UNIT II
Inheritance

Inheritance is the mechanism where an object of one class acquires the properties of object of another class.

- Inheritance allows the creation of hierarchical classifications of related classes.
- Inheritance supports code reusability

In Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits the instance variables and methods defined by the superclass and add its own, unique elements.
The general form of a **class** declaration that inherits a superclass is shown here:

class *subclass-name* extends *superclass-name* {
// body of class
}
You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass.
Access visibility

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Inheritance Basics
To inherit a class, you simply incorporate the definition of one class into another by using the e**xtends** keyword. The following program creates a superclass called **A** and a subclass called **B**.

```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + "" + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
```

```
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

```
/* In a class hierarchy, private members remain private to their class. This program contains an error and will not compile.*/
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

NOTE:*A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.*

Exercises

/*Declare an employee class with instance variables empid, name and salary and methods to input and display employee data. Extend this class in Programmer class. Programmer class has an instance variable bonus and method to input and display Programmer data and main() method to create and test programmer object*/

```
package UNIT2;
/*Declare an employee class with an instance variables empid, name and salary and methods to input and display employee data. Extend this class in Programmer class. Programmer class has an instance variable bonus and method to input and display Programmer data and main() method to create and test programmer object*/
class Employee1{
        private int empid;
        private String name;
        private   float salary;
        void readEmployee()
        {empid=100;
        name="NITTE";
        salary=2000000;
        }
        void displayEmployee()
        {System.out.println("Employee ID:"+empid);
         System.out.println("Employee Name:"+name);
         System.out.println("Employee Salary:"+salary);
        }
                }
class Programmer extends Employee1{
        int bonus;
        void readProgrammer()
        {bonus=100000;
        readEmployee();
                }
        void displayProgrammer()
        {
         displayEmployee();
         System.out.println("Bonus:"+bonus);
        }
        public static void main(String args[]){
          Programmer p=new Programmer();
          p.readProgrammer();
          p.displayProgrammer();
       }
        }
```

Output:

```
Employee ID:100
Employee Name:NITTE
Employee Salary:2000000.0
Bonus:100000
```

/*Demonstrate multilevel inheritance of Animal->Dog->BabyDog. Each class should have a method displaying a message specific to the animal. Create an object of each class and invoke the all the relevant methods*/

```
package UNIT2;
class Animal{
void eat(){System.out.println("eating…");}
}
class Dog extends Animal{
void bark(){System.out.println("barking…");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping…");}
}
class multiLevelInheritance{
public static void main(String args[]){
        Animal a=new Animal();
        System.out.println("Messages for Animal");
        a.eat();
        Dog d =new Dog();
        System.out.println("Messages for Dog Animal");
        d.eat();
        d.bark();
   BabyDog bd=new BabyDog();
   System.out.println("Messages for Baby Dog Animal");
   bd.eat();
   bd.bark();
   bd.weep();
}}
```

Output

```
Messages for Animal
eating…
Messages for Dog Animal
eating…
barking…
Messages for Baby Dog Animal
eating…
barking…
weeping…
```

1. //Animal class hierarchical inheritance

```
package UNITI;
class Animals{
void eat(){System.out.println("eating...");}
}
class Dog1 extends Animals{
void bark(){System.out.println("barking...");}
}
```

```
class Cat extends Animals{
void meow(){System.out.println("meowing...");}
}
class hierarchicalInheritance{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

```
//understand  toString()
package UNITI;

class Student{
        int rollno;
        String name;
        String city;
        Student(int rollno, String name, String city){
        this.rollno=rollno;
        this.name=name;
        this.city=city;
        }

        public String toString(){//overriding the toString() method
         return rollno+" "+name+" "+city;
        }
        public static void main(String args[]){
          Student s1=new Student(101,"XXX","Bangalore");
          Student s2=new Student(102,"YYY","Mangalore");

          System.out.println(s1);//compiler writes here s1.toString()
          System.out.println(s2);//compiler writes here s2.toString()
        }
                }
       Output:
101 XXX Bangalore
102 YYY Mangalore
```

Example

```
package UNIT2;
/*Design a Department class to have 2 private instance variables collegeName and DeptName and other
 * necessary setter and  getter methods. Extend Department class in Student class. Student class will
```

```
 *  have the private instance variables studName and studUsn and a method to check if the student is NMIT CSE student.
 *  If   yes  display such student detail. Student class can have other necessary setter and getter methods*/
class Department{
        private String collegeName;
        private String DeptName;
        String getcollegeName() {;
                return collegeName;
        }
        String getDeptName() {;
        return DeptName;
}
        void readDept()
        {
                collegeName="NMIT";
                DeptName="CSE";
        }

        void displayDept()
        {
        System.out.println("CollegeName:"+collegeName);
        System.out.println("DeptName:"+DeptName);
}
}
 class Students extends Department{
        private String studName;
        private String studUsn;
        String getstudName() {
                return studName;
}
        String getstudUsn() {
                return studUsn;
}
        void readStudent()
{  readDept();
studName="xxx";
studUsn="1NT21CS000";
}
void displayStudent()
{
        displayDept();
        System.out.println("StudentName:"+studName);
        System.out.println("Student Usn:"+studUsn);
}
void findCSE()
{if(getcollegeName()=="NMIT" && getDeptName()=="CSE")
        displayStudent();
}
```

```
public static void main(String args[]){
        Students s=new Students();
        s.readStudent();
        s.findCSE();
}
}
```

Output

```
CollegeName:NMIT
DeptName:CSE
StudentName:xxx
Student Usn:1NT21CS000
```

```
package UNIT2;
import java.util.Scanner;
/*Design a Department class to have 2 private instance variables collegeName and DeptName and other
 * necessary setter , getter and other methods. Extend Department class in nmitCseStudent class. Student class will
 *  have the private instance variables studName and studUsn and a method to check if the student is NMIT CSE student.
 *  If   yes  display such student detail. Student class can have other necessary setter ,  getter and other methods.
 *  Test nmitCseStudent class by creating an array of nmitCseStudent class in main. This array may have students of
 *  any college and any department. make the program interactive and user friendly*/
class Department1{

        private String collegeName;
        private String deptName;

        void setDepartment(String collegeName, String deptName) {
                this.collegeName = collegeName;
                this.deptName = deptName;
        }

        String getCollegeName() {
                return collegeName;
                }

        String getDeptName() {
                return deptName;
        }

        void displayDepartment() {
                System.out.println("College Name is:" + collegeName);
                System.out.println("Department Name is: " + deptName);
        }
        }

public class nmitCseStudents extends Department1 {
```

```
	private String stdName;
	private String stdUSN;

	void setSudent(String stdName,String stdUSN, String collegeName, String
deptName) {
		setDepartment(collegeName, deptName);
		this.stdName = stdName;
		this.stdUSN = stdUSN;
	}

	String getStdName() {
		return stdName;
	}

	String getStdUSN() {
		return stdUSN;
	}

	void displayStudent() {
		displayDepartment();
		System.out.println("Student name is: " + stdName);
		System.out.println("Student usn is: " + stdUSN);
	}

	void findCSE() {
		if(this.getCollegeName().contains("NMIT") &&
this.getDeptName().contains("CSE") ) {
			displayStudent();
		}
	}

	public static void main(String args[]) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the no of Students");
		int n = sc.nextInt();
		nmitCseStudents s[]= new nmitCseStudents[n];
		for (int i = 0; i < s.length; i++) {
			s[i] = new nmitCseStudents();
			System.out.println("Enter name");
			String name = sc.next();
			System.out.println("Enter usn");
			String usn = sc.next();
			System.out.println("Enter college");
			String college = sc.next();
			System.out.println("Enter dept");
			String dept = sc.next();
			s[i].setSudent(name, usn, college, dept);
		}
```

```
                for (int i = 0; i < s.length; i++) {
                        s[i].findCSE();
                        System.out.println();
                }
        }
}
```

OUTPUT

```
Enter the no of Students
5
Enter name
AAA
Enter usn
1NT20CS800
Enter college
NMIT
Enter dept
CSE
Enter name
BBB
Enter usn
1NT20CS801
Enter college
NMIT
Enter dept
ECE
Enter name
CCC
Enter usn
1NT20CS803
Enter college
NMIT
Enter dept
CSE
Enter name
DDD
Enter usn
1BY10CS900
Enter college
BMSIT
Enter dept
CSE
Enter name
EEE
Enter usn
1MS20ME800
Enter college
MSRIT
Enter dept
MECH
College Name is:NMIT
```

Department Name is: CSE
Student name is: AAA
Student usn is: 1NT20CS800

College Name is:NMIT
Department Name is: CSE
Student name is: CCC
Student usn is: 1NT20CS803

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass(inherited visible members).

Example

```
class RefDemo {
 public static void main(String args[]) {
 BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
 Box plainbox = new Box();
 double vol;
 vol = weightbox.volume();
 System.out.println("Volume of weightbox is " + vol);
 System.out.println("Weight of weightbox is " + weightbox.weight);
 System.out.println();
 // assign BoxWeight reference to Box reference
 plainbox = weightbox;
 vol = plainbox.volume(); // OK, volume() defined in Box
 System.out.println("Volume of plainbox is " + vol);
 /* The following statement is invalid because plainbox
 does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
 }
}
```

Here, weightbox is a reference to BoxWeight objects, and plainbox is a reference to Box objects. Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.

It is important to understand that it is the type of the reference variable—not the type
of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why plainbox can't access weight even when it refers to a BoxWeight object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a Box reference to access the weight field, because Box does not define one

Using super

super has two general forms.
-The first calls the superclass' constructor.
-The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of super:

super(*arg-list*);

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( ) must always be the first statement executed inside a subclass' constructor.**

Example:

```
// A complete implementation of BoxWeight. Use super()
class Box {
private double width;
private double height;
private double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
```

```
return width * height * depth;
}
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
}}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
```

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
```

```
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. **super** can also be used to call methods that are hidden by a subclass.

When Constructors Are Called(order of execution of constructors)

When a class hierarchy is created, Constructors are called in order of derivation, from superclass to subclass. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
```

```
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

The output from this program is shown here:
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Consider the following:

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + "" + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
```

```
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

k: 3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**. Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements **run-time polymorphism**.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
```

```
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

NOTE *Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.*

Why Overridden Methods?

Overridden methods allow Java to support **run-time polymorphism**. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on **code reuse** and **robustness**. **The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface** is a profoundly powerful tool

Method Overloading vs. Method Overriding

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
It helps to increase the readability of the program.	It is used to provide the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.

Method Overloading	Method Overriding
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or cannot be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Better Performance due to compile time polymorphism.	Performance not as good as compile time polymorphism. Flexible
Private and final methods can be overloaded.	Private and final methods can't be overridden.
Argument list should be different while doing method overloading.	Argument list should be same in method overriding.

Applying Method Overriding

The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area( )** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is undefined.");
return 0;
}
}
class Rectangle extends Figure {
```

```
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class FindAreas {
public static void main(String args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Exercise

```
package UNIT2;
```

```
import java.util.Scanner;
/* Define a shapes class with an instance variable dim and a method area(). There is no specific dim w.r.t. a Shapes
 *  object. Extend Shapes class into 2 other classes, Circle and Square classes. Override area()
 *  method in Circle and Square classes to find area of Circle and Square objects respectively. Use
 *  dynamic method dispatch concept(run-time polymorphism )to invoke area() method of all the 3 classes.  */
class Shapes {
	double dim;
	Shapes(double a) {
		dim = a;
	}
	void area() {
	  System.out.println("Area undefined for Shapes : "+ dim);
	}
}
class Circle extends Shapes {
	Circle(double a) {
		super(a);
	    }
  public void area() {

		System.out.println("Area of the Circle is : "+3.14*dim*dim );
  }
}
class Square extends Shapes {
	Square(double a) {
		super(a);
	    }
    public void area() {

    System.out.println("Area  of the Square is : "+  dim*dim);
  }
}
class dynamicPoly
{
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    Shapes myShapeRef;
    Shapes myShape = new Shapes(4);  // Create a Shapes object
    myShapeRef=myShape;
    myShapeRef.area();
    System.out.println("Enter Radius of the Circle : ");
    Circle myCircle = new Circle(sc.nextDouble());  // Create a Circle object
    myShapeRef=myCircle;
    myShapeRef.area();
    System.out.println("Enter side of the Square : ");
    Square mySquare = new Square(sc.nextDouble());  // Create a Square object
    myShapeRef=mySquare;
```

```
            myShapeRef.area();
          }
        }
```

Output

```
Area undefined for Shapes : 4.0
Enter Radius of the Circle :
4
Area of the Circle is : 50.24
Enter side of the Square :
5
Area  of the Square is : 25.0
```

Exercise

Write similar program to find area of a Circle and a Ractangle

Using Abstract Classes

Any class that contains one or more abstract methods. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

To declare an abstract method, use this general form:
abstract *type name(parameter-list)*;

Abstract methods are used when a superclass is unable to create a meaningful implementation for a method.

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
```

```
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.
Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

```
// Using abstract methods and classes…Figure class
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for triangle
double area() {
```

```
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

Using final with Inheritance

The keyword **final** has three uses:

1. **It can be used to create the equivalent of a named constant**. Refer previous chapter

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.

It is a common coding convention to choose all uppercase identifiers for **final** variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The other two uses of **final** apply to inheritance.

2. **Using final to Prevent Overriding**

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.

Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a **performance enhancement**: The compiler is free to ***inline*** calls to them because it "knows" they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding.*

3. Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Packages

Package is a mechanism for partitioning the class name space into more manageable chunks. The package is both a naming and a visibility control mechanism. You can define classes inside

a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package.

Defining a Package

To create a package simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

The general form of the **package** statement:
package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same **package** statement.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:
package *pkg1*[.*pkg2*[.*pkg3*]];
A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package java.awt.image; needs to be stored in **java\awt\image** in a Windows environment.

Finding Packages and CLASSPATH

Packages are mirrored by directories. This raises an important question:
How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:
package MyPack
In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.
When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

C:\MyPrograms\Java\MyPack then the class path to **MyPack** is C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("-->");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.) As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay

between classes and packages, **Java addresses four categories of visibility for class members:**

- **Subclasses in the same package**
- **Non-subclasses in the same package**
- **Subclasses in different packages**
- **Classes that are neither in the same package nor subclasses**

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table show below sums up the interactions. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Above applies only to members of classes.

A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**. The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

This is file **Protection.java**:

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);

}
}
```

This is file **Derived.java**:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **SamePackage.java**:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {

System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **OtherPackage.java**:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you want to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
```

}

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

import *pkg1* [*.pkg2*].(*classname* | *);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

import java.lang.*;

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package. you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the import statement looks like this:

```
class MyDate extends java.util.Date {
}
```

In this version, **Date** is fully-qualified.

As shown in Table 9-1, when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if

you want the **Balance** class of the package **MyPack** shown earlier to be available as a standalone
class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;
/* Now, the Balance class, its constructor, and its show() method are public. This means that they can be used by non-subclass code outside their package.*/
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("-->");
System.out.println(name + ": $" + bal);
}
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {

/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

Interfaces

- Interfaces support the "**one interface, multiple methods**" aspect of polymorphism.
- Using the keyword **interface**, you can **fully abstract a class' interface from its implementation**. That is, using **interface**, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but will not have instance variables and their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

- To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation.
- Interfaces are designed to support dynamic method resolution at run time.
- Interfaces disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

***access* interface *name* {**
***return-type method-name1*(*parameter-list*);**
***return-type method-name2*(*parameter-list*);**
***type final-varname1 = value*;**
***type final-varname2 = value*;**
//...
***return-type method-nameN*(*parameter-list*);**
***type final-varnameN = value*;**
}

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

As the general form shows, variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Example

```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface*...]] {
// class-body
}

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared

public. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier:

```
class Client implements Callback {
// Implement Callback's interface

public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that **callback( )** is declared using the **public** access modifier. **REMEMBER** When you implement an interface method, it must be declared as public. It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " + "may also define other members, too.");
}
}
```

Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee." This process is similar to using a superclass reference to access a subclass object.

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**. While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**,

```
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
}
}
```

shown here:

```
// Another implementation of Callback.
```

Now, try the following class:

```
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

As you can see, the version of **callback( )** that is called is determined by the type of object that **c** refers to at run time.

Exercise 1

```
package UNITI;

//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
```

```
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
Rectangle r=new Rectangle();
d=r;
d.draw();
}}
```

Output:

```
drawing circle
drawing rectangle
```

Exercise 2

```
package UNITI;
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("SBI ROI: "+b.rateOfInterest());
PNB p =new PNB();
b=p;
System.out.println("PNB ROI: "+b.rateOfInterest());

}}
```

OUTPUT:

```
SBI ROI: 9.15
PNB ROI: 9.7
```

Exercise 3:Multiple Inheritance

```
package UNITI;
interface Printable{
void print();
}
interface Showable{
void print();
}
  class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Output:

Hello

Partial Implementations

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**. For example:

```
abstract class Incomplete implements Callback {
int a, b;
void show() {
System.out.println(a + "" + b);
}
//...
}
```

Here, the class **Incomplete** does not implement **callback( )** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

Exercise 1

Make an interface called interface1 that has two data fields- default_length=2 and Pi=3.14
Make a second interface called interface2 that has one method: double getPerimeter(),
Make an abstract class GeometricObject that implements the 2 interfaces. And make one method getArea as an abstract method.
Make a subclass circle of GeometricObject. The circle class will inherit the superclass methods and implements the required abstract methods.
Make a subclass Triangle of GeometricObject. (This is an equilateral triangle. P=3a, $A = \frac{\sqrt{3}}{4}a^2$). Triangle class will inherit the superclass methods and implements the required abstract methods.
In another class called test, make run-time polymorphism call to objects of circle and triangle class.

Note:import java.lang.Math; Math.sqrt(3)

```
package UNIT2;
import java.lang.Math;
interface I1{
	int default_length=2;
	double PI=3.14;
}
interface I2{
	double getPerimeter();
}
abstract class GeometricObject implements I1,I2{
	abstract double getArea() ;


}
class Circle1 extends GeometricObject{
	public double getPerimeter() {
		return 2*PI*default_length;
	}
	public double getArea() {
```

```
                return PI*default_length*default_length;
        }
}
class Triangle1 extends GeometricObject{
        public double getPerimeter(){return 3*default_length;}
        public double getArea() {
                return Math.sqrt(3)*default_length*default_length/4;
        }
}
class Test{
        public static void main(String args[]) {
                GeometricObject g;
                Circle1 c =new Circle1();
                Triangle1 t=new Triangle1();
                g=c;
                System.out.println("Area of Circle=  "+ g.getArea());
                g=t;
                System.out.println("Area of Triangle=  "+ g.getArea());
                        }
        }
```

Output

```
Area of Circle=  12.56
Area of Triangle=  1.7320508075688772
```

Nested Interfaces

An interface can be declared as a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// A nested interface example. This class contains a member interface.
class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);
}
}
// B implements the nested interface.
class B implements A.NestedIF {
public boolean isNotNegative(int x) {
return x < 0 ? false: true;
}
}
class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
```

```
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be displayed");
}
}
```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying implements A.NestedIF Notice that the name is fully qualified by the enclosing class' name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

Applying Interfaces

Let's look at two examples. First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations-FixedStack and DynStack.

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop(); // retrieve an item
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
FixedStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
if(tos==stck.length-1) // use length member
System.out.println("Stack is full.");
else
stck[++tos] = item;
}

// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
```

```
}
}
class IFTest {
public static void main(String args[]) {
FixedStack mystack1 = new FixedStack(5);
FixedStack mystack2 = new FixedStack(8);
// push some numbers onto the stack
for(int i=0; i<5; i++) mystack1.push(i);
for(int i=0; i<8; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
System.out.println(mystack2.pop());
}
}
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// Implement a "growable" stack.
class DynStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
// if stack is full, allocate a larger stack
if(tos==stck.length-1) {
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++) temp[i] = stck[i];

stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
```

```
else
return stck[tos--];
}
}
class IFTest2 {
public static void main(String args[]) {
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);
for(int i=0; i<20; i++) mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.

```
/* Create an interface variable and access stacks through it.*/
class IFTest3 {
public static void main(String args[]) {
IntStack mystack; // create an interface reference variable
DynStack ds = new DynStack(5);
FixedStack fs = new FixedStack(8);
mystack = ds; // load dynamic stack
// push some numbers onto the stack
for(int i=0; i<12; i++) mystack.push(i);

mystack = fs; // load fixed stack
for(int i=0; i<8; i++) mystack.push(i);
mystack = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
System.out.println(mystack.pop());
mystack = fs;
System.out.println("Values in fixed stack:");
for(int i=0; i<8; i++)
System.out.println(mystack.pop());
}
}
```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

When you "implement" the interface, all of those variable names will be in scope as constants. If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables.

Example --- an automated "decision maker" depending on the random number generated

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)

return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
```

```
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Notice that this program makes use of one of Java's standard classes: **Random**. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble( )** is used. It returns random numbers in the range 0.0 to 1.0. In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
```

```
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

As an experiment, you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

Exercise

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Default Interface Methods

Prior to JDK 8, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface. The release of JDK 8 has changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. During its development, the default method was also referred to as an *extension method.*

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. If a new method is added to a popular, widely used traditional interface, then the addition of that method would break existing code

because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, **the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class.**

Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {
// This is a "normal" interface method declaration. It does NOT define a default implementation.
int getNumber();
// This is a default method. Notice that it provides a default implementation.
default String getString() {
return "Default String";
}
}
// Implement MyIF.
class MyIFImp implements MyIF {
// Only getNumber() defined by MyIF needs to be implemented.
// getString() can be allowed to default.
public int getNumber() {
return 100;
}
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber( )** and **getString( )**.

```
// Use the default method.
class DefaultMethodDemo {
public static void main(String args[]) {
MyIFImp obj = new MyIFImp();
// Can call getNumber(), because it is explicitly implemented by MyIFImp:
System.out.println(obj.getNumber());
// Can also call getString(), because of default implementation:
System.out.println(obj.getString());
}
}
```

The output is shown here:
100
Default String

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString( )**: class MyIFImp2 implements MyIF {
// Here, implementations for both getNumber( ) and getString( ) are provided.
public int getNumber() {
return 100;
}
public String getString() {
return "This is a different string.";
}
}
Now, when **getString( )** is called, a different string is returned.
Exercise

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Multiple Inheritance Issues

Java does not support the multiple inheritance of classes. Even interfaces do not support multiple inheritance. There is a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot. To a limited extent, default methods do support multiple inheritance of behavior. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. In such a situation, it is possible that a name conflict will occur.
For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both **Alpha** and **Beta** provide a method called **reset( )** for which both declare a default implementation? Is the version by **Alpha** or the version by **Beta** used by **MyClass**? Or, consider a situation in which **Beta** extends **Alpha**. Which version of the default method is used? Or, what if **MyClass** provides its own implementation of the method?

To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

- First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if **MyClass** provides an override of the **reset( )** default method, **MyClass**' version is used. This is the case even if **MyClass** implements both **Alpha** and **Beta**. In this case, both defaults are overridden by **MyClass**' implementation.
- Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result. Continuing with the example, if **MyClass** implements both **Alpha** and **Beta**, but does not override **reset( )**, then an error will occur.
- In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset( )** will be used.

- It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**. Its general form is shown here:

  InterfaceName*.super.*methodName( )

For example, if **Beta** wants to refer to **Alpha**'s default for **reset( )**, it can use this statement:

Alpha.super.reset();

Use static Methods in an Interface

JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a **static** method in a class is called. The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber( )**. It returns zero.

```
public interface MyIF {
// This is a "normal" interface method declaration. It does NOT define a default implementation.
int getNumber();
// This is a default method. Notice that it provides a default implementation.
default String getString() {
return "Default String";
}

// This is a static interface method.
static int getDefaultNumber() {
return 0;
}
```

}

The **getDefaultNumber( )** method can be called, as shown here:
int defNum = MyIF.getDefaultNumber();

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber( )** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.