Inheritance

- 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.
- The new class is called the subclass or derived class or child class.
- Inheritance helps to reuse the properties of existing classes.
- The inheritance allows a subclass to inherit all the variables and methods of their parent classes.
- Inheritance may take different forms. These are-
  - Single inheritance(only one super class)
  - Multiple inheritance(several super classes)
  - Hierarchical inheritance(one super class many sub classes)
  - Multilevel inheritance(derived from a derived class)

- Java does not directly implement multiple inheritance (it uses interface which will be discussed later).
- A subclass is defined as follows-

  class *subclassname* **extends** *superclassname*
  {
  variable declaration;
  method declaration;
  }

- The keyword **extends** signifies that the properties of the *superclassname* are extended to the *subclassname*.
- The subclass will now contain its own variable and methods as well those of the superclass.
- Consider the following example-

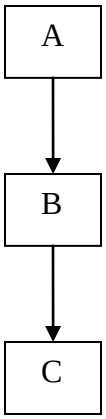
```
class Room
{
    int length;
    int breadth;
    Room(int x, int y)
    {
       length=x;
       breadth=y;
     }
     int area()
     {
       return(length*breadth);
     }
  }
```

```
class BedRoom extends Room
 {
     int height;
     BedRoom(int x, int y, int z)
      {
         super(x, y);       //pass values to superclass
         height=z;
      }
    int volume()
    {
      return (length*breadth*height) ;
    }
 }

class Inherexample
{
   public static void main(String args[])
    {
      BedRoom ob1=new BedRoom(10,20,30);
      int are1=ob1.area();                  //superclass method
      int volume1=ob1.volume();      //baseclass method
      System.out.println("Area1= "+area1);
      System.out.println("Volume1= "+volume1);
    }
 }
```

- A subclass constructor is used to construct the instance variables of both the subclass and the superclass.
- The subclass constructor uses the keyword **super** to invoke the constructor method of the superclass.
- **super** may only be used within a subclass constructor method.
- The call to superclass constructor must appear as the first statement within the subclass constructor.
- The parameter in the **super** call must match the order and type of the instance variable declared in the superclass.

- **Multilevel inheritance-**
- Here a derived class is used a super class.
- It allows to build a chain of classes as follows-



- The class A serves as a base class for the derived class B.
- B in turn serves as a base class for the derived class C.
- The chain ABC is known as inheritance path.

- **Overriding methods-**
- We can override a 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.

- **Final variable and methods-**
- Using the keyword **final** we can prevent the sub classes from overriding the members of the super classes.
- Example- final float pie=3.14;
  final void display() {…}
- Making a method **final** ensures that the functionality defined in this method will never be altered in anyway.
- Similarly the value of a **final** variable can never be changed and they behave like class variables (i.e.; they do not take any space on individual objects of the class).

- **Final classes-**
- A class that cannot be subclassed is called a final class.
- Example- **final** class Student {…}
- Any attempt to inherit these classes will cause an error and the compiler will not allow it.

- **Finalizer method-**
- Java supports a concept called finalization which is just opposite to initialization.
- We know that Java run-time 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 (file descriptors, windows system font, etc.). The garbage collector cannot free these resources.

- In order to free these resources we must use a finalizer method which 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 object.
- The **finalize()** method should explicitly define the task to be performed.

- **Abstract methods and classes-**
- Java allows us to write a method that must always be redefined is a subclass.
- 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 the example above.
- We cannot use abstract classes to instantiate objects directly. For 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 constructors or abstract static methods.

- **Visibility control-**
- Visibility modifiers prevent objects of a class to directly alter the value of a variable or access a method.
- Java provides three visibility modifiers- **public**, **private** and **protected**.
- A variable or method declared as **public** has the widest possible visibility and accessible everywhere.
- **private** fields are accessible only within their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. We cannot override a non-private method in a subclass and then make it private.
- The **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. However non-subclasses in other packages cannot access the **protected** members.
- A field can be declared with two keywords **private** and **protected** together like:

  private protected int accNumber;

  This gives a visibility level in between the **private** and **protected** access. This modifier makes the fields visible in all subclasses regardless of what package they are in. However these fields are not accessible by other classes in the same package.
- When no access modifier is specified, the member defaults to a limited version of public accessibility known as **friendly** level of access. **Friendly** access makes fields visible only in the same package, but not in other packages.