Unit-2

❖ Class Fundamentals :

- ✓ Java is a true object-oriented language and therefore the all java programs develop using the classes.
- ✓ Classes create objects and objects use methods to communicate between them.
- ✓ In java, data items are called fields and the functions are called methods.

➢ Defining a Class :

- ✓ The basic form of a class definition is:

```
class classname [extends superclassname]
{
        [fields declaration;]
        [methods declaration;]

}
```

- ✓ Everything inside the square brackets is optional.
- ✓ **Example :**

```
class Empty
{
}
```

- ✓ In above example the body is empty, this class does not contain any properties and therefore cannot do anything.
- ✓ The keyword **extends** indicates that the properties of the superclassname class are extended to the classname class.
- ✓ This concept is known as inheritance.
- ✓ Fields and methods are declared inside the body.

➢ Fields Declaration :

- ✓ We can declare the instance variables exactly the same ways as we declare local variable.
- ✓ **Example :**

```
class Rectangle
{
        int length;
        int width;

}
```

- ✓ The class Rectangle contains two integer type instance variables.

- ✓ We can declare them in one line as
  int length, width;
- ✓ Remember these variables are only declared and therefore no storage space has been created in the memory.
- ✓ Instance variables are also known as member variables.

➢ Methods Declaration :

- ✓ Methods are declared inside the body of the class but immediately after the declaration of instance variables.
- ✓ The general form of a method declaration is:

```
type methodname (parameter list)
{
        method-body
}
```

- ✓ Method declaration have four basic parts:
  1. The name of the method(methodname)
  2. The type of the value that method returns(type)
  3. A list of parameters(parameter list)
  4. The body of the method
- ✓ The type specifies the type of value that method would return.
- ✓ This could be a simple data type such as int as well as any class type.
- ✓ It could even be void type, if the method does not return any value.
- ✓ The methodname is valid identifier.
- ✓ The parameter list is always enclosed in parentheses.
- ✓ This list contains variable names and types of all the vales we want to give to the method as input.
- ✓ The variable in the list are separated by commas.
- ✓ **Example :**

```
class Rectangle
{
        int length, width;
        void getdata(int x, int y)
        {
                length=x;
                width=y;
        }
}
```

➢ Creating Objects:

- ✓ An object in java is essentially a block of memory that contains space to store all the instance variables.
- ✓ Objects in java are created using the **new** operator.
- ✓ The new operator creates an object of the specified class.
- ✓ **Example :**

```
Rectangle r1;
r1=new Rectangle();
```

- ✓ The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable.
- ✓ Both statements can be combined into one as shown below:

```
Rectangle r1=new Rectangle();
```

- ✓ The method **Rectangle()** is the default constructor of the class.
- ✓ We can create any number of objects of any class.

```
Rectangle r1=new Rectangle();
Rectangle r2=new Rectangle();
```

- ✓ Note that each object has its own copy of the instance variables of its class.
- ✓ This means that any changes to the variables of one object have no effect on the variables of another.

➢ Accessing Class Members:

- ✓ We can not access the instance variables and the methods directly.
- ✓ To do this, we must use the object and the dot operator as shown below:

```
objectname.variablename=value;
objectname.methodname(parameter list);
```

- ✓ Here, objectname is the name of the object.
- ✓ variablename is the name of the instance variable.
- ✓ methodname is the method that we wish to call and parameter list is a comma separated list of actual values.
- ✓ **Example :**

```
r1.length=15;
r1.width=10;
r2.length=20;
r2.width=12;


r1.getdata(15, 10);
```

➢ Method Overloading:

- ✓ In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading.
- ✓ When we call a method, Java matches the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism.
- ✓ **Example :**

```
Class Room
{
        float length, breath;
        void getdata(float x, float y)
        {
                length=x;
                breadth=y;
        }
        void getdata(float x)
        {
                length=breadth=x;
        }

}




Room r1=new Room();

r1.getdata(25.0, 15.0);          //using method1

Room r2=new Room();

r2.getdata(20.0);        //using method2
```

➢ Static Members:

- ✓ A class is basically contains two sections.
- ✓ One declares variables and the other declares methods.
- ✓ These variables and methods are called instance variables and instance methods.
- ✓ They are accessed using the objects with dot operator.
- ✓ Now we want to define a member that is common to all the objects and accessed without using a particular object. Such members can be defined as follows:

```
static int count;
static int max(int x, int y);
```

- ✓ The members that are declared static as shown above are called static members.
- ✓ Static variables are used when we want to have a variable common to all instances of a class.
- ✓ Java creates only one copy for a static variable.
- ✓ Like static variables, static methods can be called without using the objects.
- ✓ Note that the static methods are called using class names. No objects have been created for use.
- ✓ Static methods have several restrictions:
    1. They can only call other static methods.
    2. They can only access static data.
    3. They cannot refer to **this** or **super** in any way.

- ✓ **Example :**

```
class temp
{
        static float mul(float x, float y)
        {
                return x*y;
        }
        static float divide(float x, float y)
        {
                return x/y;
        }

}

class test

{

        public void static main(string args[ ])

        {

                float a=temp.mul(4.0,5.0);

                float b=temp.divide(4.0,5.0);

                system.out.println("A :"+a);

                system.out.println("B :"+b);

        }

}
```

➢ Nesting of Methods:

- ✓ We discussed earlier that a method of a class can be called only by an object if that class using the dot operator.
- ✓ There is an exception to this.
- ✓ A method can be called by using only its name by another method of the same class. This is known as nesting of methods.
- ✓ **Example :**

```
class Nesting
{
        int m,n;
        Nesting(int x, int y)
        {
                m=x;
                n=y;

        }

        int largest()

        {

                if(m>=n)

                        return m;

                else

                        return n;

        }

        void display()

        {

                int large=largest();

                system.out.println("Largest Value :"+large);

        }

}

class NestingTest
```

```
{
        public static void main(string args[ ])
        {
                Nesting nest=new Nesting(50, 40);
                nest.display();
        }
}
```

❖ Constructors :

- ✓ A constructor is a 'special' method whose task is to initialize the objects of its class. It is special because its name is the same as the class name.
- ✓ The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
- ✓ **Example :**

```
class person
{
        int m, n;
        person()        //constructor defined
        {
                m=10;
                n=20;
        }
}
```

- ✓ A constructor that accepts no parameters is called the default constructor.
- ✓ The default constructor for class A is A(). If no such constructor is defined, then the compiler supplies a default constructor.

➢ Some special characteristics of Constructor :

- ✓ They are invoked automatically when the objects are created.
- ✓ They do not have return types, not even void and therefore they cannot return values.
- ✓ Like other Java methods, they can have default arguments.
- ✓ Constructors can not be virtual.

❖ Parameterized Constructor :-

- ✓ The constructors that can have arguments are called parameterized constructors.
- ✓ **Example :**

```
class person
{
        int m, n;

        person(int x, int y)
        {
                m=x;
                n=y;
        }

}
```

- ✓ We must pass the values as arguments to the constructor when an object is declared.

❖ Constructor Overloading :-

- ✓ We have used two kinds of constructors. They are:

```
person();               //no argument

person(int, int);       //two arguments
```

- ✓ In the first case, the constructor itself supplies the data values and no values are passed by calling program.
- ✓ In the second case, the function call passes the appropriate values from main().
- ✓ Java permits us to use both these constructors in the same class.
- ✓ **Example :**

```
class person
{
        int m, n;

        person()
        {
                m=0;
                n=0;
        }
        person(int a, int b)
        {
                m=a;
                n=b;
```

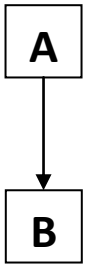
```
        }
};
```

- ✓ This declares two constructors for a person object.
- ✓ The first constructor receives no arguments, the second receives two integer arguments.
- ✓ person P1=new person();
  This statement automatically invoke the first constructor and set both m and n of P1 to 0.
- ✓ person P2=new person(20, 40);
  This statement will initialize the data members m and n of P2 to 20 and 40 respectively.
- ✓ When more than one constructors is defined in a class, than we say that the constructor is overloaded.

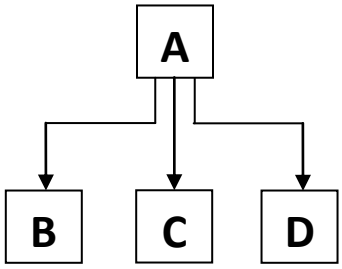
❖ Inheritance :

➢ Introduction:

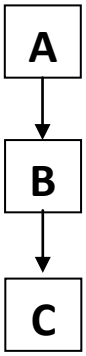
- ✓ Reusability is yet another feature of OOP.
- ✓ It is always nice if we could reuse something that already exists rather than creating the same all over again.
- ✓ Java supports this concept.
- ✓ Java classes can be reused in several ways.
- ✓ This is basically done by creating new classes, reusing the properties of existing ones.
- ✓ The mechanism of deriving a new class from an old one is called **inheritance**.
- ✓ The old class is known as the base class or super class or parent class and the new one is called the subclass or derived class or child class.
- ✓ The inheritance allows subclasses to inherit all the variables and methods of their parent classes.
- ✓ There are several types of inheritance :
  1. Single Inheritance(Only one super class)
  2. Multiple Inheritance(Several super classes)
  3. Hierarchical Inheritance(One super class, many subclasses)
  4. Multilevel Inheritance(Derived from a derived class)



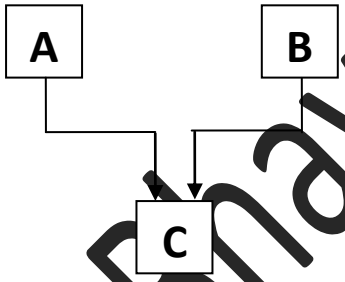
(a) Single Inheritance



(b) Hierarchical Inheritance



(c) Multilevel Inheritance



(d) Multiple Inheritance

- ✓ Java does not directly implement multiple inheritance.
- ✓ This concept is implemented using a secondary inheritance path in the form of interfaces.

➢ Defining a Subclass:

- ✓ A subclass is defined as follows:

```
class subclassname extends superclassname
{
        variable declaration;

        methods declaration;

}
```

- ✓ The keyword **extends** shows that the properties of the superclassname are extended to the subclassname.

- ✓ The subclass will now contain its own variables and methods as well those of the superclass.
- ✓ **Program :** Concept of Single Inheritance

```
class Room
{
    int length, breadth;
    Room(int x, int y)
    {
            length=x;
            breadth=y;
    }
    int area()
    {
            return(length * breadth);
    }
}
class BedRoom extends Room
{
    int height;
    BedRoom(int x, int y, int z)
    {
            super(x, y);
            height=z;
    }
    int volume()
```

```
        {
                return(length * breadth * height);
        }
    }
    class InherTest
    {
        public static void main(String args[])
        {
                BedRoom room=new BedRoom(14, 12, 10);
                int area1=room1.area();
                int volume1=room1.volume();
                System.out.println("Area1 = " + area);
                System.out.println("Volume1 = " + volume);
        }
}
```

Output :

```
Area1 = 168
Volume1 = 1680
```

- ✓ The statement

```
BedRoom room1=new BedRoom(14, 12, 10);
```

Calls first the **BedRoom** constructor method, which in turn calls the **Room** constructor method by using the **super** keyword.

Finally, the object **room1** of the subclass **BedRoom** calls the method **area** defined in the super class as well as the method volume defined in the subclass itself.

➢ Multilevel Inheritance:

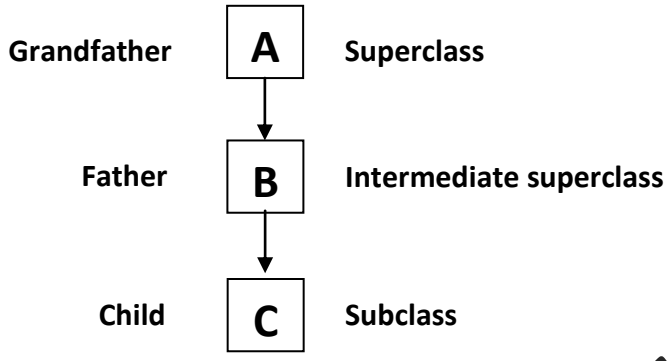


Fig : Multilevel Inheritance

- ✓ This concept allows us to build a chain of classes as shown in above figure.
- ✓ The class **A** serves as a base class for the derived class **B** which in turn serves as a base class for the derived class **C**.
- ✓ The chain ABC is known as inheritance path.
- ✓ A derived class with multilevel base classes is declared as follows:

```
class A
{
        .....
}
class B extends A       //First level
{
        .....
}
class C extends B       //Second level
{
        .....
}
```

- ✓ This process may be extended to any number of levels.
- ✓ The class C can inherit the members of both **A** and **B** as shown below figure:

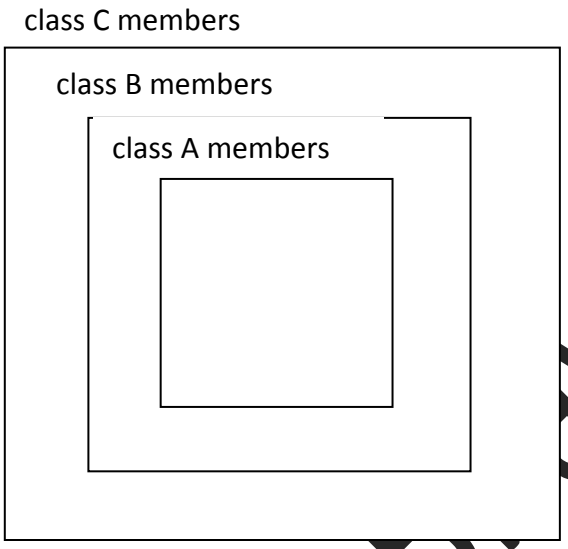


Fig : **C** contains **B** which contains **A**

➢ Hierarchical Inheritance:

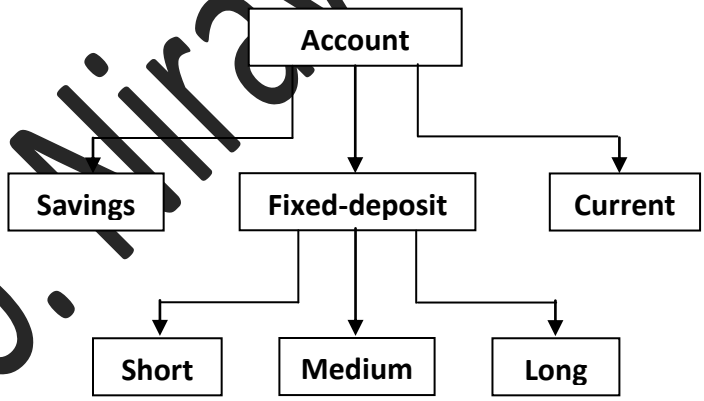


Fig : Hierarchical classification of bank accounts

- ✓ Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level.
- ✓ Above figure shows a hierarchical classification of accounts in a commercial bank.
- ✓ This is possible because all the accounts contains certain common features.

❖ Method Overriding :

- ✓ We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass.
- ✓ However, there may be occasions we should override the method defined in the superclass.
- ✓ This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass.
- ✓ Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding.

- ✓ **Program : Method Overriding**

```
class super
{
                    int x;
                    super(int x)
                    {
                            this.x=x;
                    }
                    void display()          //method defined
                    {
                            System.out.println("Super x="+x);
                    }
}
class sub extends super
{
                    int y;
                    sub(int x,int y)
                    {
                            super(x);
                            this.y=y;
                    }
                    void display()          //method defined again
                    {
                            System.out.println("Super x= "+x);
                            System.out.println("Sub y= "+y);
                    }
}
class OverrideTest
{
                    public static main(String args[ ])
```

```
{
        sub s1=new sub(100,200);
        s1.display();
}
}
```

Output :

```
Super x=100
Sub y=200
```

Note : The method display() defined in the subclass is invoked.

❖ Final Variables and Methods :

- ✓ All methods and variables can be overridden by default in subclass.
- ✓ If we wish to prevent the subclass from overriding the members of the superclass, we can declare them as final using the keyword **final** as a modifier.
- ✓ **Example :**

```
final int size=100;
final void showstatus()
{
        ……
}
```

- ✓ Making a method final ensures that the functionality defined in this method will never be altered in any way.
- ✓ Similarly, the value of a final variable can never be changed.

❖ Final Classes :

- ✓ Sometimes we may like to prevent a class being further subclasses for security reasons.
- ✓ A class that cannot be subclassed is called a final class.
- ✓ This is achieved in java using the keyword **final**.

```
final class classA
{
        …..
}
final class classB extends Someclass
{
        …..
```

```
}
```

- ✓ Above lines generate an error and the compiler will not allow it.
- ✓ Declaring a class final prevents any unwanted extensions to the class.

❖ Finalizer Methods :

- ✓ We have seen that a constructor is used to initialize an object when it is declared. This process is known as initialization.
- ✓ Similarly, Java supports a concept called finalization, which is just opposite to initialization.
- ✓ We know that Java runtime is an automatic garbage collecting system.
- ✓ It automatically frees up the memory resources used by the objects.
- ✓ But objects may hold other non-object resources such as file descriptors or window system fonts.
- ✓ The garbage collector cannot free these resources.
- ✓ In order to free these resources we must use a finalizer method.
- ✓ This is similar to destructors in C++.
- ✓ The finalizer method is simply **finalize()** and can be added to any class.
- ✓ Java calls that method whenever it is about to reclaim the space for that objects.

❖ Abstract Methods and Classes:

- ✓ We have seen that by making a method **final** we ensure that the method is not redefined in a subclass.
- ✓ Java allows us to do something that is exactly opposite to this.
- ✓ We can indicate that a method must always be redefined in a subclass, thus making overriding compulsory.
- ✓ This is done using the modifier keyword **abstract** in the method definition.

Example :

```
abstract class Shape
{
        .....
        .....
        abstract void draw();
        .....
        .....
}
```

- ✓ When a class contains one or more abstract methods, it should also be declared **abstract** as shown in above example.

- ✓ While using abstract classes, we must satisfy the following conditions :
  - We cannot use abstract classes to instantiate objects directly.
    Example: Shape s=new Shape();
    is illegal because **Shape** is an abstract class.
  - The abstract methods of an abstract class must be defined in its subclass.
  - We cannot declare abstract constructor or abstract static methods.

❖ Access Modifiers (Visibility Controls):

- ✓ We have seen that the variables and methods of a class are visible everywhere in the program.
- ✓ It may be necessary in some situations to restrict the access to certain variables and methods from outside the class.
- ✓ We can achieve this in Java by applying visibility modifiers to the instance variables and methods.
- ✓ The visibility modifiers are also known as access modifiers.
- ✓ Java provides three types of visibility modifiers :
  - public
  - private
  - protected

➢ public access

- ✓ Any variable or method is visible to the entire class in which it is defined.
- ✓ But if we want to make it visible to all the classes outside this class than this is possible by simply declaring the variable or method as **public**.
- ✓ **Example :**

```
public int number;
public void sum()
{
        .....
}
```

- ✓ A variable or method declared as **public** has the widest visibility and accessible everywhere.

➢ friendly access

- ✓ When no access modifier is specified, the member defaults to a limited versions of public accessibility known as "friendly" level of access.
- ✓ The difference between the "public" access and the "friendly" access is that the **public** modifier makes field visible in all classes, regardless of their packages while the friendly

access makes fields visible only in the same package, but not in other packages (A package is a group of related classes stored separately).

➢ protected access

- ✓ **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages.
- ✓ Note that non-subclasses in other packages cannot access the "protected" members.

➢ private access

- ✓ **private** fields enjoy the highest degree of protection.
- ✓ They are accessible only with their own class.
- ✓ They cannot be inherited by subclasses and therefore not accessible in subclasses.
- ✓ A method declared as **private** behaves like a method declared as **final**.

➢ private protected access

- ✓ A field can be declared with two keywords **private** and **protected** together like:

```
private protected int codeNumber;
```

- ✓ This gives a visibility level in between the "protected" access and "private" access.
- ✓ This modifier makes the fields visible in all subclasses regardless of what package they are in.
- ✓ Note that these fields are not accessible by other classes in the same package.
- ✓ Below table shown the visibility provided by various access modifiers.

Access Location \ Access Modifier	public	protected	Friendly(default)	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No