicle_Distance_Covered

Fuel_Capacity

Mileage (by calling vehicleMileage(double,double))

In the main method

Create an instance of class Bike and initialize all the data members with proper values.

Create an instance of class Car and initialize all the data members with proper values.

Create an instance of class Truck and initialize all the data members with proper values

. Call Display method of all vehicles and display their details.

Program :

```
package edu.jsp.Inheritance.Assignment;
```

```
public class Vehicle
```

```
{
```

```
    String Vehicle_Name;
```

```
    double Vehicle_Distance_Covered;
```

```
    double Fuel_Capacity;
```

```
}
```

```
public class Bike extends Vehicle
```

```
{
```

```
    public Bike(String Vehicle_Name,double Vehicle_Distance_Covered,double Fuel_Capacity)
```

```
    {
```

```
        super.Vehicle_Name= Vehicle_Name;
```

```
        super.Vehicle_Distance_Covered= Vehicle_Distance_Covered;
```

```
        super.Fuel_Capacity = Fuel_Capacity;
```

```
    }
```

```
    public double VehicleMileage(double Vehicle_Distance_Covered, double Fuel_Capacity)
```

```
    {
```

```
        return Vehicle_Distance_Covered/Fuel_Capacity;
```

```
    }
```

```
    public void Display()
```

```
    {
```

```
        System.out.println("Vehicle Name : "+super.Vehicle_Name);
```

```
        System.out.println("Vehicle Distance Covered : "+super.Vehicle_Distance_Covered);
```

```
        System.out.println("Fuel Capacity : "+super.Fuel_Capacity);
```

```
        System.out.println("Vehicle Mileage"+VehicleMileage(Vehicle_Distance_Covered,Fuel_Capacity));
```

```
    }
```

```
}
```

```
public class Car extends Vehicle
```

```
{
```

```
    public Car(String Vehicle_Name,double Vehicle_Distance_Covered,double Fuel_Capacity)
```

```
    {
```

```
        super.Vehicle_Name= Vehicle_Name;
```

```
        super.Vehicle_Distance_Covered= Vehicle_Distance_Covered;
```

```
        super.Fuel_Capacity = Fuel_Capacity;
```

```
    }
```

```
    public double VehicleMileage(double Vehicle_Distance_Covered, double Fuel_Capacity)
```

```
    {
```

```
        return Vehicle_Distance_Covered/Fuel_Capacity;
```

```
    }
```

```

    public void Display()
    {
        System.out.println("Vehicle Name : "+super.Vehicle_Name);
        System.out.println("Vehicle Distance Covered : "+super.Vehicle_Distance_Covered);
        System.out.println("Fuel Capacity : "+super.Fuel_Capacity);
        System.out.println("Vehicle Mileage "+VehicleMileage(Vehicle_Distance_Covered,Fuel_Capacity));
    }
}
public class Truck extends Vehicle
{
    public Truck(String Vehicle_Name,double Vehicle_Distance_Covered,double Fuel_Capacity)
    {
        super.Vehicle_Name= Vehicle_Name;
        super.Vehicle_Distance_Covered= Vehicle_Distance_Covered;
        super.Fuel_Capacity = Fuel_Capacity;
    }
    public double VehicleMileage(double Vehicle_Distance_Covered, double Fuel_Capacity)
    {
        return Vehicle_Distance_Covered/Fuel_Capacity;
    }
    public void Display()
    {
        System.out.println("Vehicle Name : "+super.Vehicle_Name);
        System.out.println("Vehicle Distance Covered : "+super.Vehicle_Distance_Covered);
        System.out.println("Fuel Capacity : "+super.Fuel_Capacity);
        System.out.println("Vehicle Mileage:"+VehicleMileage(Vehicle_Distance_Covered,Fuel_Capacity));
    }
}
public class Transport
{
    public static void main(String[] args)
    {
        Bike d = new Bike("Pulsar 220",300,5);
        System.out.println("OUTPUT");
        System.out.println("=====");
        d.Display();

        Car c = new Car("Hyundai Creta",1400,18);
        System.out.println("OUTPUT");
        System.out.println("=====");
        c.Display();

        Truck t = new Truck("Ashok Leyland",30000,500);
        System.out.println("OUTPUT");
        System.out.println("=====");
        t.Display();
    }
}

```

OUTPUT :

OUTPUT

```

=====
Vehicle Name : Pulsar 220
Vehicle Distance Covered : 300.0
Fuel Capacity : 5.0
Vehicle Mileage : 60.0

```

OUTPUT

=====

Vehicle Name : Hyundai Creta
Vehicle Distance Covered : 1400.0
Fuel Capacity : 18.0
Vehicle Mileage : 77.77777777777777

OUTPUT

=====

Vehicle Name : Ashok Leyland
Vehicle Distance Covered : 30000.0
Fuel Capacity : 500.0
Vehicle Mileage : 60.0