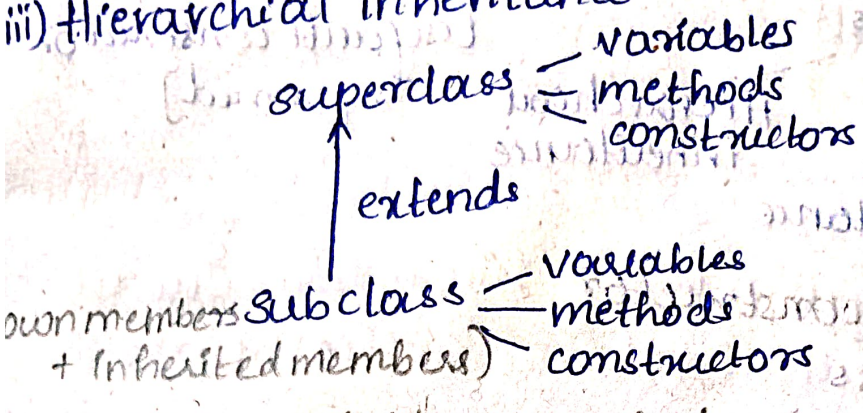
UNIT-2

Types/forms of Inheritance:

(i) Single level Inheritance

ii) Multi level Inheritance

iii) Hierarchial Inheritance



Superclass - variables & methods

⇒ Sub class contains variables, methods and constructors of its own.

⇒ Sub class contains five categories.

(i) sub class variables

(ii) sub class constructors

(iii) sub class methods

(iv) superclass variables

(v) super class methods.

NOTE: Super class constructors are not inherited to its sub class that is super class constructors are not available in sub class.

⇒ Super class contains three categories.

(i) Super class variables.

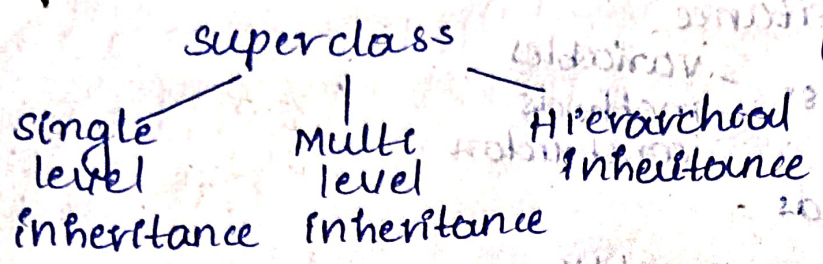
ii) Super class methods

iii) Super class constructor

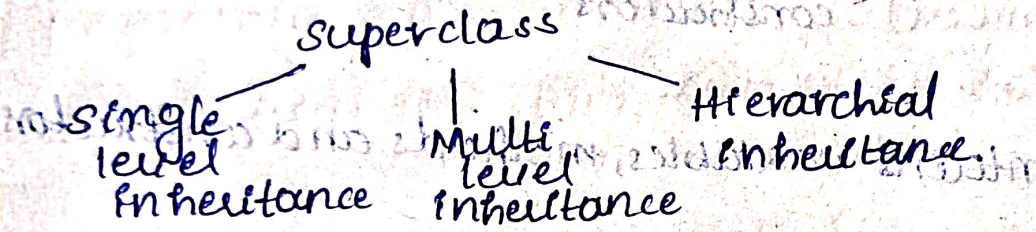
In inheritance give priority to sub class i.e, by creating sub class object we can access both sub class members and also super class members.

By creating object to sub class we can access all the members of the sub class and all the members of super class except super class constructors. Because superclass constructors are not inherited to its sub class.

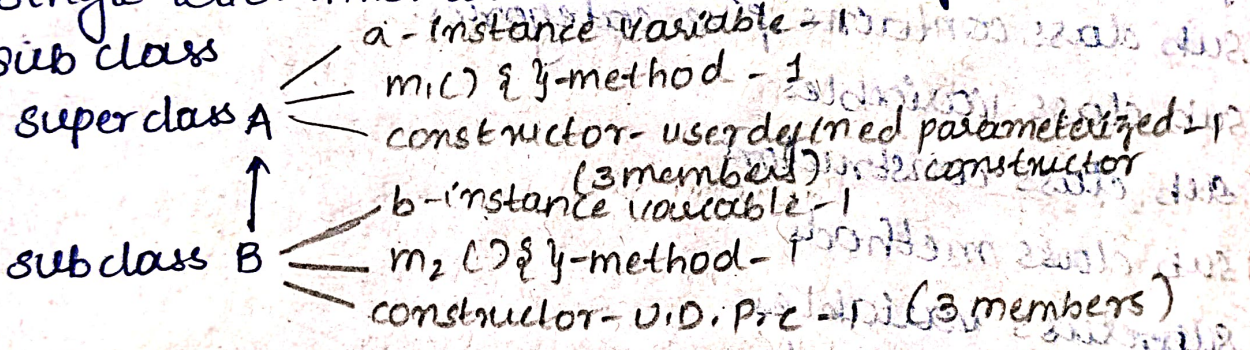
1. Super class - without constructors (3 members)



2. Super class - with constructor



⇒ Single level inheritance means one superclass, one sub class



sub class B
own members - 3 b, m2() & y constructor
Inherited members - 2
5 members

//Program.

```
Class A    // A is superclass
{ int a;
  void m1()
{ System.out.println("A's m1() method");
}
A(int a)
{
```

```
    this.a=a;
  }
}
class B extends A //B is sub class
{ int b;
  void m2()
{System.out.println("B's m2() method");
}
  B(int a, int b)
  this.a=a;
  this.b=b;
}
}
class Test
{public static void main(String[], args)
{//super class - A-Object
A. a1 = new A(1);
reference var   constructor
system.out.println(a1.a);
    a1.m1();
    //sub class - B-Object
     B b1 = new B(8,9);
reference var   constructor call
system.out.println(b1.a+"  "+b1.b);
  b1.m1();
  b2.m2();
}
}
```

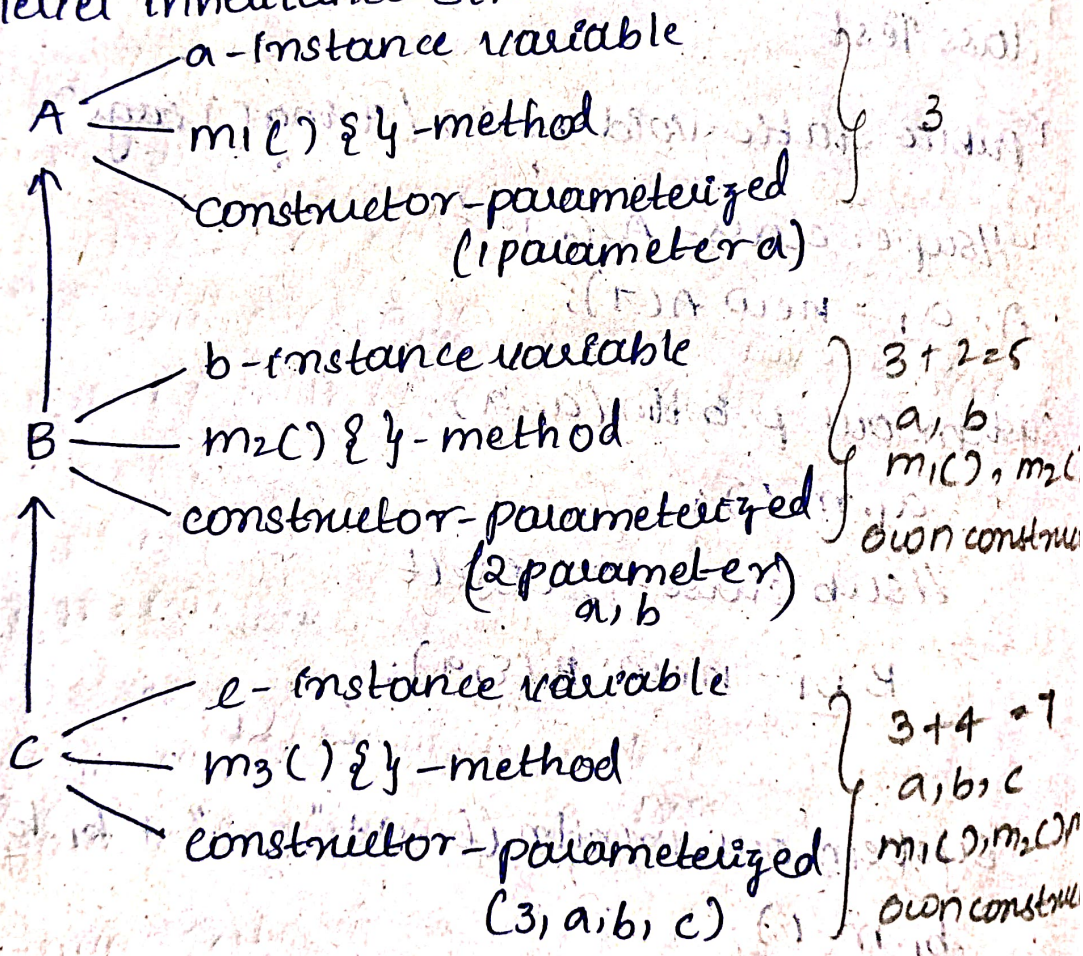
1. Save: Test.java
2. Compile: javac Test.java
3. Execute: java Test
   JVM Test.class

Output:

A's $m_2()$ method
A's $m_1()$
B's $m_2()$

. Write a java program using multi level Inheritance program using constructors

Multi level Inheritance Structure




```
//Program
class A //A is superclass
{ int a; //Instance variable
  void m1() //method
  {
```

System.out.println("A's m1() method");

}

A(int a) // constructed

{ this.a=a;

}

}

class B extends A //B is sub class.

{ int b; // Instance variable

void m2() //method

{ System.out.println("B's m2() method");

}

B(int a, int b)

8 , 9

this.a = a;

this. b = b;

}

}

class C extends B

{ int c; // Instance variable

void m3() method

{ system.out.println("C's m3() method");

}

C(int a, int b, int c)

8 9 10

this.a=a;

this.b=b;

this.c=c;

}

}

```
Class Test
{
public static void main (String[] args)
{
//super class - A-object
A.a1 = new A(7);
System.out.println (a1.a);
a1.m1();
//sub class - B-object
B.b1 = new B(8, 9)
System.out.println (b1.a + " " + b1.b);
b1.m1();
b1.m2();
//sub class-c-objelt
c.c1 = new c (8, 9, 10)
System.out.println (c1.a + " " + c1.b + " " + c1.c);
c1.m1();
c1.m2();
c1.m3();
}
}
```

save: Test.java

compile: javac Test.java → A.class, B.class, C.class, Test.class

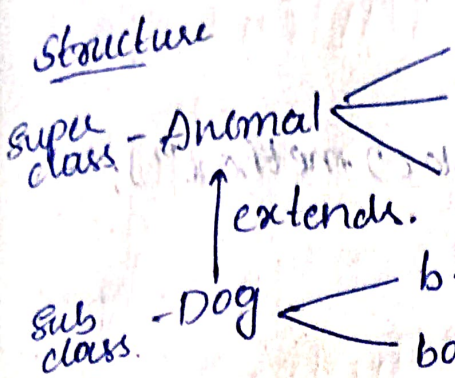
Execute: java Test
JVM Testclass

output:

A's m2() method
A's m1()
B's m2()
C's m3()

Q) Write a java program using Single-level Inheritance - Real time application.

Structure



In Inheritance programs we can write the main methods either in the last level sub class or we can take separate class bg (test class) write main method.

With inheritance we achieve is-a relationship.

2 (own) + 3 (Inheritance)

b, bark(); a, eat(), sleep()

a, b - variables - 2

eat(), sleep(), bark() - Methods - 3

5

There are 2 ways to write main methods in inheritance

1. last level sub class
2. Separate class.

```
//Program.
Class Animal
{ int a=10;
  void eat()
  {
  System.out.println("Animal's eat() method");
  }
  void sleep()
{ System.out.println("Animal's sleep() method");
}
```

```
//Default constructor
Animal()
{ a=10;
}
```

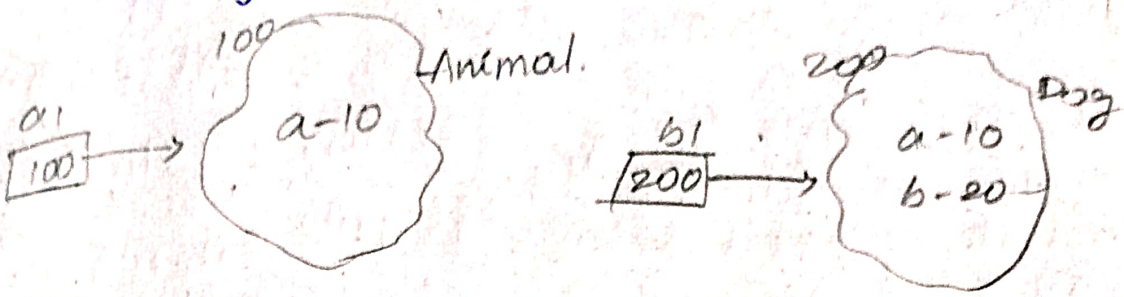
```
Class Dog extends Animal.
{ int b=20;
  void bark()
  {
  System.out.println("Dog's bark() method");
  }

  //Default constructor
  Dog()
  { a=10;
    b=20;
  }

Public static void main(String[] args)
  { Animal a1=new Animal();
    Dog d1= new Dog();
    System.out.println(a1.a);
    a1.eat();
    a1.sleep();
  System.out.println(d1.a+" "+d1.b);
    d1.eat();
    d1.sleep();
    d1.bark();
  }
}
```

1. Save: Dog.java.
2. compile: javac Dog.java → Animal.class, Dog.main
3. Execute: java Dog.
   JVM

A. flow of the program with memory:



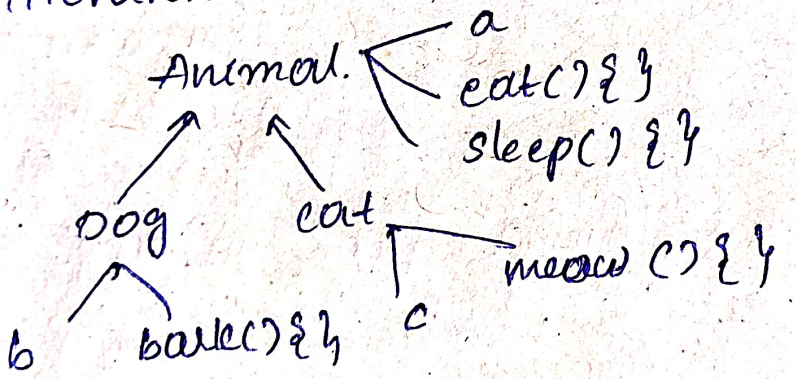
5. Output:

10
Animal's eat() method
Animal's sleep() method.

10 20
Animal's eat() method
Animal's sleep() method
Dog's bark() method.

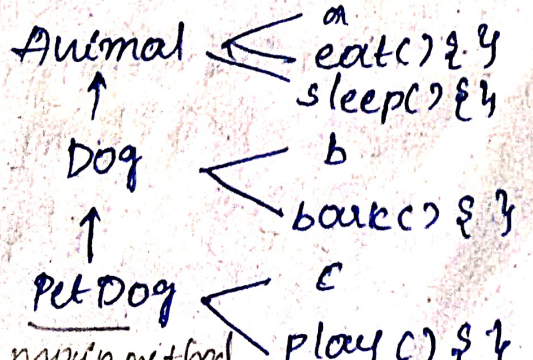
H/W

P1: Hierarchical Inheritance



In this program take separate class to write main method().

P2: Multilevel Inheritance.



1. Instance variable hiding.
2. Method overriding in Inheritance.

1. Instance variable hiding. [In Inheritance]

If both superclass and subclass are having the same instance variable with the same name then sub class instance variable will hide the superclass instance variable.

Ex1:
```
class Animal //superclass
{ int a=10;
}
class Dog extends Animal //subclass
{ int a=20;
}
Dog d1 = new Dog();
System.out.println(d1.a);
```
O/p: 20

Note:
Sub class will always give priority to its own members. If the members are not there in its own class then only it will inherit the member from its superclass.

Ex2:
```
class Animal //superclass
{ int a = 10;
}
class Dog extends Animal //subclass
{
```

```
    int b=20;
    }
    //subclass object
    Dog d1=new Dog();
    System.out.println(d1.a);
```

Output: 10.

2. Method overriding.

If both superclass and subclass are having same method name with same prototype (prototype means return type & parameter of the method). then subclass method will override the superclass.

Ex:

```
class Animal
{
  void eat()
  {
  system.out.println("Animal's eat() method");
  }
}
class Dog extends Animal
{
  void eat()
  {
  system.out.println("Dog's eat() method");
  }
}
Dog d1 = new Dog();
  d1.eat();
```

o/p:

Dog's eat() method

⇒ Sub class eat method is called overriding method, super class eat method is called overridden method.

⇒ Sub class method will override the superclass method.

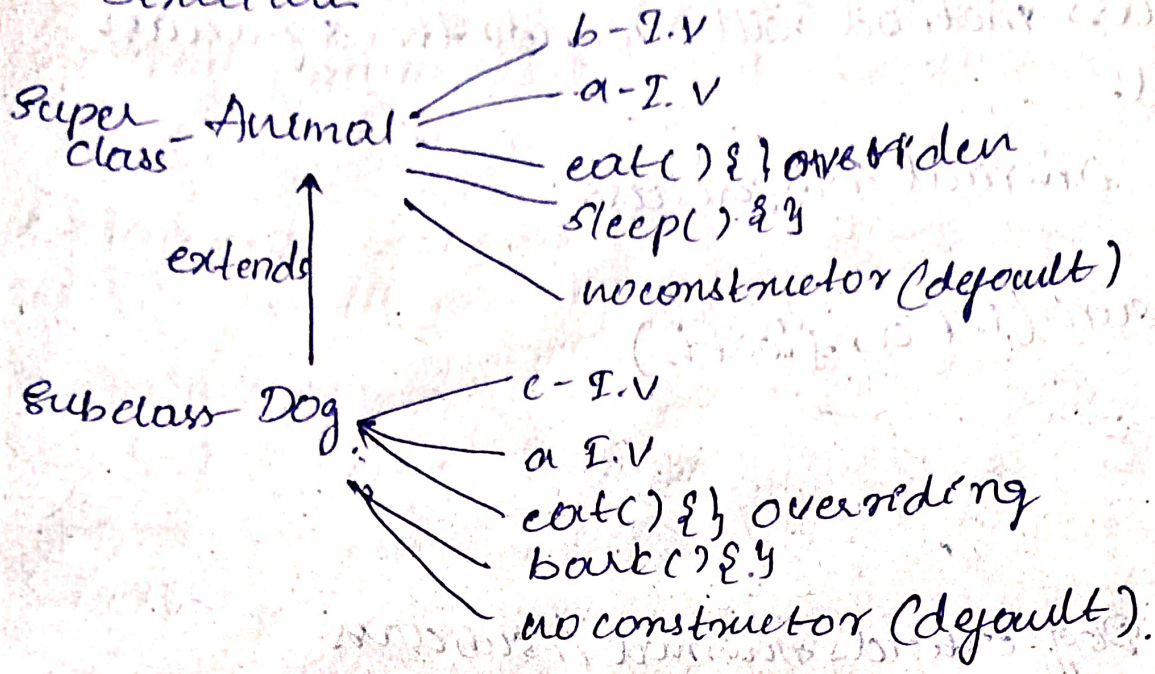
Ex: class Animal //superclass
{
int sum(int a, float b)
{ =
}
}
class Dog extends Animal //sub class
{ int sum (int c, float d) //overriding
{ =
}
}

Ex: class Animal
{ int sum (float a, int b)
{ =
}
}
class Dog extends Animal
{ int sum (int c, float d)
{ =
}
}

The above example does not comes under method overriding because order of parameters are different, it must be same.

Write a java program using Instance variable hiding and method overriding.

Structure




```
//Program.
{ class Animal
  {
    int a=10, b=20;
    void eat() //Overridden method.
    {
    System.out.println("Animal's eat() method");
    }
    void sleep()
    {System.out.println("Animal's sleep() method");
    }
```

```
//Default Constructor.
Animal()
{ a=10;
  b=20;
}
```

```
}
```

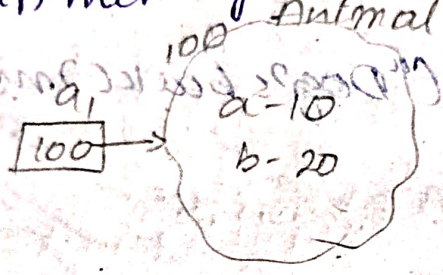
```
class Dog extends Animal
{
int a=30, c=40;
void eat() //overriding method
{
System.out.println("Dog's eat() method");
}
void bark()
{
System.out.println("Dog's bark() method");
}
//Default constructor
Dog()
{ a=50;
  b=20;
  c=40; }
}
class Test
{
Public static void main(String[] args)
{
Animal a1 = new Animal();
System.out.println(a1.a+" "+a1.b);
a1.sleep();
Dog d1 = new Dog();
System.out.println(d1.a);
System.out.println(d1.b);
System.out.println(d1.c);
d1.eat();
d1.sleep();
d1.bark();
```

1. Save : Test.java.

2. compile : javac Test.java → Animal.class, Dog.class, Test.class

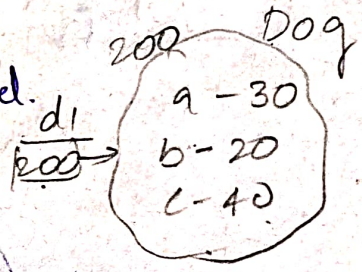
3. execute : java Test
JVM

4. flow of the program with memory



5. Output:

10 20
Animal's eat() method
Animal's sleep() method.
30
20
40
Dog's eat() method
Animal's sleep() method
Dog's bark() method.



Q. Explain super keyword in java with examples.

There are four ways to use super keyword in java

1. super.variable
2. super.method
3. super()
4. super(parameters)

1. Super. variable.

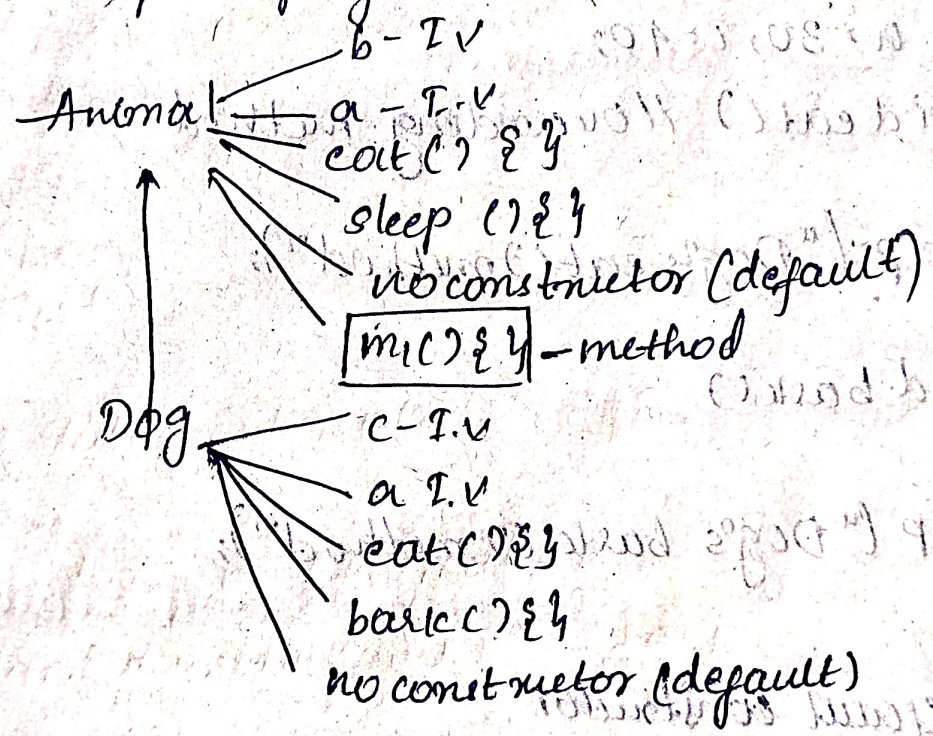
In case of Instance variable hiding to access super class Instance variable inside the sub class we use super.variable.

2. Super. method

In case of method overriding to access the super class method inside the subclass we use super. method

Write a java program using super. member (i.e super.variable & super.method).

Structure of the program:

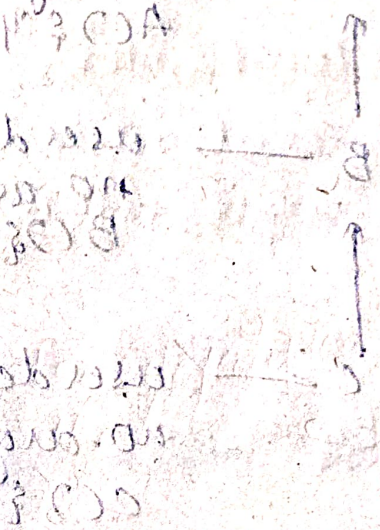


//program:

```
class Animal
{
 int a=10, b=20;
 void eat() //overridding method
 {
  s.o.p ("Animal's eat() method");
 }
 void sleep()
 {
```

```
S.O.P ("Animal's sleep() method");
}
// Default constructor
Animal()
{ a=10;
  b=20;
}
}
class Dog extends Animal
{
  int a=30, c=40;
  void eat() // overriding method
  {
  S.O.P ("Dog's eat() method");
  }
  void bark()
  {
  S.O.P ("Dog's bark() method");
  }
  // Default constructor
   Dog()
  { a=30;
    b=20;
    c=40;
  }
void m1()
{ super.a;
  super.eat();
```

```
class Test
{
public static void main(String[] args)
{
Animal a1 = new Animal();
s.o.p(a1.a + "  " a1.b);
a1.eat();
a1.sleep()
Dog d1 = new Dog()
s.o.p(d1.a);
s.o.p(d1.b);
s.o.p(d1.c);
d1.eat();
d1.sleep();
d1.bark();
d1.m1();
}
}
```



Output:

```
10  20
Animal's eat() method
Animal's sleep() method
30
20
40
Dog's eat() method
Animal's sleep() method
Dog's bark() method
10
Animal's eat() method.
```

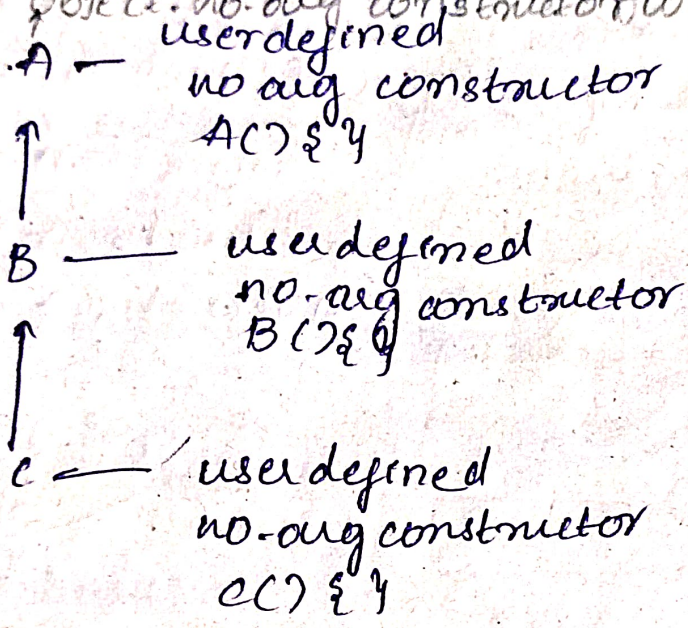
3. super()

From the subclass constructor to access the super class no arg constructor we use super()

Q) write a java program to explain super() or constructor chaining.

We use super keyword only in inheritance.

Multilevel Inheritance Structure

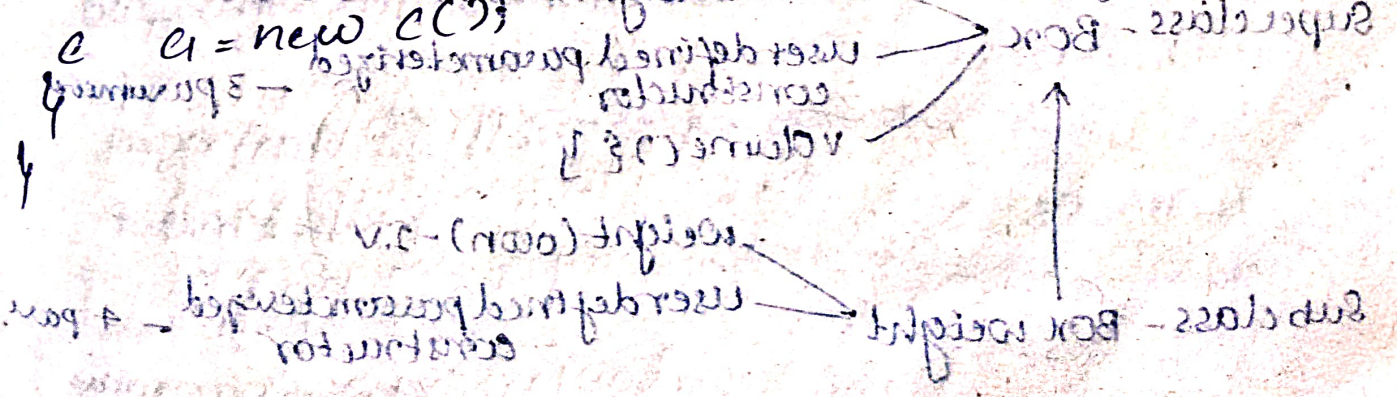


Inside each and every constructor the first statement must be either super([parameters]) or this([parameters]) By default, it is super()

```
Class A extends Object
{ //user defined no arg constructor
  A()
  {
  super(); // added by compiler
  S.O.P ["A's constructor"];
  }
}
```

```
class Object
{Object()
 {}
```

```
class B extends A
{ // user defined no-arg constructor
  B()
  { // by default
    super();
    System.out.println("B's constructor");
  }
}
class C extends B
{ // user defined no-arg constructor
  C()
  { // by default
    super();
    System.out.println("C's constructor");
  }
}
class Test
{
  public static void main(String[] args)
  {
    A a1 = new A(); // constructor call to user defined no-arg constructor
    B b1 = new B();
    C c1 = new C();
  }
}
```



1. Save : Test.java

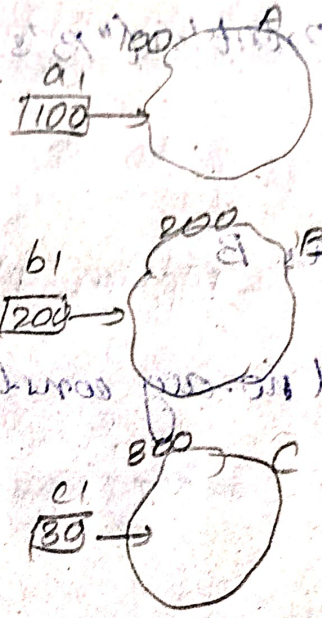
2. compile : javac Test.java → A.class, B.class, C.class, Test.class

3. execute : java Test.
JVM

7. flow of the program with memory

8. Output:

A's constructor
A's constructor
B's constructor
A's constructor
B's constructor
C's constructor.



4. super (parameters)

From the subclass constructor to access super class constructor we use super (parameters)

Q) write a java program using super (parameters).

super refers to super class.

Part I: single level Inheritance structure

Superclass - Box → width, height, depth — 3 I.Vs
→ user defined parameterized constructor — 3 parameters
→ volume() { }

Subclass - BoxWeight → weight (own) - I.V
→ user defined parameterized constructor — 4 par.
to access superclass constructor we use super(para)

Inherited -
width, height, depth - 3. I. V
volume() { } - 1M

4 - Inherited
constructor - not inherited.

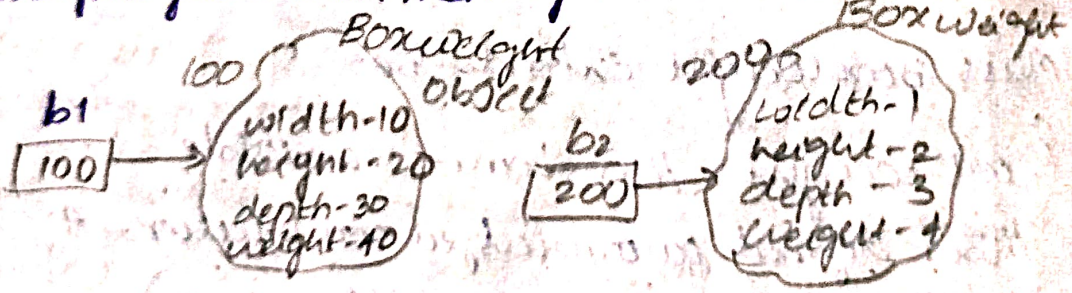
//Program.

```
class Box
{ int width, height, depth; //Instance variables. - 3
  //user defined parameterized constructor - 3P
  Box (int width, int height, int depth)
  { this.width = width;
    this.height = height;
    this.depth = depth;
  }
  // NO-arg method
  void volume()
  { System.out.println(width * height * depth);
  }
}
class BoxWeight extends Box
{ int weight; //Instance variable - 1
  //user defined parameterized constructor - 4
  BoxWeight (int w, int h, int d, int wt)
  {
    super(w, h, d); //super parameters
    weight = wt;
  }
}
```

```
class Test
{
Public static void main (String [] args)
{
//Subclass i.e Box weight class object - 2
Box weight b1 = new Box weight (10,20,30,40);
Box weight b2 = new Box weight (1,2,3,4);
b1.volume();
b2.volume();
System.out.println(b1.weight + "  " + b2.weight
}  sysO.p("First box weight is" + "  " + b1.weight);
}  s.o.p ("second box weight is" + "  " + b2.weight);
```

Part-II

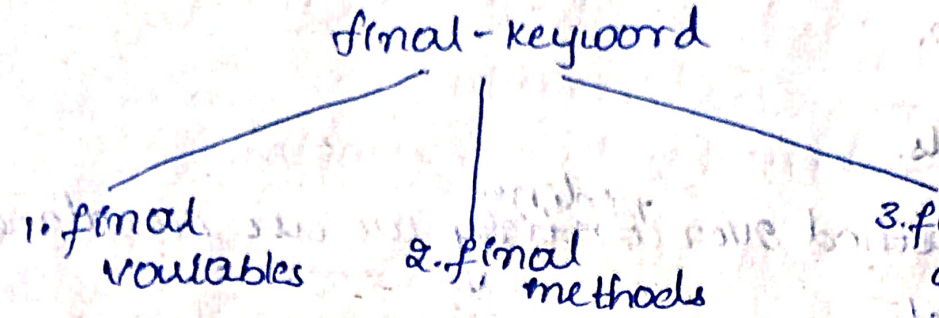
1. Save: Test.java
2. compile: javac Test.java ← Box.class, BoxWeight.class, Test.class (main)
3. execute: java Test
   JVM Test.class
4. flow of the program with memory



5. Output:

```
Volume is 6000
Volume is 6
First box weight is 40
Second box weight is 4
```

There are three ways to use final keyword in Java.



1. final variables:

//Program on final variables.

⇒ final variables cannot be modified throughout the program because in java final variables ~~are~~ are treated as constants in java.

⇒ To explain final variables inheritance is not required.

⇒ final variables must be initialised at the time of declaration.

⇒ we can use final keyword for instance variables, static variables and local variables.

```
class Test
{
final int a=10; // a is instance variable
final static int b=20; // b is static variable.
public static void main(String[] args)
 {
 final int c=30; // c is local variable.
  Test t1=new Test();
  S.O.P(t1.a+" "+t1.b+" "+t1.c);
// t1.a=t1.a+10; //C.E
// t1.b=t1.b+20; //C.E
// t1.c=t1.c+30; //C.E
 }
}
```

1. Save: Test.java.

O/p: 10 20 30.

2. final methods.

⇒ To prevent method overriding we use final meth

```
class Animal
{
final void eat().//overridden method.
  {
  }
}
class Dog extends Animal
  { void eat() //overriding method.
    {
    }
O/p: compiler error.
```

3. final classes.

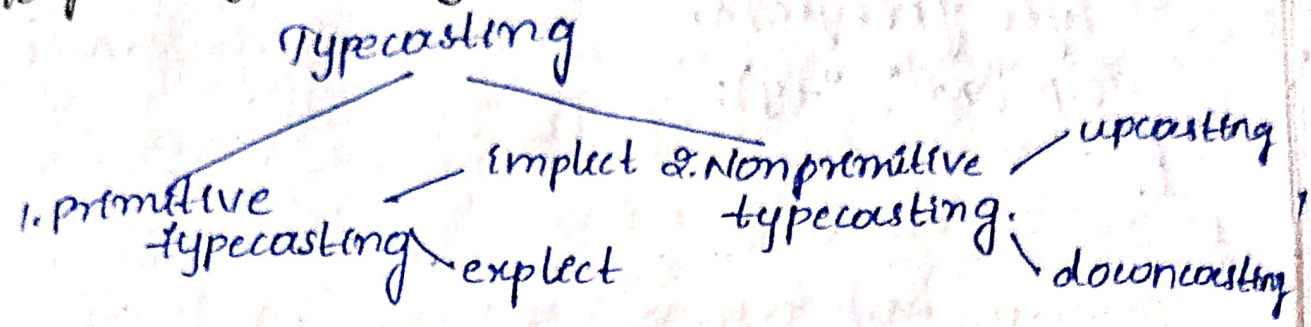
⇒ To prevent inheritance we use final classes

⇒ final classes cannot be sub class.

```
final class Animal
  {
    void eat()  //overridden method
      {
      }
  }
class Dog extends Animal
{ void eat()
  {
  }
}
```

O/p: compiler error.

Q. Explain typecasting in java with examples.



Primitive typecasting:
If the casting is done with any one of the 8 primitive datatypes then it is called primitive typecasting.

Non primitive typecasting:
If the casting is done with non primitive datatypes that is either with only classes.

Implict typecasting: Lower datatype is reassigned to higher datatype. It is taken by compiler.

Explict typecasting: Higher datatype is assigned to lower datatype. It is taken care by programmer.

Upcasting: casting is done with superclass
downcasting: casting is done with subclass.

Q. write a java program on primitive typecasting.

```
//program.
class Test
{public static void main(String[] args)
{ //Implicit typecasting
  byte b=10;
  int a=b; //int a=(int)b;
  S.O.P(b+" "+a);
  //Explict typecasting
  int x=100;
```

```
//byte y = x; // compiler error
byte y = (byte)x;
S.O.P (x+" "+y);
}
}
```

⇒ In Java typecasting is of four types. They are

1. Implict typecasting } primitive
2. Explict typecasting } datatypes(8)
3. Upcasting } non-primitive
4. downcasting } datatypes
   - classes
   - interface

Upcasting: Inheritance is must

superclass ↑ subclass

Defination: subclass object is assigned to superclass reference variable with this reference variable we can access all the members of the superclass except overridden methods

⇒ In case of overridden methods of the superclass, subclass overriding methods are executed.

Write a java program using upcasting.

Part I: Single level inheritance structure

Superclass Animal
- a, b - v - 2
- eat() {}
- sleep() {} } m-2

extends

Subclass Dog
- c - v - 1
- eat() {}
- bark() {} } m-2

⇒ Upcasting is like implict typecasting which is taken care by the compiler

Part II: Program

```
//Program on upcasting
class Animal
{ int a=10, b=20;
  void eat()  //overridden method
  {
   system.out.println("Animal's eat() method");
  }
  void sleep()
  {
  System.out.println("Animal's sleep() method");
  }
}
```

```
//Default constructor
Animal()
{ a=10; b=20;
}
```

```
class Dog extends Animal
 { int c=30;
  void eat() //overriding method.
  {
  system.out.println("Dog's eat() method");
  }
  void bark()
 {system.out.println("Dog's bark() method");
 }
//Default constructor
  Dog()
  { a=10;
    b=20;
    c=30;
  }
}
class Test
{ Public static void main(String[] args)
```

```
{
  //upcasting.
  Animal a1 = new Dog();
  //Animal a1 = (Animal) new Dog();
  System.out.println(a1.a + " " + a1.b);
  a1.eat(); //upcasting
  a1.sleep();
 }
}
```

Part-III

1. save: Test.java
2. compile: javac Test.java → Animal.class, Dog.class, Test.class
3. execute: java Test

O/P: 10 20
Dog's eat() method.
Animal's sleep() method.

Applications of upcasting

1. Abstract classes
2. Interfaces.

Abstract methods.

Concrete methods	abstract methods
r1. should have method body.	r1: should not have method body
r2. should not declare with abstract keyword.	r2: should be declare with abstract keyword

Ex: void eat()
{
=
}
eat() - concrete method

method.

r3: should end with semi column (;)

Ex:
abstract void m1();
m1() - abstract method.

Q) Explain abstract classes in java with examples.

In java classes are of two types

classes (2 types)

1. concrete classes
   1. must contain only concrete methods
   2. should not declare with abstract keyword

Ex: class A
{
void m1()
{
}
void m2()
{
}
} A - concrete class

2. Abstract classes
   1. should declare with abstract keyword
   2. must contain only concrete methods or abstract methods or both.

Ex1: abstract class A
{
void m1() // concrete method
{
=
}
void m2()
{
=
}
} A - abstract class, concrete method

Ex2:
abstract class B
{ //abstract method
abstract void m1();
abstract void m2();
}
B - abstract class

Ex3:
abstract class C
{ //both methods
void m1()
{
}
abstract void m2();
}
C - abstract class

Q) Explain abstract classes in JAVA with example

⇒ A class which is declared with abstract keyword is called abstract class.

Ex: abstract class A
{
=
}

A is called abstract class.

⇒ An abstract class may contain only concrete methods, only abstract methods or both

⇒ Abstract classes must be extended, i.e there must be atleast one sub class which extends the abstract class

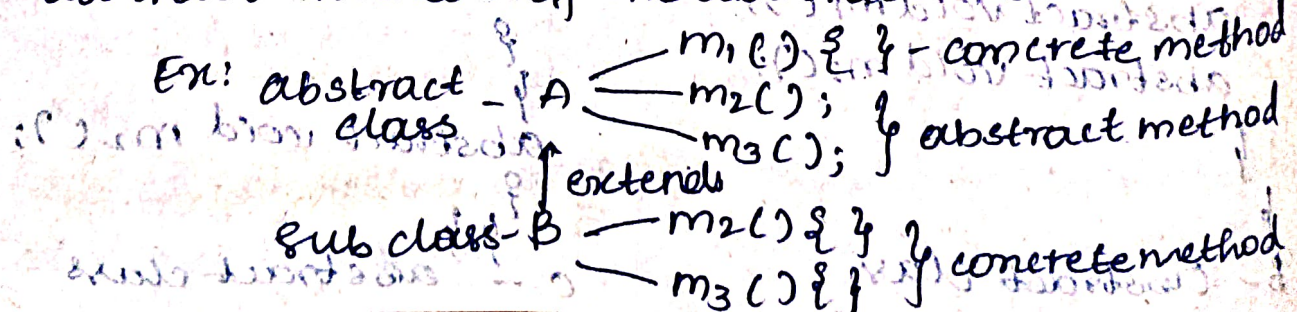
Ex: abstract class A
{
=
}

class B extends A
{
=
}

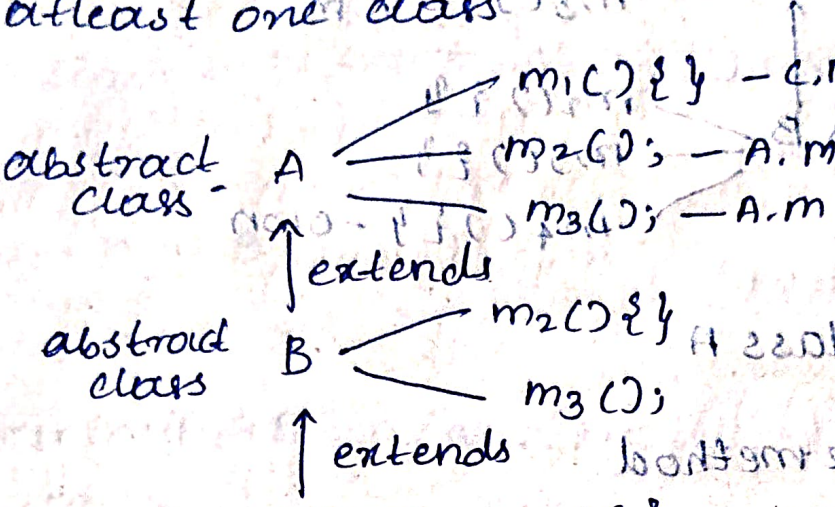
class c extends A
{
=
}

⇒ The role of the subclass which extends abstract class is to provide implementations to all the abstract methods of the abstract class

Ex:



⇒ If the subclass doesn't want to provide implementation to all abstract methods of the abstract class, then declare the sub class as abstract and the subclass must be extended by atleast one class



⇒ A class which contains atleast one abstract method must be declared as abstract.

⇒ For abstract classes, we cannot create objects with new operator.

⇒ For abstract classes, we can only create reference variables

✗ A a1 = new A();

✓ A a1;

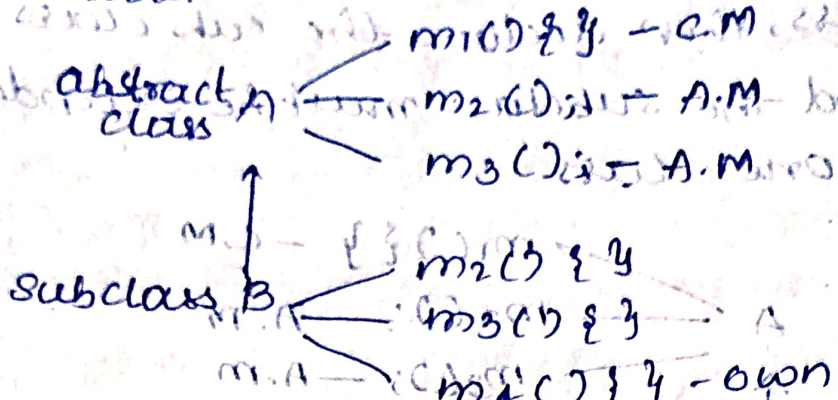
⇒ Without creating objects for abstract classes, we access the members of the abstract classes by upcasting.

⇒ With reference variable, we can access all the members of abstract class.

A a1 = (A) new B();

Write a Java program using abstract.

Structure:




```
//program
abstract class A
{
//concrete method
int a=10;
void m1()
{
System.out.println("A's m1() method");
}
//abstract methods
abstract void m2();
abstract void m3();
}
class B extends A
{
int b=20;
void m2()
{
System.out.println("B's m2() method");
}
void m3()
{
System.out.println("B's m3() method");
}
void m4()
{
```

```
System.out.println("B's m1() method");
}
}
class Test
{public static void main(String[] args)
{//upcasting
A a1 = new B();
```

reference variable — subclass object

```
system.out.println(a1.a);
a1.m1();
a1.m2();
a1.m3();
}
}
```

save: Test.java

compile: javac Test.java

Execute: java Test

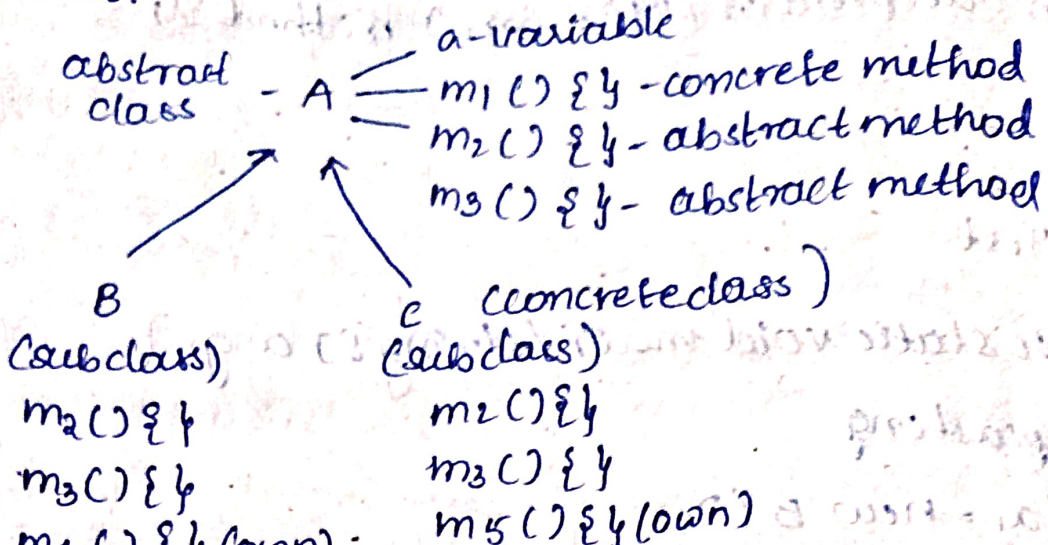
output: 10
A's m1() method
B's m2() method
C's m3() method..

Q* write a java program on using abstract classes with hierarchical inheritance

To write programs on abstract classes we must use inheritance

Hierarchical inheritance structure:

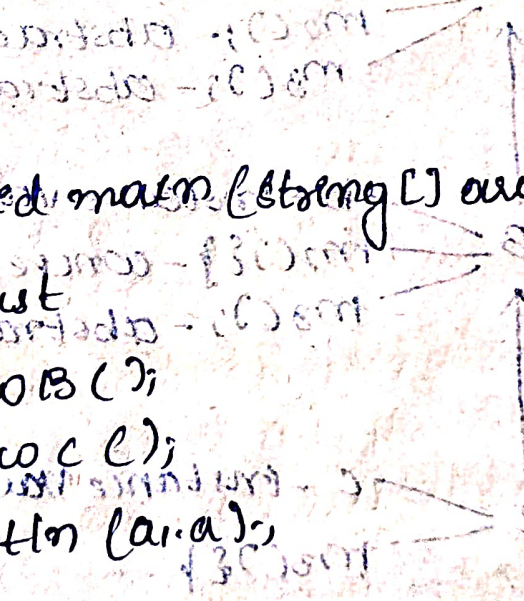
Structure:



//Program -

```
abstract class A
{
int a=10; //Instance variable
 //concrete method
  void m1()
 {
 System.out.println("A's m1() method");
 }
 //Abstract method
 abstract void m2(); //overridden
 abstract void m3(); //overridden
 }
 class B extends A
 {
  int b = 20; //Instance variable
  //concrete methods only
    void m3() //overriding
 {
 System.out.println("B's m2() method");
 }
  void m3() //overriding
 {
 System.out.println("B's m3() method");
 }
```

```
//non method - concrete
  void m4()
{
System.out.println("B's m4() method");
}
}
class C extends A
{
 int c=30;
 // concrete methods
 void m2() // overriding
{
System.out.println("C's m2() method");
}
 void m3() //overriding
{
System.out.println("C's m3() method");
}
//non method
  void m5()
{
System.out.println("C's m5() method");
}
class Test
{
public static void main(String[] args)
{
//upcasting-must
 A a1 = (A) new B();
 A a2 = (A) new C();
 System.out.println(a1.a);
   a1.m1();
   a1.m2();
   a1.m3();
 System.out.println(a2.a);
   a2.m1();
```



```
        a2.m2();
        a2.m3();
    }
}
```

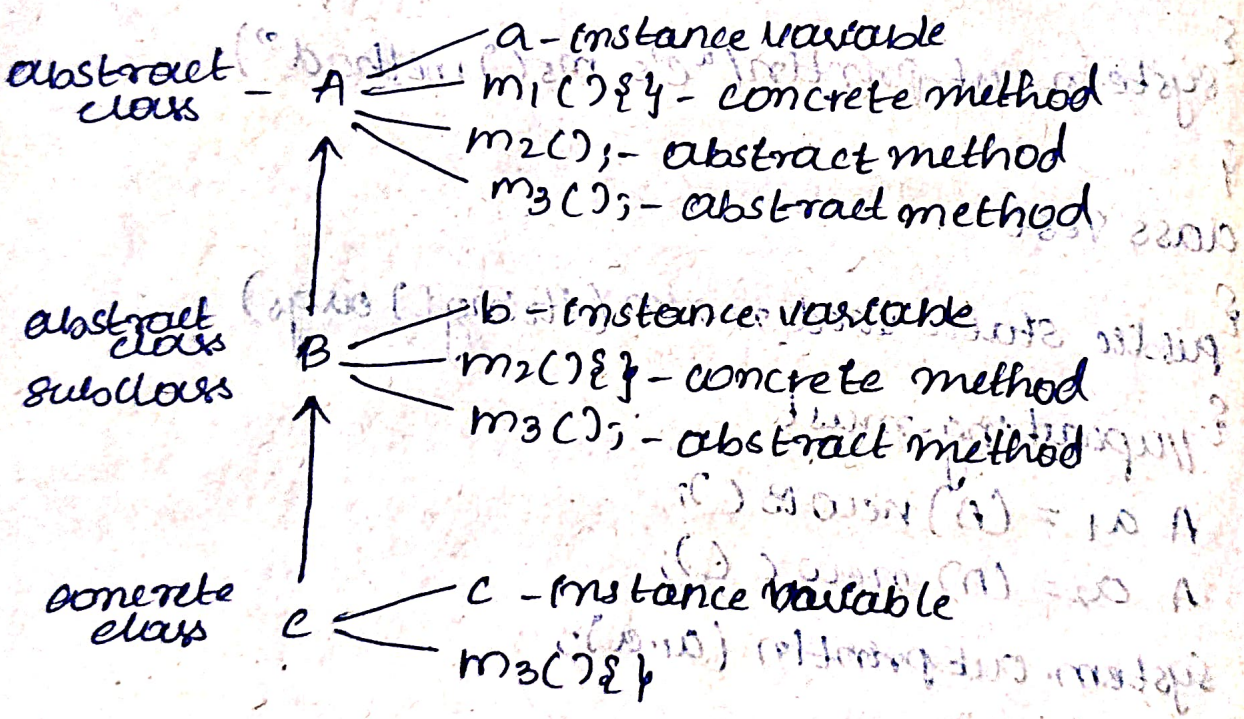
save : Test.java

compile : Javac Test.java → A.class, B.class, C.class, Test.class

Execute : Java Test

Output : 10
A's m1() method
B's m2() method
B's m3() method
10
A's m1() method
C's m2() method
C's m3() method.

Write a java program using abstract class with multi level inheritance.



A class which contains atleast one abstract method must be declared as abstract.

We access the members of the abstract class A using C i.e upcasting.

syntax - upcasting

A a1 = (A) new C();

we access the members of the abstract class B using c i.e upcasting.

syntax:

B b1 = (B) new C();

↓

only the members of class B.

// program:

//A is abstract class

```
Abstract class A
{ int a=10;
  void m1()
{
System.out.println("A's m1() method");
}
abstract void m2();
abstract void m3();
}
class B extends A
{
 int b=20;
  void m2
 {
 System.out.println("B's m2() method");
}
 abstract void m3();
} //non method
  void m4
{
```

```
System.out.println("B's m4() method");
}
//common method
  void m5()
{
System.out.println("B's m5() method");
}
class Test
{
public static void main(String[] args)
{
//upcasting
  A a1 = (a) new c();
System.out.println(a1.a);
  a1.m1();
  a2.m2();
  a3.m3();
B b1 =(B) new c();
System.out.println(b1.a);
  b1.m1();
  b1.m2();
  b1.m3();
  b1.m4();
}
}
```

save: Test.java

compile: javac Test.java

A class B class C class Test class

Execute: Java Test

output:

10
A's m1() method
B's m2() method
C's m3() method

10
A's m1() method
B's m2() method
C's m3() method
B's m4() method.

Containers in java:

There are three containers in java. They are.

1. Concrete classes
2. Abstract classes
3. Interfaces.

Interfaces:

Syntax:

keyword → Interface ← identifier
Interfacename

{
1. final variables
2. abstract methods
}

Inside interface we can write only final variables and abstract methods.

⇒ For interfaces we cannot create objects with new operator

⇒ We can only create reference variables for interfa

⇒ Interfaces must be implemented by atleast one cla

Ex1: (default)
```
interface I1
{
final int a=10;
abstract void m1();
abstract void m2();
}
```

⇒ Interfaces contain 100% abstract methods.

⇒ We cannot write non final variable and concrete methods inside the interface.

Points to remember:

1. By default (compiler) all the variables of the interf are public static final
2. All the methods of interfaces are public abstract

3. By the all the interfaces are <default>

⇒ for interfaces the programmer can use public access modifier if required.

Upcasting in interface.

Implementing class object is assigned to interface reference variable, with that ref variable we can access all the members of the interface except overridden methods

Ex 2: By default

```
<default> abstract interface I1
{
public static final int a=10, b=20;
public abstract void m1();
public abstract void m2();
}
```

In case of overridden methods of the interface implementing class overriding methods are executed.

Ex 3: By default

```
interface I1
{
int a=10, b=20;
void m1();
void m2();
}
```

I1 (interface) ← implements ← A (implementation class)

→ B/w class & interface we use the keyword implements

→ without creating objects for interface how do we access the members of interface, upcasting

Q) Explain the differences between concrete classes abstract classes and interfaces.

containers (3)

	1. concrete classes	2. Abstract classes	3. Interfaces
1. Variables	both final and non final variables	both final & non final variables	only final variables
2. methods	only concrete methods	both concrete & abstract methods	only abstract methods
3. Creating objects with new operator	Yes, we can create objects with new operator	No, we can't create objects with new operator	No, we can't create objects with new operator

4. How do we access the members	using objects	upcasting	upcasting

Interfaces:-

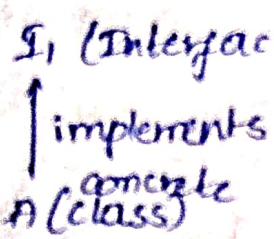
```
interface interfacename
{
1. final variables
2. abstract methods
}
```

```
for class
class classname
{
1. variables
2. methods
3. constructors
}
```

Programs on interfaces, concrete classes, abstract cla

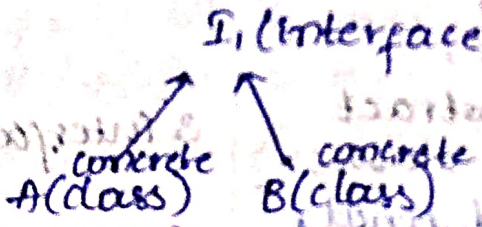
One Interface, One class

P1: Interface I_1, class A implements I_1



One Interface, Two classes

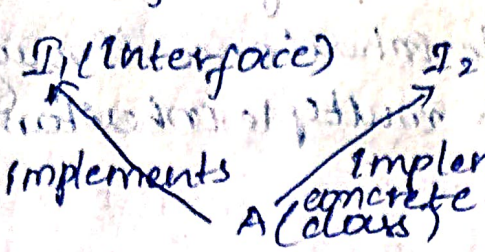
P2: Interface I_1, class A implements I_1, class B implements I_1.



⇒ onee Interface can be implemented by any number of classes

⇒ classes are denoted by A, B, C ---, and interfaces are denoted by I_1, I_2, I_3 ----

P3: Two interfaces, one class



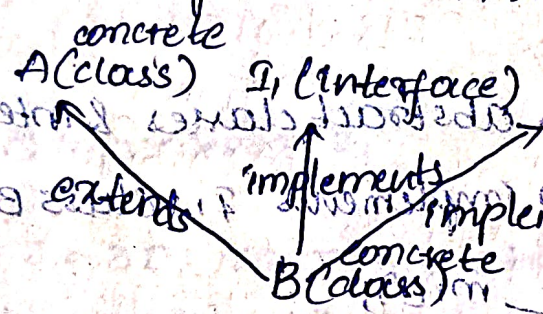
Interface I_1, Interface I_2, class A implements I_1, I_2.

⇒ one class can implements any number of interfaces.

P4: One class can extend only one class and can implement any no. of interfaces. at same time.

Two interfaces, Two classes

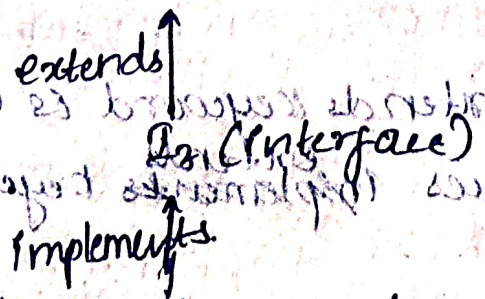
class A, Interface I_1, Interface I_2, class B extends A implements I_1, I_2.



P5: Between two interfaces we can use extends keyword.

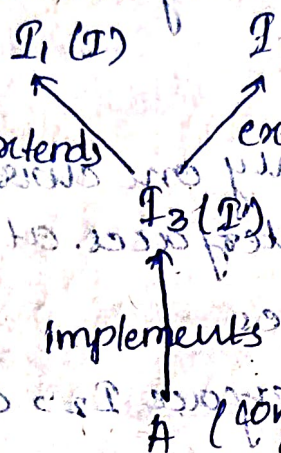
interface I_1, interface I_2 extends I_1

class A implements I_2 (Interface)



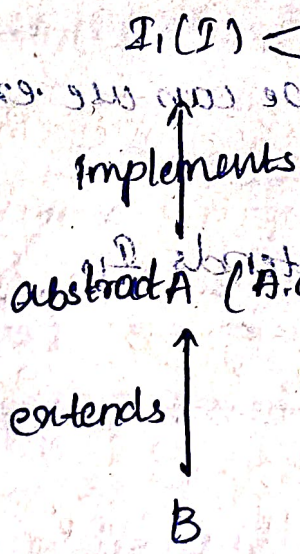
P6: One interface can extend any number of interfaces.
(Java supports multiple inheritance through interface
Java doesn't support multiple inheritance with classes)

Interface I_1, Interface I_2, Interface I_3 extends I_1, I_2
Class A implements I_3.



P7: On concrete classes, abstract classes & interfaces.

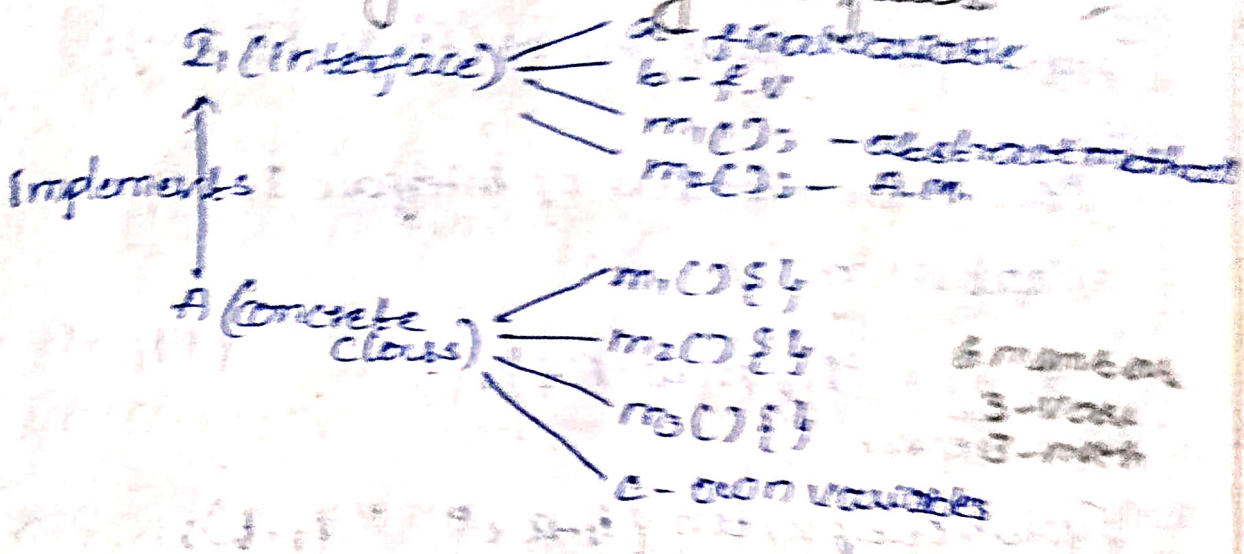
Interface I_1, abstract class A implements I_1, class B extends A



⇒ Between two classes extends keyword is used.
⇒ Between two interfaces extends keywords is used
⇒ Between class and interface implements keyword is used

Program 1:

Write a java program using interfaces




```
interface I1
{
 final int a=10, b=20;
public abstract void m1(); //overridden
public abstract void m2(); //overridden
}
class A implements I1
{
 int c = 30; //non variable
 public void m1() //overriding
{
System.out.println("A's m1() method");
}
 public void m2() //overriding
{
System.out.println("A's m2() method");
}
 void m3()
{
System.out.println("A's m3() method");
}
}
```

```
Class Test
{
Public static void main (String [] args)
{
//access the members of interface I1
//upcasting.
I1 i1 = (I1) new A();
ref.var
System.out.println (i1.a + " " i1.b);
i1.m1();
i2.m2();
A a1 = new A();
System.out.println (a1.a + " " + a1.b + " " + a1.c)
a1.m1();
a1.m2();
a1.m3();
}
}
```

Save: Test.java

compile: javac Test.java — A.java, Test.java

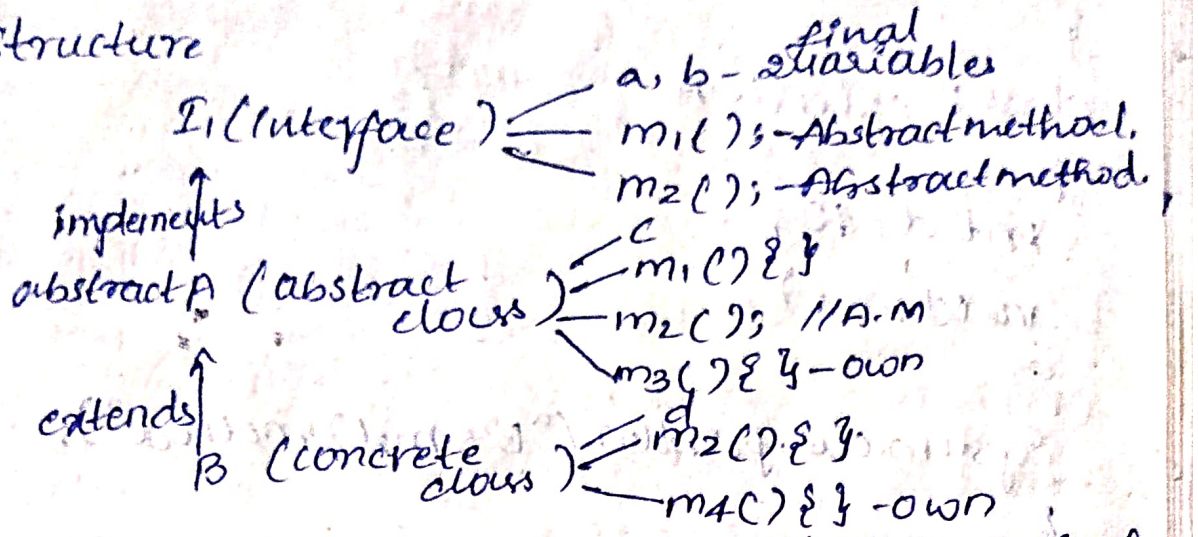
Execute: java Test

output:

```
10 20
A's m1() method
A's m2() method
10 20 30
A's m1() method
A's m2() method
A's m3() method.
```

Program 7:

Structure



Interface I1 contains 4 members that is 2 final variables and 2 abstract methods.

```
// Program on interface
interface I1
{
public static final int a=10, b=20;
public abstract void m1();
public abstract void m2();
}
abstract class A implement I1
{ int c=30;
  void m1()
  {
  System.out.println("A's m1() method");
  }
  //public abstract void m2();
  // m2() - Abstract method.
  //own method
  void m3()
  {
  System.out.println("A's m3() method");
```

```
    }
}
class B extends A
{
  int d=40;
  void m2()
  {
  System.out.println("B's m2() method");
  }
  //own method
  void m4()
  {
  System.out.println("B's m4() method");
  }
}
class Test
{
 public static void main(String[] args)
 { //accessing the members of I1 (I)
  //using upcasting - I1, B
   I1 I1 = (I1) new B();
  System.out.println(I1.a+" "+I1.b);
    I1.m1();
    I1.m2();
  //accessing the members of A (A, C)
   //using upcasting - A, B
   A a1 =(A) new B();
  System.out.println(a1.a+" "+a1.b+" "+a1.c
   a1.m1(); a1.m2(); a1.m3();
```

```
//accessing members of B(c.e)
    B b1 = new B();
System.out.println("b1.a+" "+b1.b+" "+
                    b1.c" "+b1.d);

    b1.m1();
    b1.m2();
    b1.m3();
    b1.m4();
  }
}
```

Save : Test.java

compile : javac Test.java → A.class, B.class, Test.class

execute : java Test

output :

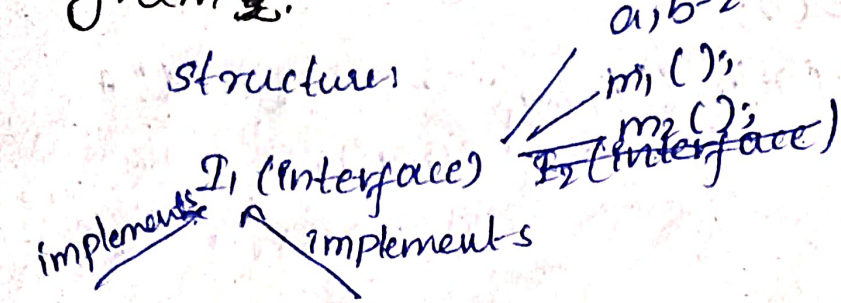
```
10  20
A's m1() method
B's m2() method
```

```
10  20  30
A's m1() method
B's m2() method
A's m3() method
```

```
10  20  30  40
A's m1() method
B's m2() method
A's m3() method
B's m4() method.
```

Program 2:

Structure:




```
// Program on interface

interface I1
{
  int a=10,b=20;
  void m1();
  void m2();
}

class A implements I1
 {
  int c=30;
public void m1()
  {
  System.out.println("A's m1() method");
  }
public void m2()
  {
  System.out.println("A's m2() method");
  }
  void m3()
  {
```

```
System.out.println("A's m3() method");
}
}
class B implements I1
{ int d = 40;
public void m1()
{
System.out.println("B's m1() method");
}
public void m2()
{
System.out.println("B's m2() method");
}
void m4()
{
System.out.println("B's m4() method");
}
}
class Test
{
public static void main(String[] args)
{ //access the members of I1 using A & B
// upcasting - 1. I1, A  2. I1, B
I1 i1 = (I1) new A();
System.out.println(i1.a+" "+i1.b);
i1.m1();
i1.m2();
I1 i2 = (I1) new B();
System.out.println(i2.a+" "+i2.b);
i2.m1(); i2.m2();
}
}
```

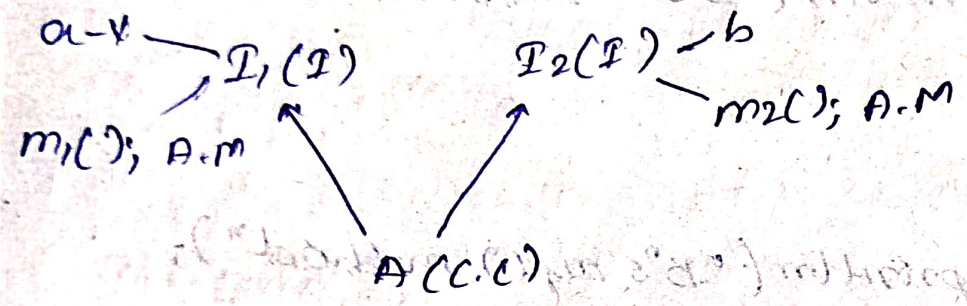
Output:

10 20
A's m1() method
A's m2() method

10 20
B's m1() method
B's m2() method.

Program 3:

Structure




```
//Program on interface
interface I1
{
 public static final int a=10;
 public abstract void m1();
}
 interface I2
{
  int b=20;
  void m2();
}
  class A implements I1, I2
  { int c=30;
public void m1()
{
```

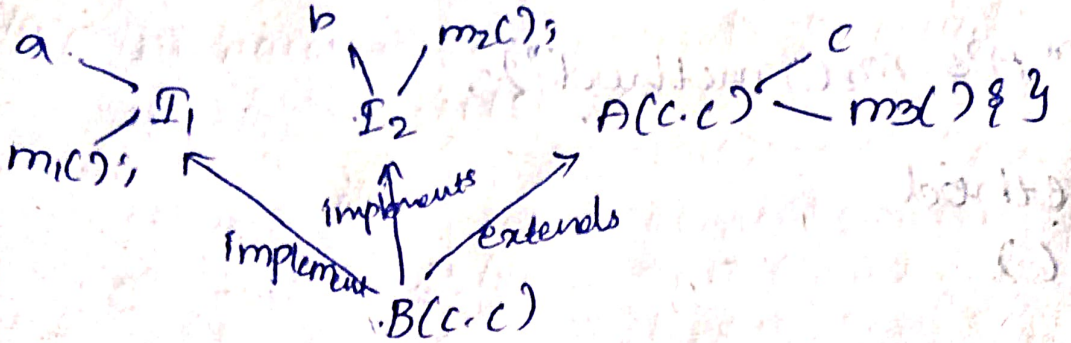
```
    System.out.println("A's m1() method");
}
public void m2()
{
    S.O.P("A's m2() method");
}
//own method
void m3()
{
    S.O.P("A's m3() method");
}
}
class Test
{
    public static void main(String[] args)
    {
    //access the members of I1 with A
    // upcasting - I1, A
    I1 i1 = (I1) new A();
    System.out.println("i1.a");
        i1.m1();
    I2 i2 = (I2) new(A);
    System.out.println(i2.b);
        i2.m2();
    }
}
```

Output

```
10
A's m1() method
20
A's m2() method.
```

Program 4:

Structure:




```
//Program
interface I1
{
 int a=10;
 void m1();
}
 interface I2
{ int b=20;
  void m2();
 }
 class A implements
 { int c=30
   void m3()
   {
   System.out.println("A's m3() method");
   }
}
class B extends A implements I1, I2
  { int d=40;
public void m1()
  { System.out.println("B's m1() method");
```

```
}
public void m2()
{
System.out.println("B's m2() method");
}
}
class Test
{public static void main(String[] args)
{
I1 i1 = (I1) new B();
System.out.println(i1.a);
   i1.m1();
I2 i2 = (I2) new B();
System.out.println(i2.b);
   i2.m2();
B b1 = new B();
System.out.println(b1.a+" "+b1.b+" "+b1.c
                   +" "+b1.d);
   b1.m1();
   b1.m2();
   b1.m3();
}
}
```

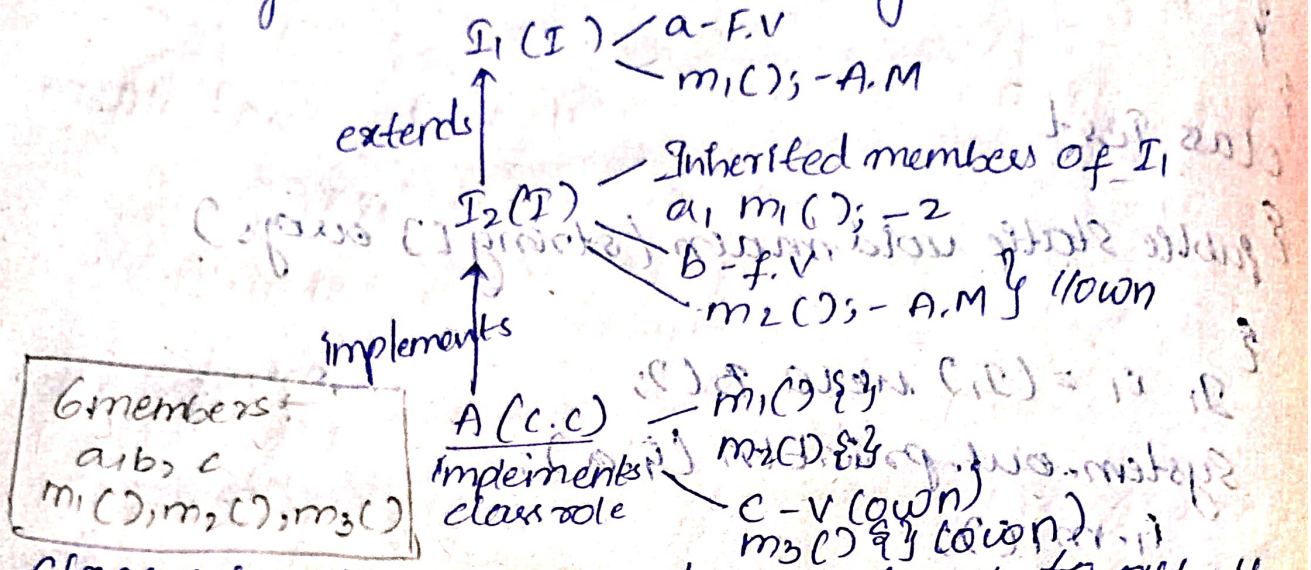
output:

```
10
B's m1() method
20
B's m2() method.

10 20 30 40
B's m1() method
B's m2() method
A's m3() method.
```

Program 5:

One interfaces can extends one interface between two interfaces we use extends keyword.



Class A has to provide implementations to all the abstract methods of the interface (I_2)

1. How do we access the members of interface I_1.

A: upcasting - 2 datatypes

→ requires one concrete class
→ I_1, A

I_1 I_1 = (I_1) new A();

2. How do we access the members of interface I_2

A: upcasting - 2 datatypes

→ requires 1 c.c
→ I_2, A

I_2 I_2 = (I_2) new A();

3. How do we access the members of concrete class A.

A: Using Object

A a_1 = new A()

```
//program on interfaces
interface I1
{ int a = 10;
  void m1(); //overridden method
}
interface I2 extends I1
{ int b = 20;
  void m2(); //overridden method.
}
class A implements I2
{
 int c = 30; //own
 //overriding methods - m1(), m2()
 public void m1()
 {
 System.out.println("A's m1() method");
 }
 public void m2()
 {
 System.out.println("A's m2() method");
 }
 void m3()
 {
 System.out.println("A's m3() method");
 }
}
class Test
{ public static void main(String[] args)
  {
```

must otherwise compile error.

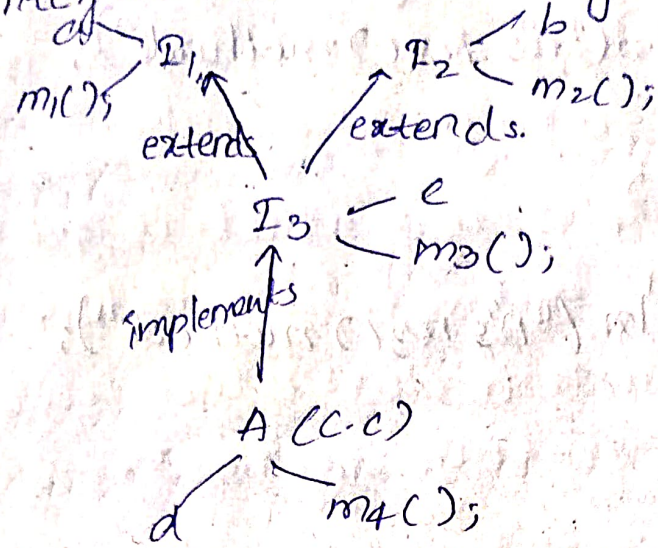
```
//G1: access the members of I1 - upcasting using I1 ref
    I1 i1 = (I1) new A();
    System.out.println(i1.a);
    i1.m1();
//G2: access the members of I2 - upcasting I2 ref A
    I2 i2 = (I2) new A();
    System.out.println(i2.b+" "+i2.b);
    i2.m1();
    i2.m2();
//G3: access the members of concrete class of A-object.
    A a1 = new A()
    System.out.println(a1.a+" "+a1.b+" "+a1.c);
    a1.m1();
    a1.m2();
    a1.m3();
  }
}
```

1. Save : Test.java
2. compile - javac Test.java
3. execute java Test
4. Output : 10

```
A's m1() method
10 20
A's m1() method
A's m2() method
10 20 30
A's m1() method
A's m2() method
A's m3() method.
```

Program 6:

one interface can extend any number of interfaces.




```
//program
interface I1
{ int a=10;
  void m1();
}
interface I2
{ int b=20;
  void m2();
}
interface I3 extends I2,I1
{ int c=30;
  void m3();
}
class A implements I3
{ int d=40;
Public void m1()
{
System.out.println("A's m1() method");
}
```

```
    public void m2()
    {
    System.out.println("A's m2() method");
    }
    public void m3()
    {System.out.println ("A's m3() method");
    }
}
class Test
{
public static void main (String[] args)
{
  I1 i1 = (I1) new A();
 System.out.println (i1.a);
     i1.m1();
 I2 i2 = (I2) new A();
 System.out.println (i2.a);
     i2.m2();
 I3 i3 = (I3) new A();
 System.out.println (i3.a+" "+i3.b+" "+i3.c);
      i3.m1();
      i3.m2();
      i3.m3();
  A a1 = new A();
 System.out.println (a1.a+" "+a1.b+" "+a1.c+" "
      a1.m1();
      a1.m2();
      a1.m3();
}
```

output:

10
A's m1() method
10 20
A's m2() method
10 20 30
A's m1() method
A's m2() method
A's m3() method
10 20 30 40
A's m1() method
A's m2() method
A's m3() method.

⇒ After compilation for interfaces also dot class files are created like classes.

javac Test.java

I1.class I2.class A.class Test.class

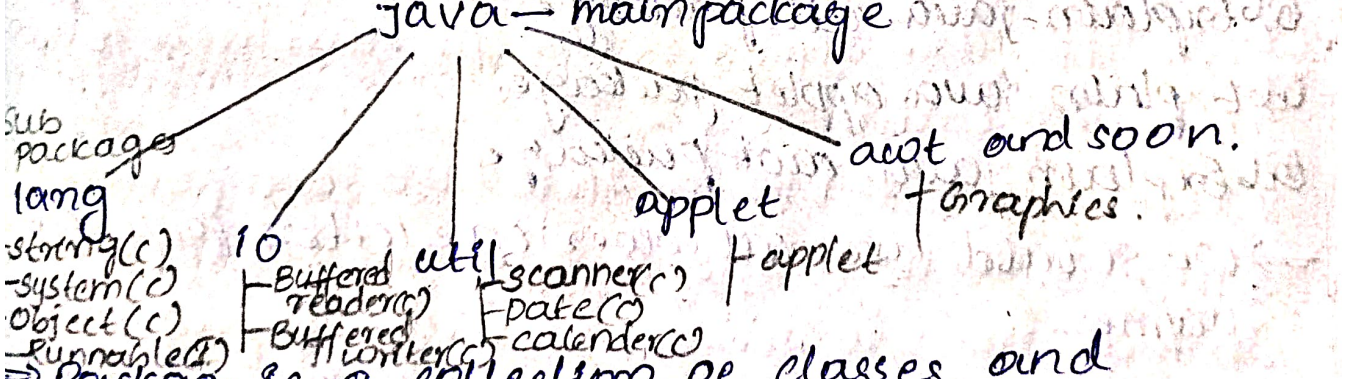
⇒ JVM uses these dot class files for executing interface programs.

Packages:

In java packages are of two types of

1. User defined packages
2. Predefined packages.

Pre-defined packages Hierarchy:



⇒ Package is a collection of classes and interfaces.

⇒ String is a predefined class which is available in java.lang package.

⇒ Runnable is a predefined interface which is available in java.lang.package

⇒ BufferedReader is a predefined class which is available in java.io package

⇒ Scanner is a predefined class which is available in the predefined package java.util.

⇒ applet is a predefined class which is available in the predefined package java.applet.

⇒ Graphics is a predefined class which is available in the predefined package java.awt.

Programs on predefined packages.

//Bufferreaderclass

To write this program we must import java.io.*

⇒ The default package is java.lang

⇒ Therefore for all the java programs the default statement is import java.lang.*

⇒ By default it is available to all java programs

@ Explain java.lang package.

@ Explain java.io package

@ Explain java.util package.

@ Explain java.applet package.

@ Explain java.awt package.

⇒ We should use all lower case letters for package names.

⇒ We can write our own packages like Pre-defined packages. These packages are called user-defined packages.

Syntax:

keyword → package packagename; → Identifier

Classes

Interfaces

//user defined package.

Ex:
```
Package pack1;
class A //concrete class
{
==
}
abstract class B //abstract class
{
==
}
Interface I1 //interface.
{
==
}
```

⇒ In this example the package pack1 ~~contains~~ is a collection of two classes and one interface

Structure of a Java program:

⇒ There should be atmost one (0 or 1) package statement in a java program.

⇒ The package statement must be the non comment statement

⇒ After package statement we can write any number of import statements.

⇒ After import statements we can write any number of classes and interfaces.

Ex1: Package pack1;
Package pack2; C.E atmost 1 package.

⇒
```
package pack1;
import java.lang.*;
import java.io.*;
import java.util.*;
class A
{
```

```
}
abstract class B
{
==
}
interface I1
{
==
}
```

Ex 2:

```
import java.util.*;
import java.lang.*;
class A
{
==
}
class B
{ == }
interface I1
{
==
}
```

Rules for packages:

⇒ There should be atmost one public class in a java package program (or)

⇒ When multiple classes are there in a java package program, there should be only one public class in that program and the name of the program or file must be same as the name of the public class

Java package programs:

1. Single package
   - P1: with one public class
   - P2: with two/more than two public classes
   - P3: with one public class and any number of default classes

P4 : with only default classes.

2. Sub packages :- java.lang
   main package, sub package

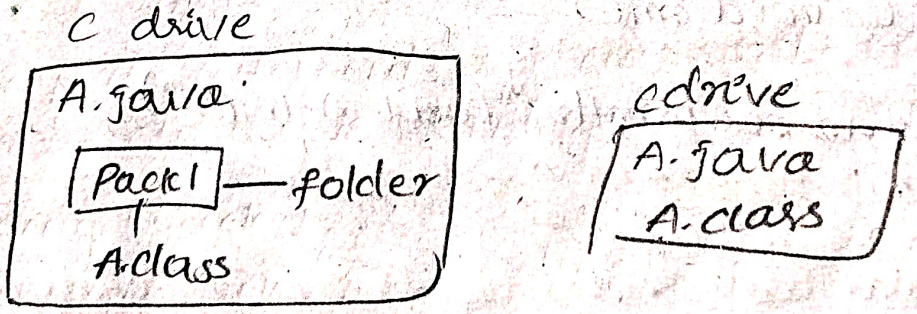
   pack1. pack
   main, sub

3. importing one package into another package.

Q) write a java program using single (or) one package with one public class.

```
// Package program-Basic
package pack1;
Public class A
{ public static void main(String[] args)
  { System.out.println("Package Program");
  }
}
```

1. Save: A.java.
2. compile: javac -d . A.java
   ↳ it directs all the .class file into
3. execute: java pack1.A directors with the name pack1



Output: Package program.

Packages (3 concepts)

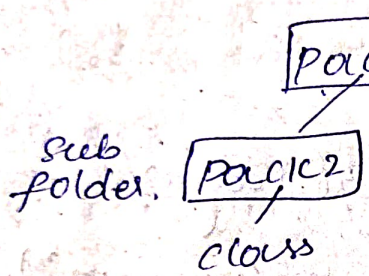
1. single package
pack1
(1 package)

2. subpackages
(2 packages)
pack1 - main package

3. importing one package into another package
(2 packages)

java.lang } predefined.
java.io

pack1.pack2 - user defined.
main package
sub package.



3. Importing one package into another package.

⇒ For this concept two packages are required.

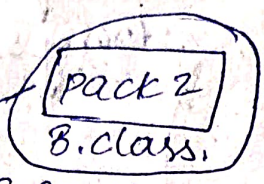
⇒ only public classes of one package are imported to another package.

```
package pack1;
public class A
{
public void m1()
{
System.out.println("Pack1-A-m1()");
}
}
```

Save: A.java.

```
Package pack2;
import pack1.*;
//import pack1.A;
//class A of pack1 is imported
                    available here
```

```
class B
{public static void main (String[] args)
 {A.a1 = new A();
  a1.m1();
 }
}
```



save: B.java.

compile: java -d . B.java

execute: java pack2.B

output: pack1-A-m1().

Q) Explain String class in java with example program.

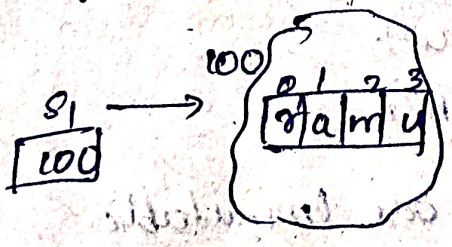
String is a predefined class which is available in java.lang package.

java.lang package is a default package.

⇒ Objects for string class

There are 2 ways to create string class object.

1. String s1 = "ramu";
2. String s1 = new String ("ramu");



⇒ String is a sequence of characters.

⇒ String objects are immutable

⇒ Once we create a string object we cannot do any modifications.

⇒ If you do any modifications separate memory is created to store the string that is the drawback of string object.

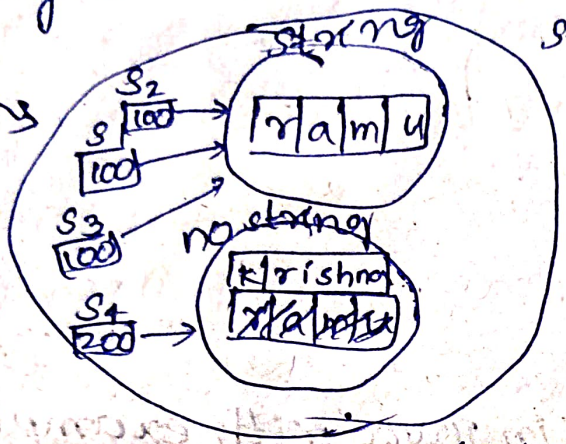
String s_1 = "ramu";

String constant pool memory.

String s_2 = "ramu";

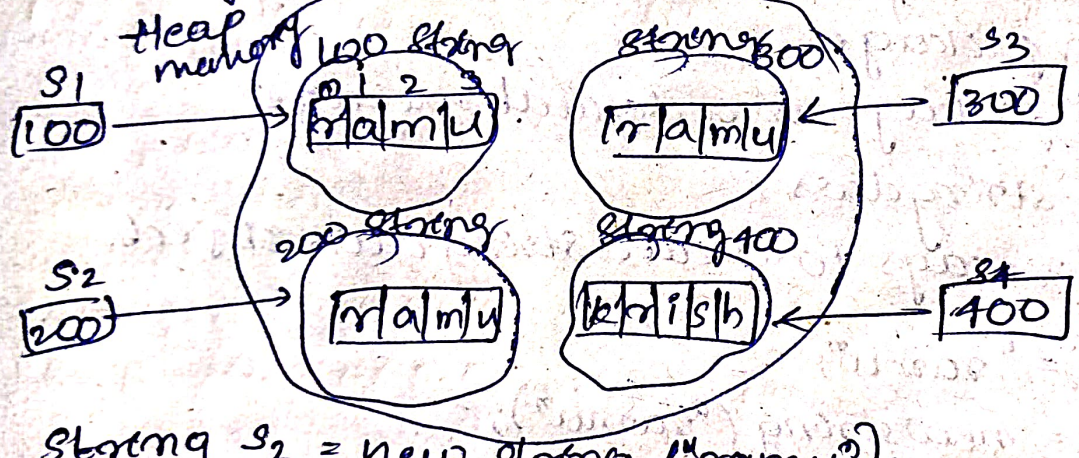
String s_3 = "ramu";

String s_4 = "krishna";



2nd way: using new operator

String s_1 = new String ("ramu");



String s_2 = new String ("ramu");

String s_3 = new String ("ramu");

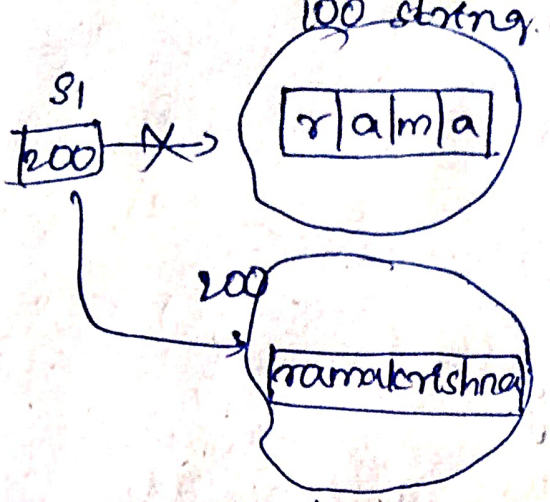
String s_4 = new String ("krish");

Q) Explain why string objects are immutable.

String s_1 = new String ("rama")

s_1 = s_1 + "krishna";

↳ concatenation operator



So string objects are immutable, modification cannot be done in same memory.
To overcome this drawback of string we can go for string buffer or string builder objects are mutable.
Mutable means in the same memory modification is done.

Q) write a java program using string class.
String class functions:
1. length() - to know the length of the string.
2. charAt (int index) - return the particular character in that particular index.
3. toupper() - uppercase letters.
4. to lower() - coverse letters.

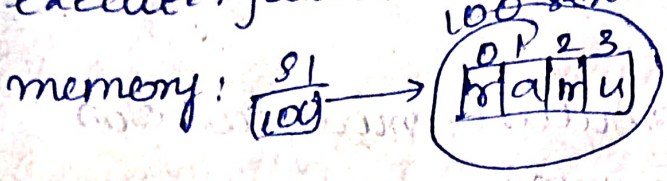
```
//program
class Test
{
public static void main (String[] args)
{String s1 = new String ("rama");
System.out.println (s1.length());
System.out.println (s1.charAt(2));
```

System.out.println(s1);
}}

Save: Test.java

compile: javac Test.java

execute: java Test

memory: 

output: 4
m
ramu

Nested inner classes or Innerclasses.

⇒ In java inside one class we can write one more class

Ex: class Outer //outerclass
{
class Inner // Inner class.
{
} //end of I.C
} //end of O.C.

Programs on Innerclasses:

1. class Outer
{
public class Inner
{ public void m1()
{ System.out.println("Inner class - m1() method");
}
}
public static void main (String[] args)
{ // Inner class object

```
Outer.Inner i = new Outer().new Inner();
i.m1();
}
}
```

save: Outer.java
compile: javac outer.java
execute: java Outer
Output: Inner class - m1() method.

⇒ In the above program main method is written inside the outer class.

//Program 2
We will write the main method in separate class Test. (We are not writing the main method in Outer class).

```
class Outer
{ public class Inner
  { public void m1()
    { System.out.println("Inner class-m1() method");
    }
  }
}
class Test
{ public static void main(String[] args)
  { //Inner class object to access members of Inner
    class
    Outer.Inner i = new Outer().new Inner();
    i.m1();
  }
}
```

Save: Test.java

compile: javac Test.java

execute: java Test.

output: Inner class-m1() method.

Q) Explain access-modifiers or specifiers.

In java there are four access modifiers. They are

1. Private - keyword
2. <default> - not keyword.
3. protected - keyword.
4. public - keyword.

	Private	<default>	protected	Public
1. within the same class	Yes	Yes	Yes	Yes
2. within the same package by both subclasses & non-subclasses	NO	Yes	Yes	Yes
3. another package by only subclasses	NO	NO	Yes	Yes
4. another package by both sub classes and non-subclasses	NO	NO	NO	Yes

1. private members (variables & methods) of a class are accessable only within the same class. and not accessed outside the class.

2. <default> members of a class are accessed within the same package. by both subclasses and non-sub classes of the same package.

3. protected members of a class can be accessed with in the same package by both sub classes and non-sub classes and also outside the package by only subclasses. Cannot be accessed by ~~same~~ non-subclasses of another package

4. Public members of a class can be accessed within the same package by both subclasses and non-subclasses and also another package by both subclasses and non-subclasses

⇒ In java for variables and methods we can use all the four access modifiers that is private, <default>, protected, public

⇒ For classes we can use only 2 access modifiers they are <default> and public.

⇒ For classes we cannot use private and protected

private members (variables & methods) of a class are accessable only within the same class and not accessed outside the class.

<default> members of a class are accessed within the same package by both subclasses and non-sub classes of the same package.

protected members of a class can be accessed within the same package by both sub classes and non-sub classes and also outside the package by only subclasses. cannot be accessed by non-subclasses of another package.

public members of a class can be accessed within the same package by both subclasses and non-subclasses and also another package by both subclasses and non-subclasses

⇒ In java for variables and methods we can use all the four access modifiers that is private, <default>, protected, public

⇒ For classes we can use only 2 access modifiers they are <default> and public

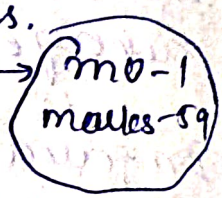
⇒ For classes we cannot use private and protected

Write a java program using the access modifiers.

Private members of a class can be accessed only within the same class. Outside the class they are not visible.

```
// private access modifier within the same class.
class student
{
private int rno=1;
```

```
Private int marks = 59;
private void display() //private method
{
 System.out.println("rno is" + rno);
 System.out.println("marks are" + marks);
}
}
class Test
{
Public static void main(String[] args)
 {
  Student s1 = new student();
  System.out.println(s1.rno+" "+s1.marks); //compile error
  s1.display(); // compiler error.
//To use this we have to not declare the method as
 private (display method) in Student class.
 }
}
```




```
Olp: rno is 1
     marks are 59.
```

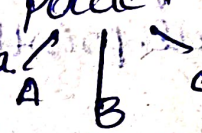
Default members of a class can be accessed by sub classes and non sub classes of same package.

- To explain default access modifier, we need to take one package.

```
//Default access modifier.
package pack1;
class A
{ int a=10, b=20; //a & b are default access modifier
  void display() //default method.
  {
   System.out.println(a+" "+b);
  }
}
```

```
class B extends A //subclass of A
{public static void main(String [] args)
 {A a1 = new A();
  System.out.println(a1.a+" "+a1.b);
 }
}
class C //non-sub class of A.
{public static void main(String [] args)
 {A a2 = new A();
  a2.display();
  System.out.println(a2.a+" "+a2.b);
 }
}
```

package
all default classes — one public rem default



save: A.java (or) B.java (or) C.java

compiler: javac -d A.java

execute: java pack1.B
(or)
java pack1.C

Protected members of a class can be accessed by sub classes and non sub classes of the same package and also by the sub classes of other package.

-To explain protected access modifier, we need to take 2 packages.

```
Package pack1;
public class A // public is to import classes
{
 protected int a=10, b=20;
 protected void display()
 {
  System.out.println(a+" "+b)
 }
}
```

```
class B extends A
{
public static void main (String [] args)
 {
  A a1 = new A ();
  System.out.println (a1.a + "  " + a1.b);
  a1.display ();
 }
}
class C
{
public static void main (String [] args)
 {
  A a2 = new A ();
   a2.display ();
  System.out.println (a2.a + "  " + a2.b);
 }
}
package pack2;
import pack1.A;
// class A of pack1 is available here.
class D extends A // subclass of A in another Package
{
public static void main (String [] args)
 {
  A a3 = new A ();
  System.out.println (a3.a + "  " + a3.b);
  a3.display ();
 }
}
```

Public members of a class can be used by sub-classes and non-sub classes of the same package and also by subclasses and non-subclasses by another package.

```
package pack1;
public class A
{public int a=10, b=20;
 public void display()
 {System.out.println(" a+" " +b);
 }
}
class B extends A
{public static void main(String[] args)
 {A a2 = new A();
  a2.display();
  System.out.println(a2.a+" " +a2.b);
 }
}
Package pack2;
import pack1.A;
class D extends A
{public static void main(String[] args)
 {A a3=new A();
  System.out.println(a3.a+" "+a3.b);
  a3.display();
 }
}
class E
```

```
{
public static void main (String []args)
{
    A a4 = new A();
    System.out.println(a4.a+" "+a4.b);
    a4.display();
}
}
```