Chapter - 5

Inheritance

Inheritance

Inheritance is the process of creation of new things from already existing ones. The advantage of this feature is reusability and time saving.

For example:

```
class A
{
        public  void sum(...)
        {
                :
        }
        public  void sub(...)
        {
                :
        }
}
```

The above class can perform two operations such as **sum()** and **sub()**. Let say, to write one method it takes one minute therefore for two methods it will take two minutes.

Now we would like to write another class which should perform three operations such as **sum()**, **sub()**, **mult()**. Therefore the class is.

```
class B
{
        public void sum(...)
        {
                :
        }
        public void sub(...)
        {
                :
        }
        public void mult(...)
        {
                :
        }
}
```

The above class takes three minutes to write three methods. Therefore the dis-advantages of above class are.

1) **Duplication of code:** The class A contains two operations and the same operations are written in class B which is duplication of code.
2) **Time:** Writing the new classes with duplication of methods is a wastage of time.
3) **Size:** Because of duplication of code the size of program increases as a result it occupies more memory and program executes slower.

To overcome from the above drawbacks inheritance concepts can be implemented.

For example consider the following:

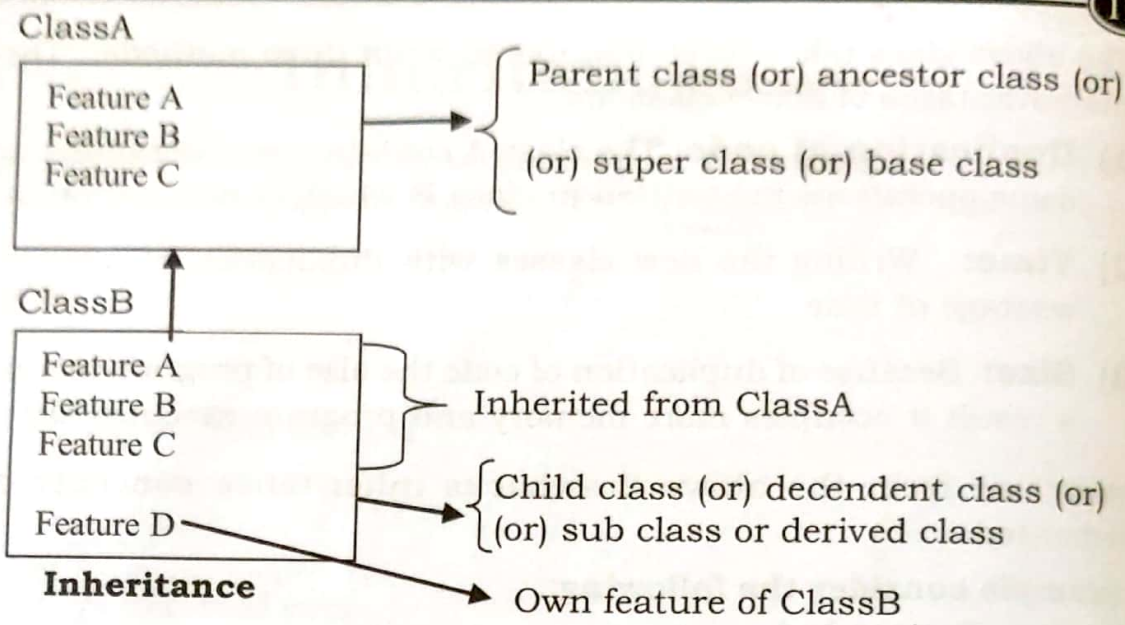
```
class B extends A
{
        public void mult()
        {
            :
        }
}
```

The **class B** extends from **class A** therefore all the operations from **class A** (**sum()** and **sub()**) are inherited into **B**. Therefore **class B** is written with only one operation **mult()**. Now, the **class B** can perform three operations such as **sum()**, **sub()** and **mult()**. The **class B** takes only one minute to complete the class.

Advantages of Inheritance:

1) **Re-usability:** The code or methods written in one class can be used in another class without rewriting.
2) **Elimination of duplication of code:** Because of re-usability of code inheritance eliminates the duplication of code.
3) **Time:** In less time new classes can be developed using existing classes.
4) **Size:** The size of program is reduced as a result programs execute faster.

Inheritance Definition: Creation of new classes from already existing classes is known as **inheritance**.



Inheritance

In the above diagram, **ClassB** inherited from **ClassA** therefore all the features of **ClassA** are inherited into **ClassB** and **ClassB** can have its own features.

ClassA is known as **parent class** (or) **ancestor class** (or) **base class** (or) **super class** and ClassB is known as **child class** (or) **decendent class** (or) **derived class** (or) **sub class**.

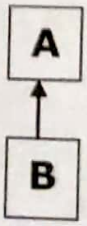
In Java, parent class is called as **super class** and child class is called as **sub class**.

Forms of inheritance

Based on level and relation ship among the classes the inheritances are divided into six forms. They are

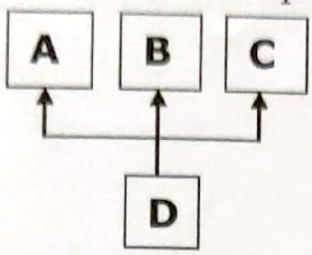
1) Single Inheritance
2) Multiple Inheritance
3) Hierarchal Inheritance
4) Multi-Level Inheritance
5) Hybrid Inheritance
6) Multipath Inheritance

1) Single Inheritance: Creation of a class from only one super class is known as single Inheritance.



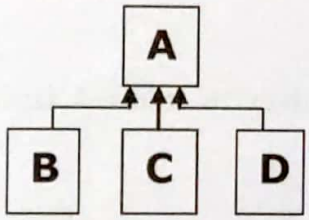
Single Inheritance

2) **Multiple Inheritance:** Creation of a class from two or more super classes is known as multiple inheritance.



Multiple Inheritance

3) **Hierarchial Inheritance:** Creation of several classes from single super class is known as hierarchal Inheritance.

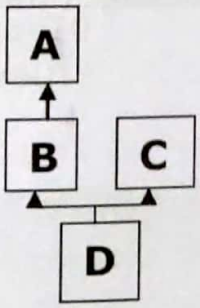


Hierarchial Inheritance

4) **Multi-Level Inheritance:** Creation of a sub-class from another sub-class is known as Multi-Level Inheritance.

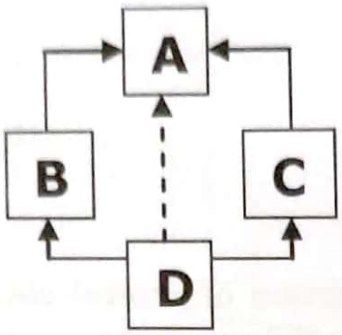
Multi-Level Inheritance

5) **Hybrid Inheritance:** Combination of forms of inheritances is known as hybrid Inheritance.



Hybrid Inheritance

6) **Multipath-Inheritance:** Creation of a class from other super classes which are derived from common super class is known as multipath Inheritance.



Multipath Inheritance

The dotted line indicates that D inherits from A indirectly.

Syntax of Inheritance:

```
class subclassname extends superclassname
{
        :
}
```

In Java, **extends** keyword is used to inherit a sub class from super class.

Access specifiers in inheritance:

The **private** members of super class are not inherited into sub class, where as **default**, **public** and **protected** members inherited. Therefore private of super class cannot be access inside or outside of subclass where as **default**, **public** and **protected** of super class can be accessed inside or outside of subclass, but outside the subclass using object name.

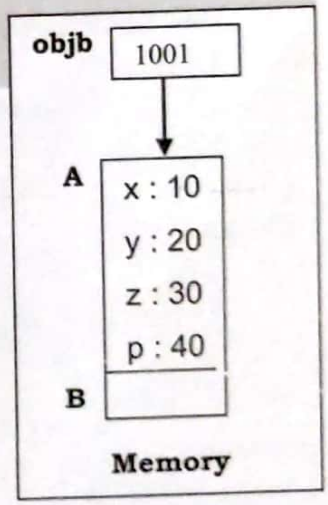
We understand the difference between **default**, **public** and **protected** in the packages chapter because differences can be identified when inheritances are used in combination with packages therefore we postpone till that time.

Example:

Bin>edit inh1.java

```
class A
{
    private int x=10;
    public int y=20;
```

```
    protected int z=30;
    int p=40;  //default access
}
class B extends A
{
    public void show()
    {
        //System.out.print("\n private x="+x);//error
        System.out.print("\n public y="+y);
        System.out.print("\n protected z="+z);
        System.out.print("\n default p="+p);
    }
}
class inh1
{
    public static void main(String argv[])
    {
        B objb=new B();
        objb.show();
    //System.out.print("\nprivate objb.x="+objb.x);
                                        //error
        System.out.print("\n public objb.y="+objb.y);
        System.out.print("\n private objb.z="+objb.z);
        System.out.print("\n default objb.p="+objb.p);
    }
}
```



Bin>javac inh1.java
Bin>java inh1

public y=20
protected z=30
default p=40
public objb.y=20
protected objb.z=30
default objb.p=40

Explanation:

In the above program, **class B** inherited from **class A** therefore the private variable **x** of **A** class is not inherited into **B** class as a result **x** can't access inside or outside of **B** class.

The public, protected and default variables **y**, **z** and **p** of **A** class inherited into sub class **B**, therefore **y**, **z** and **p** can be accessed inside or outside the **B** class.

When an object is created to sub class, the memory of super and sub classes is allocated as shown in above memory.

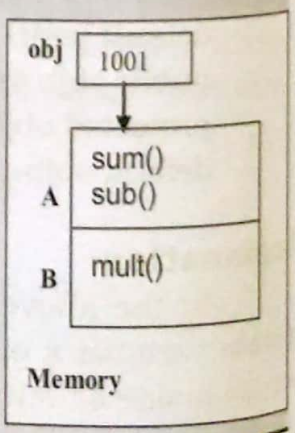
Note:

1) Re-execute the above program without comments, the compiler generates the errors.

A program on Inheritance

Bin>edit inh2.java

```
class A
{
    public void sum(int x, int y)
    {
          System.out.print("\n sum=" + (x+y) );
    }
    public void sub(int x, int y)
    {
          System.out.print("\n sub=" + (x-y));
    }
}
class B extends A
{
    public void mult(int x, int y)
    {
          System.out.print("\n mult=" + (x*y));
    }
}
class inh2
{
      public static void main(String argv[])
      {
          B obj=new B();
          obj.sum(10,20);
          obj.sub(10,5);
          obj.mult(5,3);
      }
}
```



```
Bin>javac inh2.java
Bin> java inh2
        sum=30
        sub=5
        mult=15
```

Explanation:

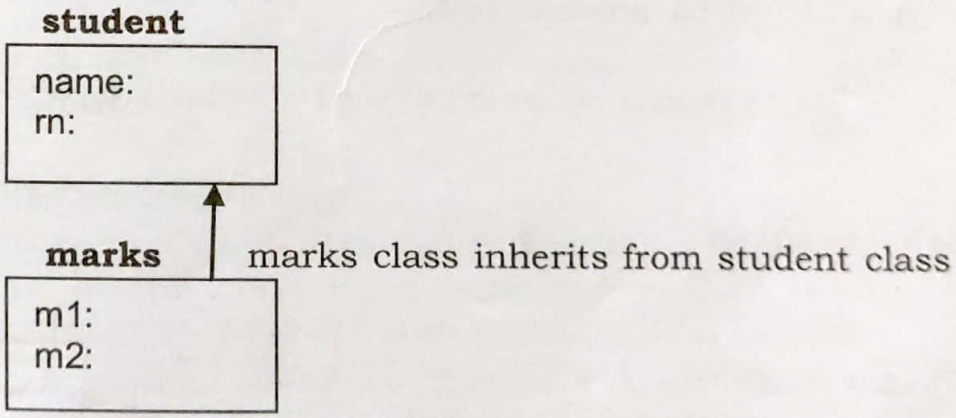
In the above program, **class B** is inherited from **class A** therefore all the methods of **A** are inherited into **B** and **B** class contains three methods(**sum()**, **sub()**, **mult()**). As the methods are public, they can be accessed outside the class using object name.

In the main(), the statement **B obj=new B();** creates the object **obj** of **class B** and the object contains the memory of **A** and **B** classes as shown in memory.

Using the sub class object **obj** we can access the methods of super and sub classes provided they are public and we get the output as shown above.

Single Inheritance:

Let us write a program on single Inheritance.
Understand the following image.

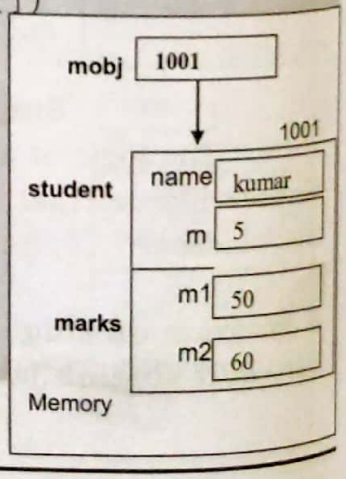


Single Inheritance

The logic of writing the inheritance programs is simple i.e. write the classes like normal classes and establish the inheritance relation between the classes.

A program on single inheritance
Bin>edit singinh.java

```
class student
{
    private String name;
    private int rn;
    public void setstudent(String n,int r)
    {
          name=n;
          rn=r;
    }
    public void showstudent()
    {
          System.out.print("\nName="+name+"\tRn="+rn);
    }
}
class marks extends student    //establishing relationship
    //between marks and student
{
    private int m1,m2;
    public void setmarks(int s1,int s2)
    {
          m1=s1;
          m2=s2;
    }
    public void showmarks()
    {
          System.out.print("\nm1="+m1+"\tm2="+m2);
    }
}
class singinh
{
      public static void main(String argv[])
      {
          marks mobj=new marks();
          mobj.setstudent("kumar",105);
          mobj.setmarks(50,60);
          mobj.showstudent();
          mobj.showmarks();
      }
}
```



Bin>javac singinh.java
Bin>java singinh

Name=kumar Rn=5
m1=50 m2=60

Explanation:

In the main, **mobj** is created to **marks** class, but **marks** is inherited from **student** therefore **mobj** is allocated with the memory of student and **marks** classes as shown in above memory. Using **mobj,** methods of **student** and **marks** classes are called.

The statement **mobj.setstudent("kumar",105);** calls the **setstudent(String n,int r)** method of **student** class which sets **name=kumar** and **rn=5**.

The statement **mobj.setmarks(50,60);** calls the **setmarks(int s1,int s2)** method of **marks** class which sets **m1=50** and **m2=60**.

The statement **mobj.showstudent()** and **mobj.showmarks()** displays the details of **student** and **marks** classes respectively as shown in above output.

Overriding

Overriding is the concept of hiding a member of super class by the member of sub class. The member may be either instancevariable or method.

Overriding is of two types

1) Instancevariable overriding
2) Method overriding

1. Instancevariable overriding

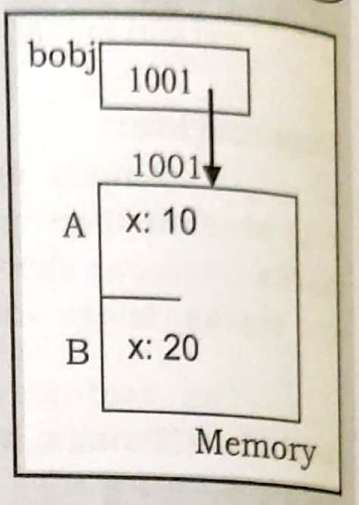
In the inheritance, if the instancevariable of super class and instancevariable of sub class have same names then it is said to be instancevariable overriding i.e. the instancevariable of sub class overrides (hides) the instancevariable of super class as a result inside the sub class only the instancevariable of sub class is accessed but not super class.

Example:

Bin>edit override1.java

```
class A
{
    public int x=10;
}
```

```
class B extends A
{
    public int x=20;
    public void showvalues()
    {
          System.out.print("\n x="+x);
          System.out.print("\n x="+x);
    }
}
class override1
{
    public static void main(String argv[])
    {
          B bobj=new B();
          bobj.showvalues();
    }
}
```



Bin>javac override1.java

Bin>java override1

x=20
x=20

Explanation:

In the above program, super class **A** and sub class **B** contains a variable with same name called **x**. The **x** of **A** class is public therefore it can be accessed in **class B**. Since the super and sub classes variables are having same names, in the sub class only the variable of sub is accessed but not the super class because sub class is given first preference.

In the main(), the statement **B objb=new B();** creates objb object and it contains memory of A and B classes as shown in memory. The statement **objb.showvalues();** calls **showvalues()** method of subclass **B** and it prints value of **x** as **20** because inside the subclass first preference is given to subclass variables therefore super class **A** variable **x** can not be accessed. This is the dis-advantage of instance variable overriding and we get the output as shown above.

Drawback of overriding:

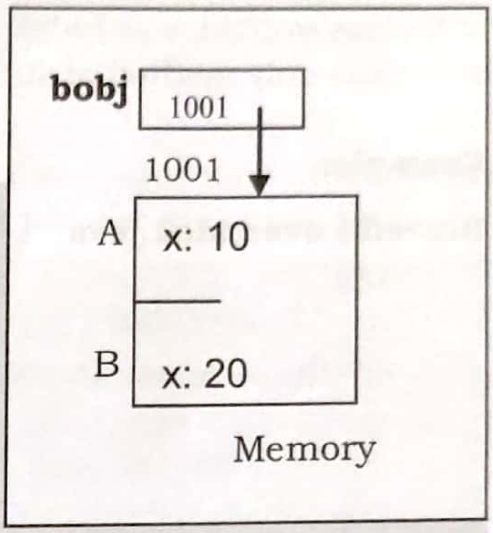
When super class variable and sub class variable have same name then inside the subclass only instance variables of subclass is accessed but not super.

Solution for overriding is *super* keyword

In java, **super** keyword can be used to access the super class instance variables in the sub class explicitly when super and sub classes variables have same names.

Syntax:

```
super.instancevariable
```



Example:

Bin>edit super1.java

```
class A
{
    public int x=10;

}
class B extends A
{
    public int x=20;
    public void showvalues()
    {
          System.out.print("\n A.x="+super.x);
          System.out.print("\n B.x="+x);
    }
}
class super1
{
      public static void main(String argv[])
      {
          B bobj=new B();
          bobj.showvalues();
    }
}
```

Bin>javac super1.java

Bin>java super1

A.x=10

B.x=20

Explanation:

In the above program, **super.x** represents **x** variable of super class and therefore **10** is printed where as **x** represents variable of sub class and it prints **20.**

Note:

1. The **super** keyword should be used only in the methods of sub class.

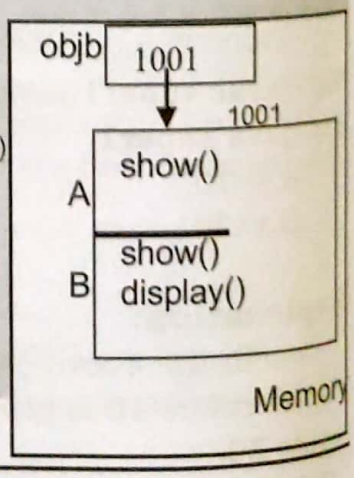
Method overriding

If the methods of super class and sub class have same name along with their signatures then it is said to be method overriding i.e. method of sub class overrides or hides the method of super class as a result inside the sub class only method of sub class is accessed but not super class.

Example:

Bin>edit override2.java

```
class A
{
    public void show()
    {
          System.out.print("\n show of A class…");
    }
}
class B extends A
{
    public void show()
    {
          System.out.print("\n show of B class…");
    }
    public void display()
    {
          show();      // calls show() of B
          show();      // calls show() of B
    }
}
class overrides2
{
    public static void main(String argv[])
    {
          B objb=new B();
          objb.display();
    }
}
```



Bin>javac override2.java

Bin>java override2

show of B class...
show of B class...

Explanation:

In the above program, from the display method the **show()** is called twice. In these two cases the **show()** method of sub class(**B**) is called but not from super class(**A**), because sub class method is given first preference.

Drawback of method overriding:

If the method of super and sub classes have same name then inside the sub class only method of sub class is called but not the super class.

Solution for method overriding is *super* keyword

In java, **super** keyword is used to access the super class methods explicitly in the sub class when super and sub classes methods have same name.

Syntax:

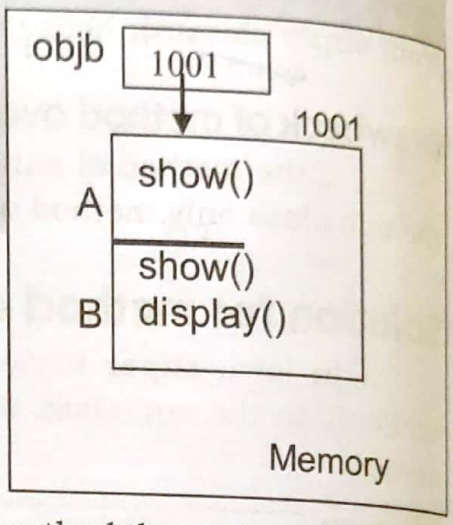
```
super.methodname( [arg1,arg2, . . .] );
```

Example:

Bin>edit super2.java

```
class A
{
    public void show()
    {
          System.out.print("\n show of A class…");
    }
}
class B extends A
{
    public void show()
    {
          System.out.print("\n show of B class…");
    }
    public void display()
    {
          show();      // calls show of B
```

```
        super.show();          // calls show of A
    }
}
class overrides2
{
    public static void main(String argv[])
    {
        B objb=new B();
        objb.display();
    }
}
```



Bin>javac override2.java
Bin>java override2

show of B class...
show of A class...

Explanation:

In the above program, from the **display()** method the statement **show();** calls **show()** method of sub class **B** and **super.show();** calls **show()** of super class **A**.

Constructors in Inheritance

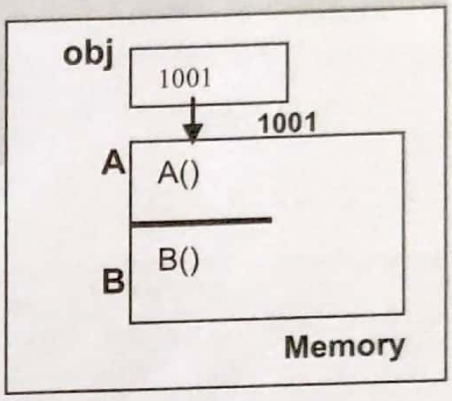
When an object is created to the sub class the memory of super and sub classes are allocated. Therefore when and object is created the constructors of both the classes are executed but constructor of super class is executed first and then the constructor of sub class.

Example:

Bin>edit inhcons1.java

```
class A
{
    public A()
    {
        System.out.print("\n constructor of A...");
    }
}
class B extends A
{
    public B()
    {
```

```
        System.out.print("\n constructor of B...");
    }
}
class inhcons1
{
    public static void main(String argv[])
    {
        B obj=new B();
    }
}
```



Bin>javac inhcons1.java

Bin> java inhcons1

constructor of A...

constructor of B...

Explanation:

In the main, when **obj** is created for **B** class, the constructor of **B** is called first and from the **B** class constructor, the constructor of super class **A** is called. After execution of **A** constructor the control comes back to **B** constructor and **B** constructor executes finally the control returns back to main().

In this way when an object is created to sub class, the constructor of sub class is called first and then super class. But super class constructor is executed first and then sub class constructor.

The default constructor of super class is called implicitly, in the above program the super class constructor is called implicitly. The super class constructor can be called explicitly using **super()** method.

The *super( )* method

The **super()** method can be used to call the constructor of super class explicitly. It is useful when we want to pass arguments to super class constructor.

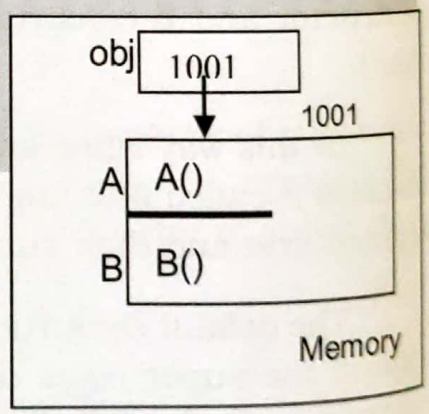
Syntax:

```
super( [argv1, argv2, . . . ] );
```

Example:

Bin>edit inhcons2.java

```
class A
{
    public A()
    {
        System.out.print("\n constructor of A…");
    }
}
class B extends A
{
    public B()
    {
        super();
        System.out.print("\n constructor of B…");
    }
}
class inhcons2
{
    public static void main(String argv[])
    {
        B obj=new B();
    }
}
```



Bin>javac inhcons2.java

Bin> java inhcons2

constructor of A...

constructor of B...

Explanation:

In the main, when **obj** is instantiated to **B** class, the constructor of **B** is called first and from **B** constructor the statement **super();** calls the constructor of **A** super class. After completion of **A** constructor the sub class constructor **B** is executed. In this is way the **super()** method can be used to call the super class constructor explicitly.

Note:

1. The **super()** method should be used only in the constructor of subclass, and it should be the first statement.

When to call super class constructor explicitly?

When we want to pass arguments to super class constructor then it is compulsory to call the constructor explicitly. The arguments required for super and sub class constructors should be passed to sub class constructor and it is the responsibility of

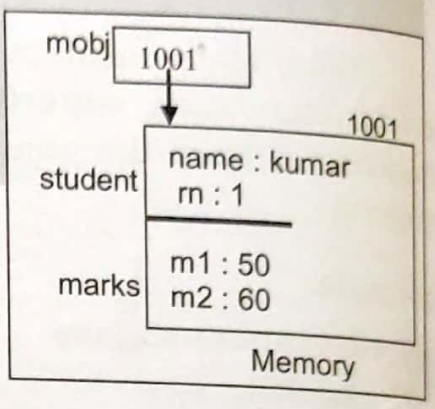
Sub class constructor to pass some of the arguments to super class constructor using **super()** method. Arguments are passed to sub class constructor from the object creation. For example consider the following program.

Example:

Bin>edit inhcons3.java

```
class student
{
    private String name;
    private int rn;
    public student(String s, int r)
    {
          name=s;
          rn=r;
    }
    public void showstudent()
    {
          System.out.print("\nName="+name+"\tRn="+rn);
    }
}
class marks extends student
{
    private int m1,m2;
    public marks(String s,int r,int s1,int s2)
    {
          super(s,r);
                //passing arguments to super class contructor
          m1=s1;
          m2=s2;
    }
    public void showmarks()
    {
          System.out.print("\nM1="+m1+"\tM2="+m2);
    }
}
```

```
class inhcons3
{
      public static void main(String argv[])
      {
          marks mobj=new marks("kumar",1,50,60);
          mobj.showstudent();
          mobj.showmarks();
      }
}
```



Bin>javac inhcons3.java
Bin>java inhcons3

Name=kumar Rn=1
M1=50 M2=60

Explanation:

In the main, when **mobj** object is created to marks class in the memory, it contains instance variables of super class (**student**) and sub class (**marks**). As the **mobj** is created, it calls the constructor **marks(String s, int r, int s1,int s2)** and argument values **("kumar", 1,50,60)** are passed to **(s,r,s1,s2)**. From the sub class marks constructor, the statement **super(s,r);** calls the super class constructor **student(String s,int r)** and passes the arguments(**s="kumar"** and **r=1**) to the student constructor **(s,r).**

In the student constructor **s**("kumar") assigns into **name** and **r**(1) assigns to **rn**. After the execution of student constructor the control gets back to the sub class where it assigns **m1** with **s1**(50) and **m2** with **s2**(60).

The statements

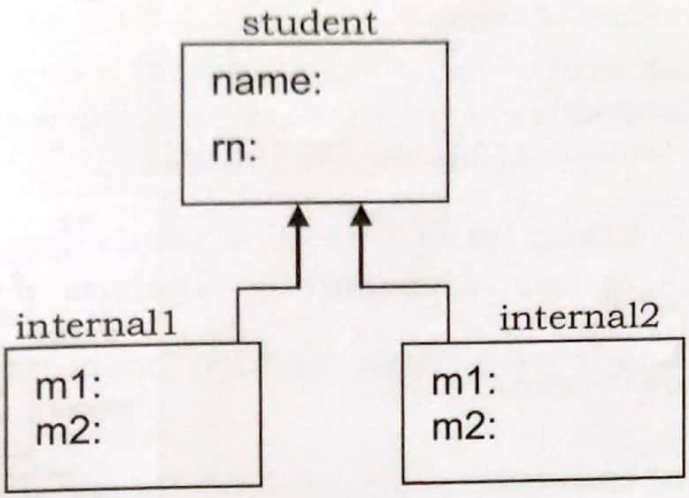
mobj1.showstudent(); ⟶ displays student details
mobj1.showmarks(); ⟶ displays marks details

We get output as shown above.

Hierarchial Inheritance

Creation of several classes from single super class is known as hierarchial inheritance.

Example:



A program implementing above hierarchical inheritance.

Bin>edit hierarinh.java

```
class student
{
    private String name;
    private int rn;
    public void setstudent(String n,int r)
    {
      name=n;
      rn=r;
    }
    public void showstudent()
    {
      System.out.print("\nName="+name+"\tRn="+rn);
    }
}
class internal1 extends student
{
    private int m1,m2;
    public void setinternal1(int s1,int s2)
    {
          m1=s1;
          m2=s2;
    }
    public void showinternal1()
    {
          System.out.print("\nm1="+m1+"\tm2="+m2);
    }
}
```

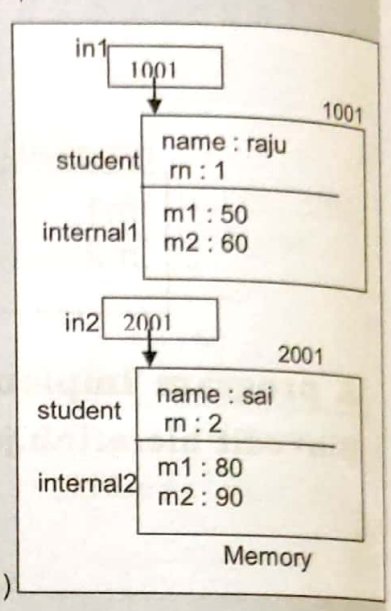
```
class internal2 extends student
{
    private int m1,m2;
    public void setinternal2(int s1,int s2)
    {
          m1=s1;
          m2=s2;
    }
    public void showinternal2()
    {
          System.out.print
          ("\nm1="+m1+"\tm2="+m2);
    }
}
class hierarinh
{
    public static void main(String argv[])
    {
          internal1 in1=new internal1();
          internal2 in2=new internal2();

          in1.setstudent("raju",1);
          in1.setinternal1(50,60);

          in2.setstudent("sai",2);
          in2.setinternal2(80,90);

          System.out.print("\n internal1 details..");
          in1.showstudent();
          in1.showinternal1();

          System.out.print("\n internal2 details..");
          in2.showstudent();
          in2. showinternal2();
    }
}
```



Bin>javac hierarinh.java
Bin>java hierarinh

internal1 details...
Name=raju Rn=1
m1=50 m2=60
iInternal2 details...
Name=sai Rn=2
m1=80 m2=90

Explanation:

In the main, **in1** object is created to **internal1** class. As **internal1** is inherited from **student** class, **in1** object is allocated with memory of **student** and **internal1** as shown in memory above.

When **in2** object is created to **internal2** class. It is also allocated with memory of **student** and **internal2** classes.

in1.setstudent("raju",1); assigns "raju" and 1 to name and rn of in1 object.

in1.setinternal1(50,60) assigns 50 and 60 to m1 and m2 of in1 object.

in2.setstudent("sai",2) assigns "sai" and 2 to name and rn of in2 object.
in2.setinternal2(80,90) assigns 80 and 90 to m1 and m2 of in2 object.

in1.showstudent() prints name and rn of in1 object.
in1.showinternal1() prints m1 and m2 of in1 object.

In2.showstudent() prints name and rn of in2 object.
In2.showinternal2() prints m1 and m2 of in2 object.

We get the output as shown above.

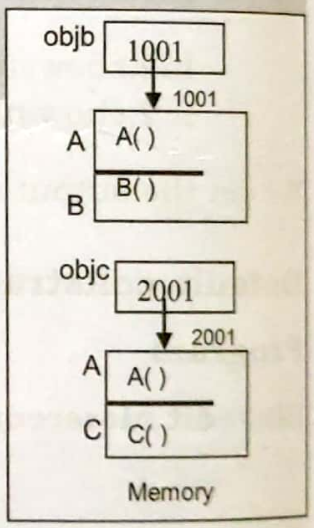
Default constructors in hierarchial inheritance

Program

Bin>edit hierarcons1.java

```
class A
{
    public A()
    {
        System.out.print("\n constructor of A…");
    }
}
class B extends A
{
    public B()
    {
        System.out.print("\n constructor of B…");
    }
}
```

```
class C extends A
{
    public C()
    {
        System.out.print("\n constructor of c…");
    }
}
class hierarcons1
{
    public static void main(String argv[])
    {
          B objb=new B();
          System.out.print("\n ----------");
          C objc=new C();
          System.out.print("\n program terminates");
    }
}
```



Bin>javac hierarcons1.java
Bin>java hierarcons1
constructor of A...
constructor of B...
---
constructor of A...
constructor of C...

Explanation:

In the main, when **objb** is created it calls the constructor of super class **A** and then the constructor of sub class **B**.

When **objc** is created it calls the constructor of super class **A** and then the constructor of sub class **C**.

Parameterized constructors in Hierarchial Inheritance

Program:
Bin>edit hierarcons2.java

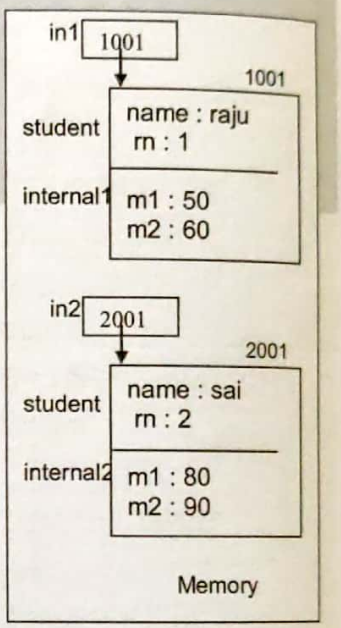
```
class student
{
    private String name;
    private int rn;
```

```
    public student(String n,int r)
    {
          name=n;
          rn=r;
    }
    public void showstudent()
    {
      System.out.print("\nName="+name+"\tRn="+rn);
    }
}
class internal1 extends student
{
    private int m1,m2;
    public internal1(String n,int r,int s1,int s2)
    {
          super(n,r);
          m1=s1;
          m2=s2;
    }
    public void showinternal1()
    {
          System.out.print("\nm1="+m1+"\tm2="+m2);
    }
}
class internal2 extends student
{
    private int m1,m2;
    public internal2(String n,int r,int s1,int s2)
    {
          super(n,r);
          m1=s1;
          m2=s2;
    }
    public void showinternal2()
    {
          System.out.print("\nm1="+m1+"\tm2="+m2);
    }
}
class hierarcons2
{
```

```
    public static void main(String argv[])
    {
        internal1 in1=new internal1("raju",1,50,60);
        internal2 in2=new internal2("sai",2,80,90);

        System.out.print("\n internal1 details...");
        in1.showstudent();
        in1.showinternal1();

        System.out.print("\n internal2 details...");
        in2.showstudent();
        in2.showinternal2();
    }
}
```



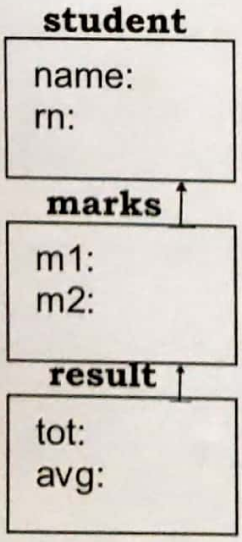
Bin>javac hierarcons2.java
Bin>java hierarcons2

internal1 details...
Name=raju Rn=1
m1=50 m2=60
iInternal2 details...
Name=sai Rn=2
m1=80 m2=90

The above program is similar to parameterized constructors in single Inheritance. So it is left as self explanatory.

Multi-Level Inheritance

Creation of a class from another sub class is known as multi-level Inheritance.



Multi-level Inheritance

Implementation of multi-level inheritance

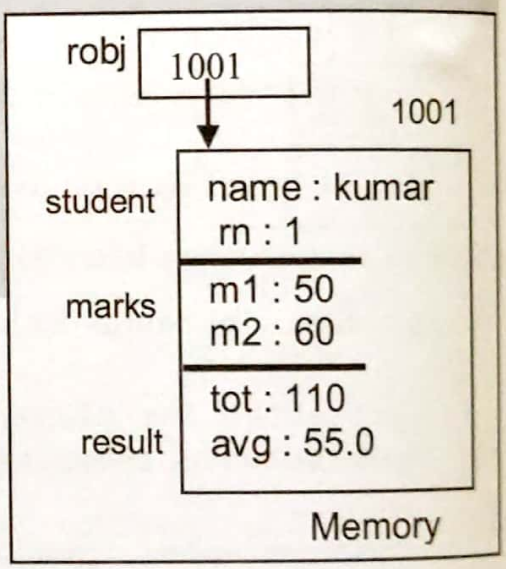
Write all the classes like normal classes and link up them with inheritance concept. The **result** class wants to access **m1** and **m2** of **marks** class to calculate **tot** and **avg**. Therefore **m1** and **m2** should be declared as **protected** because **protected** is inherited in to sub class where as **private** does not.

Program:

Bin>edit mlevel1.java

```
class student
{
    private String name;
    private int rn;
    public void setstudent(String n,int r)
    {
          name=n;
          rn=r;
    }
    public void showstudent()
    {
          System.out.print("\nName="+name+"\tRn="+rn);
    }
}
class marks extends student
{
    protected int m1,m2;
    public void setmarks(int s1,int s2)
    {
          m1=s1;
          m2=s2;
    }
    public void showmarks()
    {
          System.out.print("\nm1="+m1+"\tm2="+m2);
    }
}
class result extends marks
{
    private int tot;
    private double avg;
```

```
    public void calculate()
    {
          tot=m1+m2;
          avg=tot/2.0;
    }
    public void showresult()
    {
          System.out.print("\ntot="+tot+"\tavg="+avg);
    }
}
class mlevel1
{
    public static void main(String argv[])
    {
          result robj=new result();
          robj.setstudent("kumar",1);
          robj.setmarks(50,60);
          robj.calculate();
          robj.showstudent();
          robj.showmarks();
          robj.showresult();
    }
}
```



Bin>javac mlevel1.java
Bin>java mlevel1

Name=kumar Rn=1
m1=50 m2=60
tot=110 avg=55.0

Explanation:

In main(), **robj** object is created to **result** class and **robj** is allocated with the memory of **student**, **marks** and **result** classes as shown in memory.

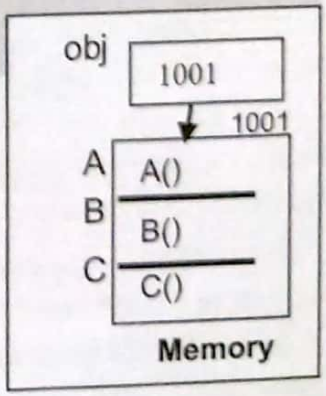
robj.setstudent("kumar",1); ⟶ sets name="kumar" and rn=1
robj.readmarks(50,60); ⟶ sets m1=50 and m2=60
robj.calculate(); ⟶ calculate tot and avg
robj.showstudent(); ⟶ prints name,rn
robj.showmarks(); ⟶ prints m1 and m2
robj.showresult(); ⟶ prints tot and avg.

Constructors in Multi-Level Inheritance:

A program on execution of default constructors in multi-level Inheritence.

Bin>edit mlevelinhcons1.java

```
class A
{
    public A()
    {
        System.out.print("\n constructor of A…");
    }
}
class B extends A
{
    public B()
    {
        System.out.print("\n constructor of B…");
    }
}
class C extends B
{
    public C()
    {
        System.out.print("\n constructor of C…");
    }
}
class mlevelinhcons1
{
     public static void main(String argv[])
     {
         C obj=new C();
    }
}
```



Bin>javac mlevelinhcons1.java
Bin>java multiinhcons1

constructor of A...
constructor of B...
constructor of C...

Explanation:

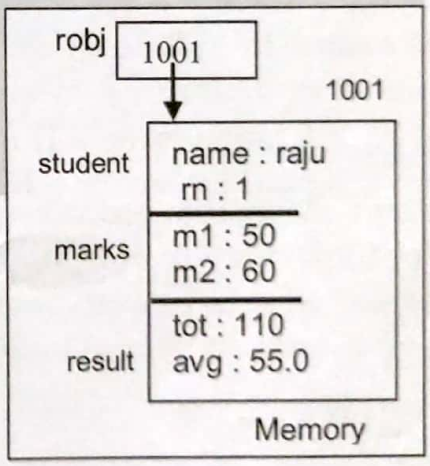
In the main(), when **obj** is created to **C** class, the constructor of **C** is called but **C** inherits from **B,** therefore from **C** class constructor **B** class constructor is called. But **B** inherits from **A,** therefore from **B** class constructor **A** class constructor is called. As **A** class is topmost super class, the constructor of **A** is executed and then gets back to **B** class constructor and after it's execution the control goes to **C** constructor and is executed. We get the output as shown above.

Parameterized constructors in multi-level Inheritance Program

Bin>edit mlevelinhcons2.java

```
class student
{
    private String name;
    private int rn;
    public student(String s, int r)
    {
        name=s;
        rn=r;
    }
    public void showstudent()
    {
        System.out.print("\nname="+name+"\trn="+rn);
    }
}
class marks extends student
{
    private int m1,m2;
    public marks(String s,int r,int s1,int s2)
    {
        super(s,r);
        m1=s1;
        m2=s2;
    }
    public void showmarks()
    {
        System.out.print("\nm1="+m1+"\tm2="+m2);
    }
}
```

```
class result extends marks
{
    private int tot;
    private double avg;
    public result(String s,int r,int s1,int s2,int t,double a)
    {
          super(s,r,s1,s2);
          tot=t;
          avg=a;
    }
    public void showresult()
    {
          System.out.print("\ntot="+tot+"\tavg="+avg);
    }
}
class mlevelinhcons2
{
    public static void main(String argv[])
    {
          result robj=new result("raju",1,50,60,110,55.0);
          robj.showstudent();
          robj.showmarks();
          robj.showresult();
    }
}
```



Bin>javac mlevelinhcons2
Bin>java mlevelinhcons2

name= raju rn=1
m1=50 m2=60
tot=110 avg=55.0

Explanation:

In main(), when **robj** is created, it calls the result class constructor and passes **("raju",1,50,60,110,55.0)** to **result(s,r,s1,s2,t,a)**. In the result class constructor the statement **super(n,r,s1,s2)** calls the marks class constructor **marks(s,r,s1,s2)** by passing **("raju",1,50,60)**. In the marks class constructor the statement **super(n,r)** calls the student class constructor **student(s,r)** by passing **("raju",1)**.

The student class constructor copies **s**("raju") into **name** and **r**(1) into **rn** and then **marks** class constructor assigns **s1**(50) to **m1** and **s2**(60) to **m2**. Finally, the **result** class constructor assigns **t** (110)to **tot** and **a**(55.0) to **avg.**

The statements

```
robj.showstudent();    // prints name and rn
robj.showmarks();      // prints m1 and m2
robj.showresult();     // prints tot and avg
```

Object Delegation and composition

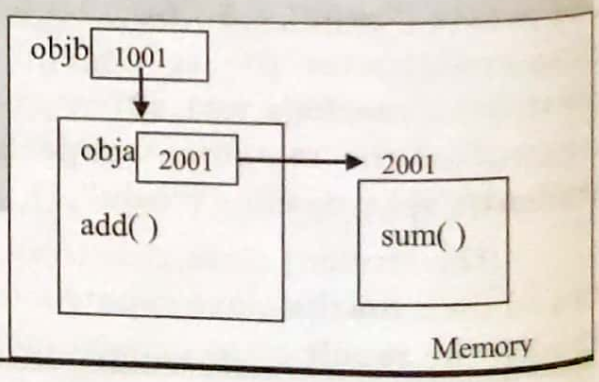
Delegation:

Delegation is the process of bypassing the request (or) operation of an object to another object. It is as powerful as inheritance for re-use. In delegation, objects are involved in handling a request i,e a receiving object delegates operations to its delegate (i.e a subclass by-passing its request to the parent class). Object delegation is an alternative of inheritance.

A program demonstrating object delegation

Bin>edit objdel.java

```
class A
{
    public void sum(int x,int y)
    {
          System.out.print("\n sum="+ (x+y));
    }
}
class B
{
    A obja=new A();
    public void add(int x,int y)
    {
          obja.sum(x,y);
    }
}
class objdel
{
    public static void main(String argv[])
    {
          B objb=new B();
          objb.add(10,20);
    }
}
```



Bin>javac objdel.java

Bin>java objdel

sum=30

Explanation:

In the main(), the statement **objb.add(10,20);** calls **add(int x,int y)** of **B** class by passing 10, 20 to x, y and the statement **obja.sum(x,y)** calls **sum(int x,int y)** of **A** class which prints sum of two numbers i,e **sum=30**.

In the main(), **objb** object is assigned with the operation of addition of two numbers and **objb** delegates its operation or request to object of **A** i.e **obja** and **obja** is completed the operation.

This process of delegating the operation of one object to another object is known as object delegation which is an alternative of inheritance.

The difference between inheritance and object delegation is, inheritance forms **"IS-A relationship"** where as object delegation forms **"HAS-A relationship"** i.e inheritance is direct relationship among parent and child classes where as in delegation it is in-directed relationship.

Object Composition

We have understood the concepts of classes, objects and inheritance. The most common techniques for re-using functionality in object-oriented systems are class inheritance and object composition. As explained, inheritance is a mechanism of building new classes by deriving certain properties from existing classes. In inheritance, if the class **B** is sub class of **A**, it is said to be, the class **B** has all the properties of **A** in addition to the features of its own i.e **B**.

Another possible re-usability is through object composition. In some situations we want to create objects of classes as data members in another class. This process of creation of objects as data members in another class is the concept of **object composition**. Object composition is an alternative to multiple inheritance.

In the case of composition, new functionality is obtained by assembling of composing objects to support more powerful functionality. This new approach takes a view that an object and the relationship is called a "**has-a relationship**" or "**containership**".

In OOP, the has-a-relationship occurs when objects of classes is contained in another class as a data member(instance variables).

Example:

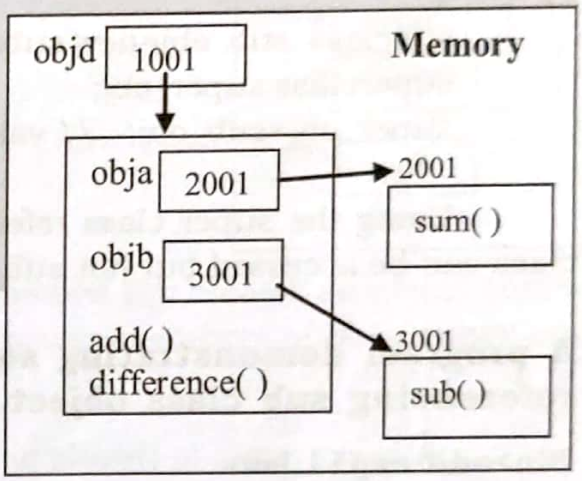
```
class A
{
        :
        :
}
class B
{
        :
        :
}
class C
{
        A obja=new A();
        B objb=new B();      } object composition
        :
}
```

A program demonstrating objects composition

Bin>edit objscomp.java

```
class A
{
    public void sum(int x,int y)
    {
        System.out.print("\n sum="+ (x+y) );
    }
}
class B
{
    public void sub(int x,int y)
    {
        System.out.print("\n sub="+ (x-y) );
    }
}
class D
{
    A obja=new A();
    B objb=new B();
    public void add(int x, int y)
    {
        obja.sum(x,y);
```

```
    }
    public void difference(int x,int y)
    {
        objb.sub(x,y);
    }
}
class objscomp
{
    public static void main(String argv[])
    {
        D objd=new D();
        objd.add(20,5);
        objd.difference(10,4);
    }
}
```



Bin>javac objscomp.java
Bin>java objscomp

```
sum=25
sub=6
```

Explanation:

In the above program, the **D** class is composed with objects of **A** and **B** classes i.e. class **D** can do all operations of **A** and **B** classes.

In the main(), the statement **objd.add(20,5);** calls the **add(int x,int y)** method of **class D** and in this method the statement **obja.sum(x,y);** calls the **sum(int x,int y)** method of **class A** which actually prints the addition of two numbers as shown in the output.

Similarly, the statement **objd.difference(10,4);** calls the **difference(int x,int y)** method of **class D** and in this method the statement **objb.sub(x,y);** calls the **sub(int x,int y)** method of **class B** which actually prints the substraction of two numbers as shown in the above output.

Super class reference variable referencing sub class object

In inheritance, a super class reference variable can store the address of sub class object where as a sub class reference variable can not store the address of super class.

Example:

```
class superclass
{
        :
        :
}
class subclass extends superclass
{
        :
        :
}
subclass sub_obj=new subclass();
superclass super_obj;
super_obj=sub_obj;  // valid statement
```

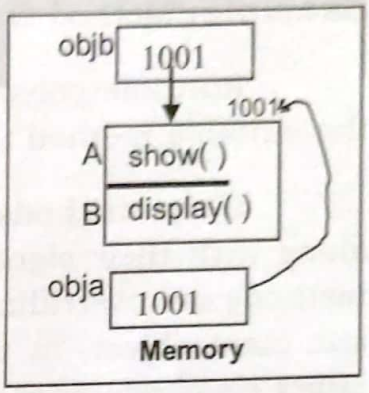
Using the super class reference variable only the members of super class can be accessed but not sub class.

A program demonstrating super class reference variable referencing sub class object.

Bin>edit exp11.java

```
class A
{
    public void show()
    {
        System.out.print("\n show() of super class…");
    }
}
class B extends A
{
    public void display()
    {
        System.out.print("\ndisplay() of sub class…");
    }
}
class exp11
{
    public static void main(String argv[])
    {
        B objb=new B();
        objb.show();            // A.show() is called
```

```
        objb.display(); // B.show() is called
        A obja;  // obja is reference variable of A
        obja=objb; //  valid assignment
        obja.show();
        //obja.display();
        //Error: display() does not exist in A
    }
}
```



Bin>javac exp11.java
Bin>java exp11

show() of super class...
display() of sub class...
show() of super class...

Explanation:

In the main(), the statement **B objb=new B();** creates an object objb to sub class **B** and using this **objb** we can access members of both **super** and **sub** classes. Therefore

The statement **objb.show()** calls the **show()** of super class **A**.
The statement **objb.display()** calls the **display()** of sub class **B**.

The statement **A obja;** creates only the reference variable of **A** class and the statement **obja=objb;** assigns the address of **objb** of subclass **B** to **obja** of super class **A.** This assignment is a valid assignment because a super class reference variable can store the address of sub class.

The statement **obja.show();** calls the **show()** of super class **A**, because **obja** is a super class reference variable therefore it can access the members of super class where as the statement **obja.display();** is error because a super class reference variable can't access the members of sub class because super class does not know about sub class.

Dynamic Method Dispatch or Dynamic Binding

In Java, a method call can be bound to the actual function either at compile time or at runtime.

Binding of method call to the called method if takes place at compile time then it is known as **compile time binding** or **early binding** or **static binding**.

Compile time binding is an example of compile time polymorphism.

Example: Method Overloading

Binding of method call with the called method if takes place at runtime then it is known as **runtime binding** or **late binding** or **dynamic binding**.

Runtime binding is an example of runtime polymorphism.

Example: Method Overriding

Runtime polymorphism allows to postpone the decision of selecting the suitable method until runtime.

If the methods of super class and sub class are having same name along with their signature, it is said to be method overriding. When the methods are overriding and the super class reference variable points to the sub class object, in this case if an overridden method is called using the super class reference variable then the method of sub class is called but not the super class. This happens so because the jvm checks the reference variable to which class object it is pointing, based on that it selects the method for execution. As the reference variable is having the reference (address) of sub class object therefore it calls the method of sub class. This is how **runtime binding** is achieved.

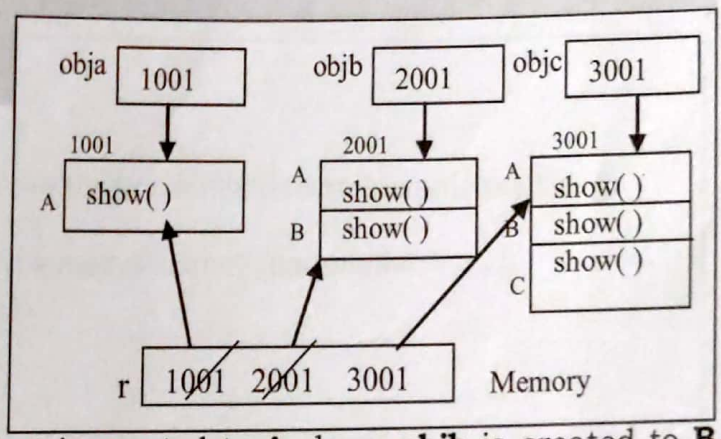
A program on dynamic method dispatch.

Bin>edit dynmethod.java

```
class A
{
      public void show()
      {
            System.out.print("\n show of A...");
      }
}
class B extends A
{
      public void show()
      {
        System.out.print("\n show of B...");
      }
}
class C extends B
{
```

```
        public void show()
        {
              System.out.print("\n show of C...");
        }
}
class dynmethod
{
      public static void main(String argv[])
      {
            A obja=new A();
            B objb=new B();
            C objc=new C();
            A r;    // r is a reference variable
                    // of supermost class A
            r=obja;
            r.show(); //  calls show() of A
            r=objb;
            r.show(); //  calls show() of B

            r=objc;
            r.show();  //  calls show() of C
      }
}
```



Bin>javac dynmethod.java
Bin>java dynmethod

show of A...
show of B...
show of C...

Explanation:

In the main(), **obja** object is created to **A** class, **objb** is created to **B** class and objc is created to **C** class as shown in above memory.

In the statement **r=obja;** assigns the **A** class object **obja** address(**1001**) to **r** which is also belongs to **A** class. Now **r** is a reference variable pointing to **A** class object **obja** therefore the statement **r.show();** calls the **show()** method of **A** class.

In the statement **r=objb;** assigns the **B** class object **objb** address(**2001**) to **r** which is also belongs to **A** super class and this is valid assignment because super class reference variable can store the address or reference of sub class.

Now as **r** is pointing or referencing to **B** class object **objb** therefore the statement **r.show();** calls the **show()** method of **B** class instead of **A** class.

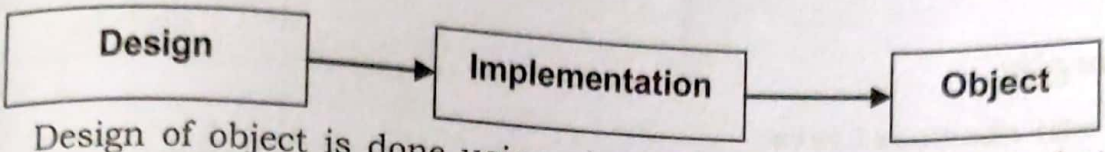
In the statement **r=objc;** assigns the **C** class object **objc** address(**3001**) to **r** which is also belongs to **A** super class and this is valid assignment because super class reference variable can store the address or reference of sub class. Now as **r** is pointing or referencing to **C** class object **objb** therefore the statement **r.show();** calls the **show()** method of **C** class instead of **A** class.

In the way java calls the methods of classes at runtime based on the address of object it is holding and it leads to dynamic method dispatch.

Abstract Classes

Generally, to construct an object it goes in two phases i.e. first is designing and second is implementation.

In the designing phase only the structure of object is declared and in the implementation phase the actual implementation of structure designed is defined.



Design of object is done using abstract classes and these abstract classes are implemented by other classes which are known as concrete classes.

An Abstract class is a class which contains the frame work or design of an object using which new classes can be constructed to provide new functionality.

An abstract class is a class with minimum one abstract method. An abstract method is one which have only declaration but no definition. The abstract method and abstract class should be declared with the keyword **abstract**.

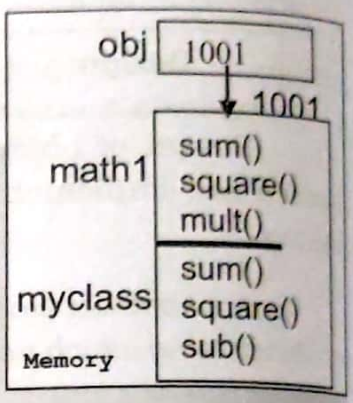
Syntax:

```
abstract class classname
{
        :
        abstract access_specifier returntype methodname([parameters list..]);
        :
        access_specifier returntype methodname2([parameters list...])
        {
                :
                :
        }
}
```

Example:

```
abstract class math1
{
        :
        abstract public void sum(int x, int y);
        abstract public int square(int x);
        :
}
```

The above class **math1** is an abstract class because it contains abstract methods. To the abstract classes objects can not be created because abstract classes are half developed classes. Abstract classes are used for extending into sub classes for implementation of abstract methods. It is the responsibility of sub classes that they should implement the abstract classes and such implemented classes are known as **concrete classes**. To the concrete classes objects can be created and make use of them.



Example:

Bin>edit absclass1.java

```
abstract class math1
{
    abstract public void sum(int x,int y);
    abstract public int square(int x);
    public void mult(int x, int y)
    {
          System.out.print("\n mult="+(x+y));
    }
}
class absclass1
{
    public static void main(String argv[])
    {
          math1 obj=new math1();  // error
          obj.sum(10,20);  // error
          System.out.print("\n square ="+obj.square(5));//error
          obj.mult(3,5);  //error
    }
}
```

Bin>javac absclass1.java

Error: math1 is abstract; can not be instantiated
math1 obj=mew math1();

The above program generates errors because, to the abstract class object is created which is not allowed and methods can not be called.

A program extending the abstract class.

Bin>edit absclass2.java

```
abstract class math1
{
    abstract public void sum(int x,int y);
    abstract public int square(int x);
    public void mult(int x, int y)
    {
          System.out.print("\n mult="+(x+y));
    }
}
class myclass extends math1
{
    public void sum(int x,int y)
    {
          System.out.print("\n sum="+ (x+y) );
    }
    public int square(int x)
    {
          return(x*x);
    }
    public void sub(int x,int y)
    {
          System.out.print("\n sub="+ (x-y) );
    }
}
class absclass2
{
    public static void main(String argv[])
    {
          myclass obj=new myclass();
          obj.sum(10,20);
          System.out.print("\n square ="+obj.square(5));
          obj.mult(2,2);
          obj.sub(9,4);
    }
}
```

Bin>javac absclass2.java
Bin>java absclass2
sum=30
square=25
mult=4
sub=5

Explanation:

In the main(), **obj** is created to myclass and this is allowed because myclass is a concrete class. Using **obj**, methods of sub class(**myclass**) and superclass(**math1**) can be accessed, but first preference is given to subclass incase overriding methods are present. The subclass can also contain its own methods apart from superclass methods.

Note: An abstract class may not contain abstract methods. These classes are used only for inheriting into subclasses.

The *final* keyword

The final keyword in java can be used for three different purposes.

1. The final variables.
2. The final methods.
3. The final classes.

1. The final variables

The final keyword can be used to create final or constant variables. The final variables once initialized whose contents cannot be modified throughout the program.

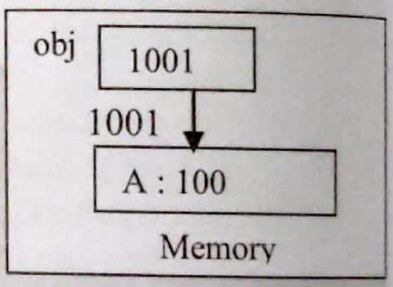
Syntax:

```
[access] final type variable1=value1 [, variable2=value2,..];
```

Example:

Bin> edit final1.java

```
class sample
{
    public final int A=100;
}
class final1
{
    public static void main(String argv[])
    {
          sample obj=new sample();
          System.out.print("\n obj.A="+obj.A);
          // obj.A=200; // Error: cannot assign a value to
                        // final variable.
    }
}
```



```
Bin>javac final1.java
Bin>java final1
obj.A=100
```

Explanation:

In the above program, **A** is defined as final variable and is initialized with 100, therefore **objA.A** prints 100.

The statement **obj.A=200;** is error because final variable value cannot be modified.

Note:

1. Execute the above program without comments and understand the errors.

2. The final methods

The final keyword can be used to create final methods. In java, methods can also be declared with the final keyword and such methods are called as final methods. Such final methods cannot be overridden in the sub class i.e. final method prevents overriding.

Syntax:

```
class classname
{
        :
        access final returntype methodname([type para1, ...])
        {
                //code of final method here
        }
}
```

Example:

Bin>edit final2.java

```
class A
{
        public final void show()
        {
                System.out.print("\n show of A...");
        }
}
```

```
class B extends A
{

      public void show()  // Error:cannot override final
      {                   // method
            System.out.print("\n show of B...");
      }
      public void display()
      {
            System.out.print("\n display of B...");
      }
}
class final2
{
      public static void main(String argv[])
      {
            B obj=new B();
            obj.show();
            obj.display();
      }
}
```

Bin>javac final2.java
Bin> java final2

Error: final method show() of A cannot be overridden in subclass B.

Explanation:

The above program generates error because final method show() of A is overridden in subclass B.

Note:

1. Execute the above program without **final** keyword and the program executes successfully.

The final classes

The final keyword can be used to create final classes. In java, classes can be declared using final keyword and such classes are called as final classes. The final classes cannot be used for inheriting into subclasses i.e. final classes prevents inheritance.

Syntax:

```
final class classname
{
        :
}
```

Example:

Bin>edit final3.java

```
final class A
{
        public void show()
        {
                System.out.print("\n show of A...");
        }
}
class B extends A  // Error: final class cannot inherit
{
        public void display()
        {
                System.out.print("\n display of B...");
        }
}
class final2
{
        public static void main(String argv[])
        {
                B obj=new B();
                obj.show();
                obj.display();
        }
}
```

Bin>javac final3.java

Bin> java final3

Error: final class A cannot be extended into subclass B.

Explanation:

The above program generates error because **final class A** cannot be inherited into subclass **B**

Note:

1. Execute the above program without **final** keyword and program executes successfully.

Benefits of Inheritance

There are many important benefits of inheritance. They are code reuse, ease of code maintenance and extension, reduction in the time to market. The following situations explain benefits of inheritance:

When inherited from another class, the code that provides a behavior required in the sub class need not have to be rewritten. Benefits of reusable code include increased reliability and a decreased maintenance cost because of sharing of the code by all its users.

Code sharing can occur at several levels. For example, at a higher level, many users or projects can use the same class. These are referred to as software components. At the lower level, code can be shared by two or more classes within a project.

When multiple classes inherit from the same super class, it guarantees that the behavior they inherit will be the same in all cases.

Inheritance permits the construction of reusable software components. Already, several such libraries are commercially available and many more are expected to be available in the near future.

When a software system can be constructed largely out of reusable components, development time can be concentrated on understanding the portion of a new system. Thus, software systems can be generated more quickly and easily by rapid prototyping.

Cost of Inheritance

Apart from many benefits inheritance also have some drawbacks, it incurs compiler overhead. In Inheritance Relationship, there are certain members in the super class that are not at all used, however data space is allocated to them. This necessitates the need for specialized inheritance, which is complex to develop. The following are some of the perceived costs of inheritance:

Inherited methods, which must be prepared to deal with arbitrary subclasses, are often slower than specialized codes.

The use of any software library frequently imposes a size penalty over the use of systems specially constructed for a specific project. Although

this expense may in some cases be substantial, it is also true that as memory cost decreases, the size of programs is becoming less important.

Message passing by its very nature is a more costly operation than the invocation of simple procedures. The increased cost is however marginal and is often much lower in statically bound languages like c++. Therefore, the increased cost must be weighed against the benefits of the object oriented techniques.

Although object oriented programming is often thought as a solution to the problem of software complexity, overuse or improper use of inheritance can simply replace one form of complexity with another.