Chapter 4: Inheritance and Interface:

4.1 Introduction to Inheritance and its uses

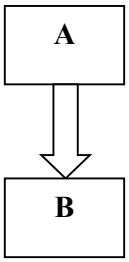
- If one class can access the properties of another class that is called inheritance.
- Inheritance is process by which new classes can be created using old class.
- It is concept by which sub classes are created using some properties of super class.
- The use of inheritance is that base class data we can reuse in sub class.
- Inheritance is referred as "**is-a**" relationship.

4.2 Types of inheritance

4.2.1 Single inheritance.
4.2.2 Multilevel inheritance
4.2.3 Hierarchical inheritance
4.2.4 Multiple inheritance
4.2.5 Hybrid inheritance

4.2.1 Single inheritance with using constructor.

This is a simple form of inheritance in which one super class per one sub class. In below structure A class is super class and B class is sub class.



Example

```
class Mclass
{
        int a;
        String s;
        Mclass()
        {
                a=5;
                s="single inheritance";
        }
        void show_mclass()
        {
                System.out.println("a= "+a);
                System.out.println("s= "+s);
        }
}
class Nclass extends Mclass
```

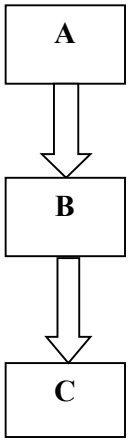
```
{
        double d;
        Nclass()
        {
                d=12.345;
        }
        void show_nclass()
        {
                System.out.println("d= "+d);
        }
}
class pb1
{
        public static void main(String args[])
        {
                Mclass n1=new Mclass();
                Nclass n2=new Nclass();
                n1.show_mclass();
                n2.show_nclass();
        }
}
```

Output:

```
a= 5
s= single inheritance
d= 12.345
```

4.2.2 Multilevel inheritance.

If sub class is derived from super class which itself is derived from some other class this type of structure is called multilevel inheritance.



Example:

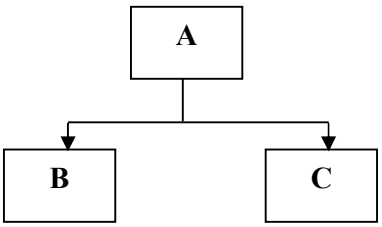
```
class Mclass
{
        int a;
}
class Nclass extends Mclass
{
        double b;
}
class Oclass extends Nclass
{
        Oclass()
        {
                a=10;
                b=45.678;
        }
        void show()
        {
                System.out.println("a= "+a);
                System.out.println("b= "+b);
        }
}
class pb1
{
        public static void main(String args[])
        {
                Oclass n1=new Oclass();
                n1.show();
        }
}
```

Output:

```
a= 10
b= 45.678
```

4.2.3 Hierarchical inheritance.

This is form of inheritance where one or more classes are derived from common base class.



Example:

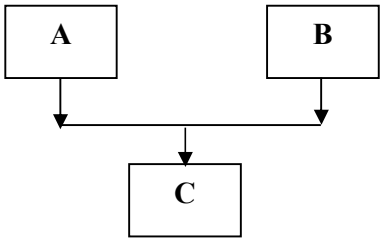
```
class Mclass
{
        int a;
}
class Nclass extends Mclass
{
        Nclass()
        {
                a=10;
        }
        void show_nclass()
        {
                System.out.println("a= "+a);
        }
}
class Oclass extends Mclass
{
        Oclass()
        {
                a=20;
        }
        void show_oclass()
        {
                System.out.println("a= "+a);
        }
}
class pb1
{
        public static void main(String args[])
        {
                Nclass n1=new Nclass();
                Oclass n2=new Oclass();
                n1.show_nclass();
                n2.show_oclass();
        }
}
```

Output:

```
a= 10
a=20
```

4.2.4 Multiple inheritances.

This type of inheritance is not supported in JAVA for that we use the concept of interface and that we will discuss later in this chapter.



4.2.5 Hybrid inheritance.

When two or more types of inheritance are combined together then it forms the hybrid inheritance.

4.3 Method overriding. (June'11)

Definition: "In class hierarchy, when method in sub class has the same name and signature as a method in its super class, then the method in sub class is said to override the method in super class"

Method overriding involves inheritance. Method overriding occurs when a class declares a method that has the same type of signature as a method declares by one of its super classes.

Method overriding is a very important capability because it forms the basis for **run-time polymorphism**.

An overriding method replaces the method it overrides.

Example:

```
class Aclass
{
        void show()
        {
                System.out.println("in A class");
        }
}

class Bclass extends Aclass
{
        void show()
        {
                System.out.println("in B class");
        }
}
class pb1
{
        public static void main(String args[])
```

```
        {
                Aclass a1=new Aclass();
                a1.show();
                Bclass b1=new Bclass();
                b1.show();
        }
}
```

Output:

```
in A class
in B class
```

4.3.1. Difference between method overloading and overriding. (Winter'14, June'11)

Method overloading	Method overriding
Method overloading occurs when method name is same but different number and different type of arguments within same class.	Method overriding occurs when a class declares a method that has the same type of signature as a method declares by one of its super classes
It does not use the concept of inheritance.	It uses the concept of inheritance.
In case of method overloading, parameters must be different.	In case of method overriding, parameters must be same.
It achieves compile time polymorphism.	It achieves run time polymorphism.
Private and final method can be overloaded.	Private and final method can not be override.
Static method can be overloaded.	Static method can not be overridden.
Static binding is being use for overloaded method.	Dynamic binding is being used for overriding method.

4.4 Super keyword (Winter'14,Summer'14,Winter'13, Summer'13, June'11)

The **super** is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of super Keyword

1. Super is used to refer immediate parent class instance variable.
2. Super () is used to invoke immediate parent class constructor.
3. Super is used to invoke immediate parent class method.

1) Super is used to refer immediate parent class instance variable.

Example:

```
class abc
{
```

```
        int a=10;
}
class demo extends abc
{
        int a=20;
        void show()
        {
                System.out.println(a);
                System.out.println(super.a);
        }
}
class test
{
        public static void main(String s[])
        {
                demo d1=new demo();
                d1.show();
        }
}
```

Output:

```
20
10
```

2) super is used to invoke parent class constructor.

Example:

```
class abc
{
        abc(int x)
        {
                System.out.println("x="+x);
        }
}
class demo extends abc
{
        demo()
        {
                super(100);
                System.out.println("in demo class");
        }
}
class test
{
```

```
        public static void main(String s[])
        {
                demo d1=new demo();
        }
}
```

Output:

```
x=100
in demo class
```

3) super is used to invoke parent class method.

Example:

```
class abc
{
        void show()
        {
                System.out.println("in abc class ");
        }
}
class demo extends abc
{
        void show()
        {
                super.show();
                System.out.println("in demo class");
        }
}

class test
{
        public static void main(String s[])
        {
                demo d1=new demo();
                d1.show();
        }
}
```

Output:

```
in abc class
in demo class
```

4.5 Final keyword (Winter'14,Summer'14,Winter'13, Summer'13,June'11)

The **final keyword** in java is used to restrict the user. The final keyword can be used in many contexts. Final can be:

1. variable
2. method
3. class (Stop inheritance)

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Final variable.

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Example:

```
class test
{
        public static void main(String s[])
        {
                final int a=10;
                a=20;   //it will generate error
        }
}
```

Output:

This code will generate **error** because **a** is declare as final and final variable value can not be change.

2) Final method

If you make any method as final, you cannot override it.

Example

```
class abc
{
        final void show()
        {
                System.out.println("hello");
        }
}
class demo extends abc
{
        void show()
        {
                System.out.println("in demo class");
        }
}
class test
{
        public static void main(String s[])
```

```
        {
                demo d1=new demo();
                d1.show();
        }
}
```

Output:
This code will generate **error** because **show** method in abc class is declared as final method so that you cannot override it.

3) Final class (Stop inheritance)

If you make any class as final, you cannot extend it.

```
final class abc
{
}
class demo extends abc
{
}
class test
{
        public static void main(String s[])
        {
                demo d1=new demo();
        }
}
```

Output:
This code will generate **error** because abc class is final, and final class can not be inherited

4.6 Interface (May-June'12)

Interface is collection of abstract methods and final data members. Interface is similar as classes but their methods do not have any definition body and data members contain only constants. Thus interface is used to hide the implementation details. **Interfaces** are used to implement multiple.

When we create an interface, we are basically creating a set of methods without any implementation that must be overridden by the implemented classes. The advantage is that it provides a way for a class to be a part of two classes: one from inheritance hierarchy and one from the interface.

Creation and implementation of an interface

An interface is defined much like a class. The difference between class and interface is that none of the methods in your interface may have a body.

Syntax:

```
access-specifier interface name_of_interface
{
        return-type method_name1(parameter-list);
```

```
        return-type method_name2(parameter-list);
        . . .
        type varname1 = value;
        type varname2 = value;
        . . .
}
```

Hear, access specifier is either **public** or any. When no access specifier is included, then default access specifier is **public**. interface is keyword. name_of_interface can be any valid identifier.

The methods which are declared in interface have no body. They end with semicolon after the parameter list. Each class that includes an interface must implements all of the methods .
Variable can be declared inside of interface declarations. They are implicitly final and static, meaning they can not be change by the implementing class.

Example:
Write a program that illustrates interface inheritance. Interface **P** is extended by **P1** and **P2**. Interface **P12** inherits from both **P1** ans **P2**.Each interface declares one constant and one method. class**Q** implements **P12**.Instantiate **Q** and invoke each of its methods. Each method displays one of the constants.

```
interface P
{
        int c=10;
        void display();
}
interface P1 extends P
{
        int c1=11;
        void display();
}
interface P2 extends P
{
        int c2=12;
        void display();
}
interface P12 extends P1,P2
{
        void display();
}
class Q implements P12
{
```

```
        public void display()
        {
                System.out.println(c);
                System.out.println(c1);
                System.out.println(c2);
        }
}
class pb1
{
        public static void main(String args[])
        {
                P p;
                p=new Q();
                p.display();
        }
}
```

Output:

```
10
20
30
```

4.7 instanceof operator **(Summer'13)**

"instanceof is an operator used to test if an object is instance of some class."

Syntax

```
name_of_object instanceof class_name
```

Consider the following example in which, instanceof operator is used.

Example: Write a program that demonstrates the **instanceof** operator.Declare interfaces **I1** and **I2**. Interface **I3** extends both of these interfaces.Also declare interface **I4**.class **X** implements **I3**.Class **W** extends **X** and implements **I4**.create an object of class **W**.Use the **instanceof** operator to test if that object implements each of the interfaces and is of type **X**.

```
interface I1
{
                void display();
}
interface I2
{
                void display();
}
interface I3 extends I1,I2
{
                void display();
```

```
}
interface I4
{
        void display();
}
class X implements I3
{
        public void display()
        {
                System.out.println("In class X");
        }
}
class W extends X implements I4
{

}
class pb2
{
        public static void main(String args[])
        {
                W w= new W();
                w.display();

               if(w instanceof I1)
                System.out.println("w is the instance of I1");
                if(w instanceof I2)
                System.out.println("w is the instance of I2");
                if(w instanceof I3)
                System.out.println("w is the instance of I3");
                if(w instanceof I4)
                System.out.println("w is the instance of I4");
        }
}
```

Output:

```
In class X
w is the instance of I1
w is the instance of I2
w is the instance of I3
w is the instance of I4
```

4.8 Dynamic Method dispatch (Winter'14, Nov-Dec'11)

The dynamic method dispatch is also called **run time polymorphism**. During the run time polymorphism, a call to overridden method is resolved at run time. The overridden method is called using the object of base class.

In below program, the **show** method is an overridden method because same signature we using in sub class. This method is invoked in the **main()** function using the reference object of base class(Bclass) that is **a2**. At run time it is decide that the **show** method of Bclass should be invoked. Hence it is known as run time polymorphism.

```
class Aclass
{
        void show()
        {
                System.out.println("in A class");
        }
}

class Bclass extends Aclass
{
        void show()
        {
                System.out.println("in B class");
        }
}
class pb1
{
        public static void main(String args[])
        {
                Aclass a1=new Aclass();
                a1.show();
                Aclass a2=new Bclass();          //object is reference to base class
                a2.show();
        }
}
```

Output:

```
in A class
in B class
```

4.9 Abstract class

"Abstract class is collection of abstract and non-abstract methods and data members."

When we create an abstract class, we are creating a base class that might have one or more completed methods but at least one or more methods are left uncompleted and declared abstract. If all the methods of an abstract class are uncompleted then it is same as an interface. The purpose of an abstract class is to provide a base class definition for how a set

of derived classes will work and then allow the programmers to fill the implementation in the derived classes.

If a class contain any abstract method then the class is declared as abstract class. An abstract class is never instantiated. It is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method.

Syntax :

```
abstract class class_name
{
        // body
}
```

Abstract method

Method that are declared without any body within an abstract class is known as abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax :

```
abstract return_typefunction_name ();    // No definition
```

Example: Create a class to find out whether the given year is leap year or not. (using abstact class)

```
abstract class A
{
        int year;
        abstract void Find();
}

class B extends A
{
        B(int y)
        {
         year = y;
        }
        void Find()
        {
         if((year % 4)==0)
         System.out.println(year+" is a leap year");
         else
         System.out.println(year+" is not a leap year");
        }

}
```

```
Class pb1
{
        Public static void main(String s[])
        {
                int a=Integer.parseInt(s[0]);
                B b=new B(a);
                b.Find();
        }
}
```

Output:

```
D:\>java pb1 2012
2012 is a leap year
```

Example: Describe abstract class called Shape which has three subclasses say Triangle, Rectangle and Circle. Define one abstract method area()in the abstract class and override this area() in these three subclasses to calculate for specific object i.e. area() of Triangle subclass should calculate area of triangle etc. Same for Rectangle and Circle.

```
abstract class Shape
{
        abstract void Area();
}

class Circle extends Shape
{
        float r;
        Circle(float R)
        {
           r=R;
        }
        void Area()
        {
           System.out.println("Area of Circle="+(3.14*r*r));
        }

}
class Triangle extends Shape
{       int b,h;
        Triangle(int B,int H)
        {
           b=B;
           h=H;
        }
        void Area()
```

```
                    {
                        System.out.println("Area of Triangle="+0.5*h*b);
                    }
}
class Rectangle extends Shape
{                   int b,l;
                    Rectangle(int L,int B)
                    {
                        l=L;
                        b=B;
                    }
                    void Area()
                    {
                        System.out.println("Area of Rectangle="+l*b);
                    }
}

class pb6
{
                    public static void main(String args[])
                    {

                        Circle c1= new Circle(3.0f);
                        Triangle t1= new Triangle(5,6);
                        Rectangle r1 =new Rectangle(9,10);
                        c1.Area();
                        t1.Area();
                        r1.Area();
                    }
}
```

Output:

```
Area of Circle=28.259
Area of Triangle=15.0
Area of Rectangle=90
```

4.10 Comparison between interface and abstract class. (Winter'14,Dec'10)

Interface	Abstract class
Interface is collection of abstract method.	Abstract class is collection of abstract and non-abstract method.
Interface keyword is use to declare interface.	Abstract keyword is use to declare abstract class.
Interfaces are used to implement multiple inheritances.	Abstract class does not support multiple inheritances.

By default all methods are abstract.	The default method is non abstract method
By default all variable are public, final and static.	Abstract class **can have final, non-final, static and non-static variables**.
interface cannot have constructor.	Abstract class can have constructor.
interface can't provide the implementation of its methods.	Abstract class can provide implementation of its methods.
interface **can't have static methods.**	Abstract class **can have static methods.**

Example:
public interface shape
{
void area();
}
	Example:
	public abstract class shape
	{
	abstract void area();
	}

4.11 Polymorphism (Winter'12,Dec'10)

Polymorphism is ability to take more than one forms. The word "poly" means many and "morphs" means forms.

There are two type of polymorphism.

1. Compile time polymorphism (static binding or early binding)
2. Run time polymorphism (dynamic binding or late binding)

static binding	dynamic binding
It is called compile time polymorphism.	It is called run time polymorphism.
Memory allocated at compiled time.	Memory allocated at run time.
Method overloading is example of static binding.	Method overriding is example of dynamic binding.
Hear, java compiler knows which method is called.	Hear, java compiler doesn't know which method is called.JVM decide which method is called at run time.
Private or final method is example of static binding.	Private or final method cannot be overridden

4.12Understanding of System.out.println()

```
System.out.println("Hello World");
```

1)System:

- It is the name of standard class that contains objects that encapsulates the standard **I/O** devices of your system. This class has a final modifier, which means that, it cannot be inherited by other classes.
- It is contained in the **package** java.lang. Since java.lang package is imported in every java program by default, therefore **java.lang package** is the only package in Java API which does not require an import declaration.

2)**out:**

- The object out represents output stream(i.e Command window)and is the static data member of the class **system**.
- So note here **System.out** (System -Class & out- static object i.e why its simply refered by classname and we need not to create any object). static fields and methods must be accessed by using the class name, so (System.out).

3).**println():**

- The println() is **method** of out object that takes the text string as an argument and displays it to the standard output i.e*on monitor screen*.
- println() is a public method in PrintStream class to print the data values. Hence to access a method in PrintStream class, we use out.println() (as non static methods and fields can only be accessed by using the refrencevarialble)

Note

System -Class
out -static Object
println() –method

Conceptually the System class looks like this:

```
class System
{
        public static final PrintStream out;
        //…
}
// the printstream class belongs to java.io package
class PrintStream {
        public void println();
 //…
}
```

4.13 Explain Cosmic superclass and its methods.

The **Object** class is the ultimate ancestor—every class in Java extends Object. However, you never have to write ‒class Employee extends Object‖.
In Java, every class that is defined without an explicit extends clause automatically extends the class
Object. That is, the class Object is the direct or indirect superclass of every class in Java.
The **Object class** is the parent class of all the classes in java bydefault. In other words, it is the topmost class of java.
The Object class is **beneficial if you want to refer any object whose type you don't know**.
Notice that **parent/super class reference variable can refer the child class object**, know as upcasting.

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.

protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Use Of toString() Example:

```
class A
{
        int a;
        A(int a)
        {
                this.a = a;
        }
        public String toString()
        {
                return "Class-A Object – a = " + a;
        }
}
class ToStringDemo
{
        public static void main(String args[])
        {
          A a1 = new A(10);
          System.out.println(a1); // Automatic call toString()
        }
}
```

Output:

Class-A Object – a =10 // Object String Value – no need display

4.14 Immutable objects and classes

The immutable classes are created to create immutable objects. The content of immutable objects cannot be changed.

Normally the **final** keyword is used to create immutable objects.

Following rules are used to create immutable objects and classes.

1) The class name must be declared as final

2) The data members in the class must be declared as final

3) The parameterized constructor is used.

Example

```
final class demo
{
        final int a;
        final String name;
        demo(int x,String s)
        {
                a=x;
                name=s;
        }
        void show()
        {
                System.out.println("a="+a);
                System.out.println("name="+name);
        }
}
class test
{
        public static void main(String arhs[])
        {
                demo d1=new demo(10,"AAA");
                d1.show();
                d1.a=100;//this line will generate an error
        }
}
```

4.15 Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

Methods of Java ArrayList

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Example:

```
import java.util.*;
class ArrayList1
{
        public static void main(String args[])
        {
                ArrayList list=new ArrayList();//Creating arraylist
                list.add("Ravi");//Adding object in arraylist
            list.add("Vijay");
            list.add("Ravi");
            list.add("Ajay");
            //Invoking arraylist object
            System.out.println(list);
            System.out.println("inserting elements in the array between");
                list.add(2,45);
            System.out.println(list);
            System.out.println("array size="+list.size());
        }
}
```

In above program,

1) We have created array list of String we can also pass some integer elements to array list.
2) As we insert the elements it grows and as we remove the elements it gets shrunk.

4.16 BigInteger and BigDecimal class

- The BigInteger and BigDecimal are the classes that are used to represent the integer or decimal number of any size and any precision.
- Both these classes are use from java.math package
- Both the classes are immutable
- The BigInteger class allows representation of and calculations on arbitrarily large integer (whole number).
- The BigDecimal class allows precise representation of any real number that can be with arbitrary precision.

Example of BigInteger

```
import java.math.*;
class demo
{
    public static void main(String args[])
    {
            BigInteger x=new BigInteger("12345678910111213141516171819");
            BigInteger y=new BigInteger("5");
            BigInteger z=x.multiply(y);
            System.out.println("Z="+z);
    }
}
```

Output:

Z=617283945505560657075808590 95

Example of BigDecimal

```
import java.math.*;
class demo
{
        public static void main(String args[])
        {
                BigDecimal x=new BigDecimal("1.0");
                BigDecimal y=new BigDecimal("6.0");
                BigDecimal z=x.divide(y,30,BigDecimal.ROUND_UP);
              System.out.println(z);
        }
}
```

Output:

Z= 0.166666666666666666666666666667

Example: Factorial of 50

```
import java.math.*;
class extra
{
        public static void main(String args[])
        {
                BigInteger fact=new BigInteger("1");
                for(int i=1;i<=50;i++)
                {
                        fact=fact.multiply(BigInteger.valueOf(i));
                }
                System.out.println("fact="+fact);
        }
}
```