05 Implementing Inheritance in Java

IN THIS CHAPTER

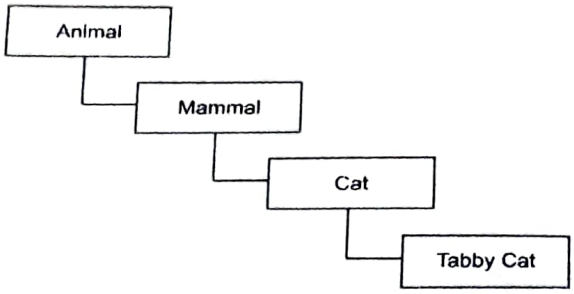
Inheritance is one of the most important features of Object Oriented Programming (OOP). It facilitates the creation of a new class by extending a previously declared class. The class that is used to create a new class is called a base class, superclass or parent class, and the newly created class is called a subclass, child class or derived class. The process of creating a derived class from a base class is called inheritance.

This chapter discusses the various forms of inheritance in Java. It describes the concept of inheriting `public`, `protected` and `private` members. This chapter also explains how to refer to a subclass object, use the `super` keyword, and invoke the `super()` and `this()` constructors. It also discusses how to use the `final` keyword and declare an `abstract` class. Moreover, this chapter describes the interfaces in Java, including their definition and implementation, and also describes how to extend interfaces and access `interface` variables. Finally, this chapter discusses class hierarchy in Java.

We begin by describing the concept of inheritance.

5.1 Understanding Inheritance

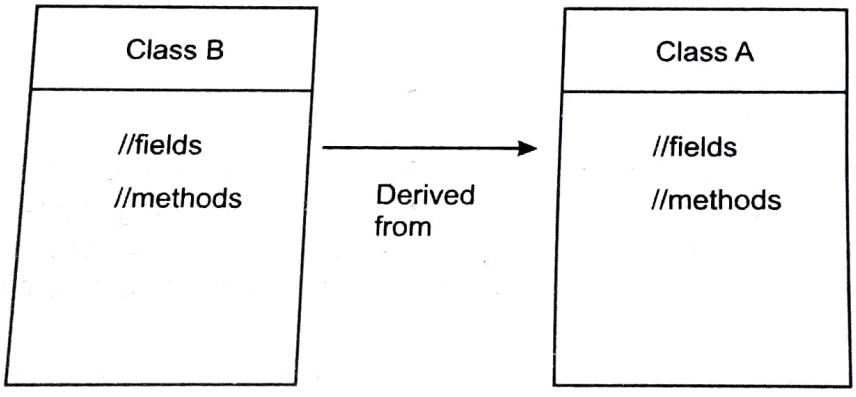
Inheritance facilitates the creation of a subclass from a superclass by inheriting the properties of the superclass. The relationship between a subclass and superclass forms a hierarchy, which can be graphically represented. For example, the inheritance relationship between the `Animal`, `Mammal`, `Cat`, and `TabbyCat` classes form an inheritance hierarchy, as shown in Figure 1:



▲ **Figure 1: Representing the Inheritance Hierarchy**

In Figure 1, the inheritance hierarchy of the `TabbyCat` class is represented as deriving from the `Cat` class. Moreover, the `Mammal` class is the superclass of the `Cat` class and the subclass of the Animal class.

In `Java`, inheritance is implemented by extending a class and adding fields and methods in its subclass. Let's consider two classes, class A and class B, where class B is formed by extending class A. In this case, class B is called a subclass and class A is called a superclass. Generally, the subclass inherits the structure and behavior of the superclass, but the subclass can also add its own members. Figure 2 shows the relation between a subclass and a superclass:



▲ **Figure 2: Class B Inherited from Class A**

Figure 2 represents the relation between a subclass and a superclass. The arrow towards class A denotes that class B is derived from class A.

Programmatically, a derived class is created by using the `extends` keyword. The following code snippet shows how to declare the base class A and the derived class B:

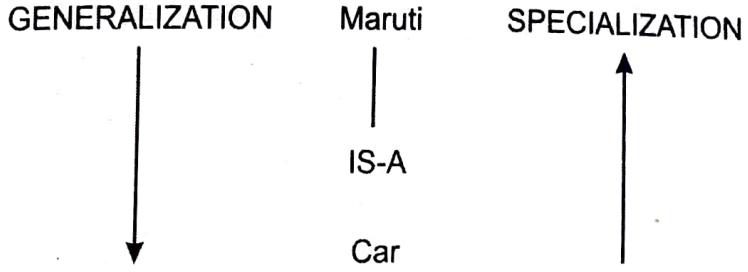
```
class A
{
//members of class A
}
class B extends A
{
// members of class B
}
```

In the preceding code snippet, the derived B class is extended by class A by using the extends keyword.

Inheritance is an important feature of OOP, which helps to improve the code structure and enhance programming flexibility. A key advantage of inheritance is that it allows code to be reused. This reusability of code ensures that programs are written only once and used many times with slight modifications. This feature relieves a programmer from writing the same code repeatedly, thereby saving both time and effort.

Inheritance also helps in the refinement of code, which allows you to improve an existing code of a superclass by modifying its subclass. For example, in case of method overriding, you can implement a method in a subclass with the same name as in the superclass. However, the method definition is different in the subclass.

Inheritance is analogous to the generalization and specialization relation between classes. For example, the `Car` superclass is a generalization of the `Maruti` subclass. Moreover, the `Maruti` subclass also inherits those properties of the `Car` superclass that are common to cars, such as four wheels, brakes and an engine. Figure 3 shows the generalization and specialization relation between the `Car` and `Maruti` classes:



▲ **Figure 3: Displaying Generalization and Specialization in a Class**

Figure 3 represents the generalization - specialization relation between the Car and Maruti classes. In this figure, the upward arrow represents specialization and the downward arrow represents generalization.

Now, let's consider the various forms of inheritance that are implemented in Java programming.

5.2 Using Forms of Inheritance

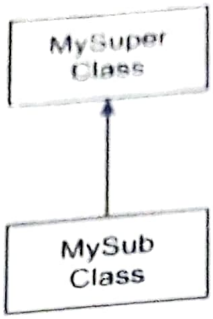
Inheritance is an extension of a previously declared base class. In practice, a class can be inherited from more than one class. Depending on the number and nature of a class, inheritance is classified into the following forms:

- Single inheritance
- Hierarchical inheritance
- Multilevel inheritance

Let's discuss these forms one by one.

Implementing Single Inheritance

Single inheritance is the most common type of inheritance. It is simply creating a derived class from a single base class. It represents a linear relationship between a superclass and its subclass. Figure 4 shows a single inheritance relationship between two classes, `MySuper` and `MySub`:



▲ Figure 4: Representing Single Inheritance

Figure 4 represents a single inheritance relationship between the MySuper and M
classes. An arrow pointing towards the MySuper class denotes that the MySub class
subclass and is inherited from the MySuper class, which is a superclass.

The implementatic of single inheritance is simple and can be easily programmed. Let
consider the MySub and MySuper classes, which are a subclass and superclass, respectively
according to the inheritance principle. Listing 1 shows the MySuper class (you can find the
MySuper.java file on the CD in the code\chapter5 folder):

▶ **Listing 1: The MySuper Class**

```
//declaring class Super
class MySuper {
  public void msgFromSuper() {
      System.out.println("In Superclass");
  }
}
```

In Listing 1, the MySuper class is declared, which contains the msgFromSuper() method.
The msgFromSuper() method is marked public so that it can be accessed by the subclass.
Next, the MySub class is created, which extends the MySuper class and which contains a new
method, msgFromSub(). The MySub class is now the subclass as per the inheritance
relationship. Listing 2 shows the declaration of the MySub class (you can find the MySub.java
file on the CD in the code\chapter5 folder):

▶ **Listing 2: The MySub Class**

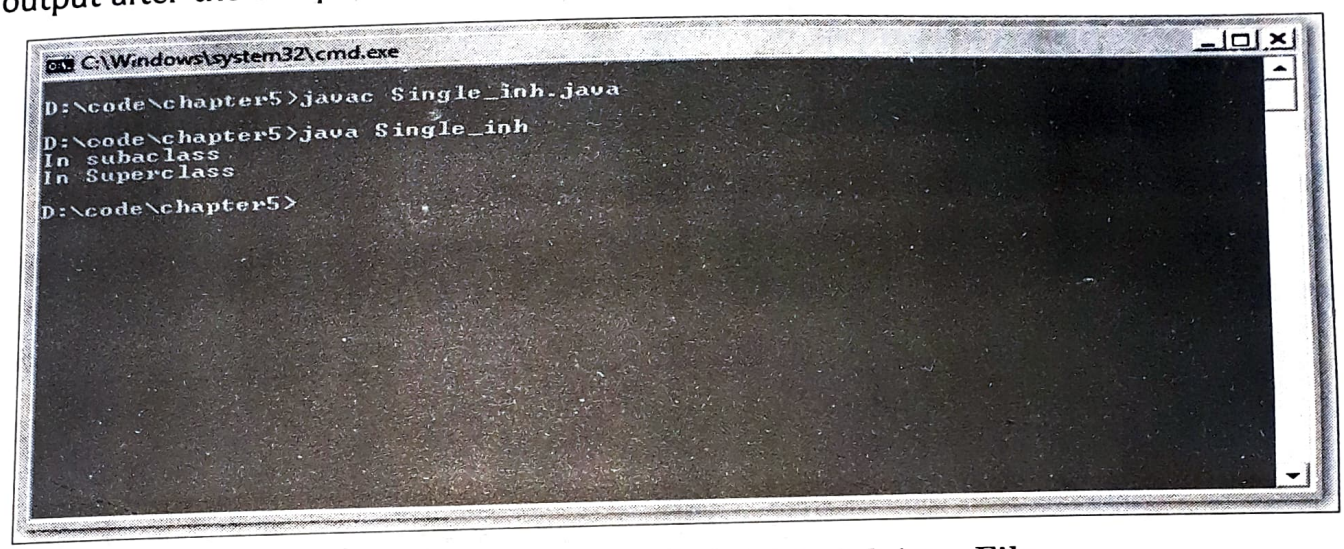
```
// declaring subclass
class MySub extends MySuper
{
  //declaring a method of the subclass
  public void msgFromSub(){
      System.out.println ("In subclass");
  }
}
```

In Listing 2, the MySub class is declared by extending the MySuper class and it defines its
own method, that is, msgFromSub(). The MySub class inherits the members of its
superclass, MySuper. Listing5.3 shows the Single_inh class, which contains the main()
method and creates an instance of the MySub class (you can find the Single_inh.java file on
the CD in the code\chapter5 folder):

▶ **Listing 3: The Single_inh Class**

```
public class Single_inh
{
  public static void main(String[] args)
  {
      MySub c = new MySub();
      c.msgFromSub();
      c. msgFromSuper();
  }
}
```

In Listing 3, the `Single_inh` class is declared, which accesses the methods of the `MySuper` superclass and the `MySub` subclass by creating an instance of the subclass. Figure 5 shows the output after the compilation and execution of the `Single_inh` class:



▲ **Figure 5: Executing the Single_inh.java File**

In Figure 5, the methods of the `MySuper` superclass and `MySub` subclass are invoked and the messages are accordingly displayed.

TEST YOUR KNOWLEDGE

Q1. Consider an employee class, which contains fields such as name and desg, and a subclass, which contains a field sal. Write a program for inheriting this relation.

Ans.

```
class Emp {
  String name = "Niraj";
  String desg = "Programmer";
  public void display1()
  {
      System.out.println("NAME:" + name);
      System.out.println("DESG:" + desg);
  }
}
class TestYourKnowledge1 extends Emp
{
  int sal = 5000;
  public void display2()
  {
      System.out.println("SALARY:" + sal);
```

```
    }
    public static void main(String s[])
    {
        TestYourKnowledge1 obj = new TestYourKnowledge1 ();
        obj.display1();
        obj.display2();
    }
}
```

Execution:

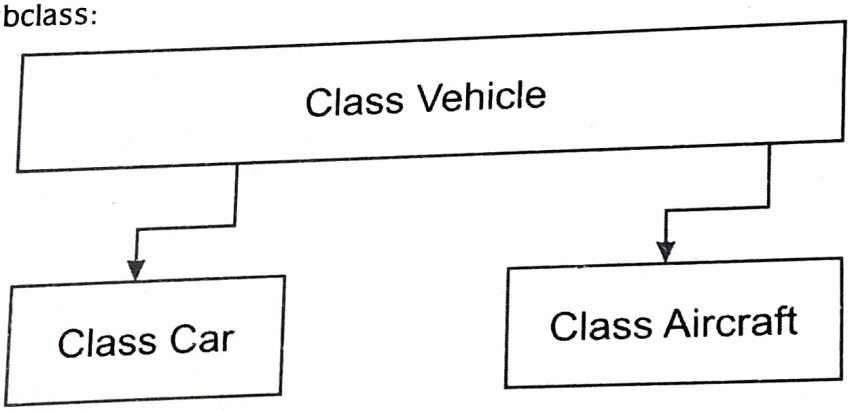
```
D:\code\chapter5>javac TestYourKnowledge1.java
D:\code\chapter5>java TestYourKnowledge1
```

Output:

```
NAME: Niraj
DESG: Programmer
SALARY: 5000
```

■ Implementing Hierarchical Inheritance

The next form of inheritance is hierarchical inheritance, in which more than one subclass is derived from a single superclass. Figure 6 shows a hierarchical relationship between a superclass and subclass:



▲ **Figure 6: Representing Hierarchical Inheritance**

In Figure 6, two classes, `Car` and `Aircraft`, are derived from the `Vehicle` class. Therefore, as shown in Figure 6, it is possible to inherit more than one subclass from a single superclass in Java. This is known as hierarchical inheritance. Let's consider a class named `Vehicle` with the `start()` method. The `Vehicle` class has a subclass named `Car` with the `drive()` method, as shown in Listing 4 (you can find the Car.java file on the CD in the code\chapter5 folder):

▶ **Listing 4: Vehicle Class and its Car Subclass**

```
class Vehicle
{
  public void start()
  {
      System.out.println("Starting a vehicle");
  }
}
```

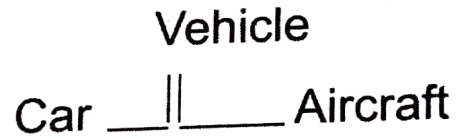
```
class Car extends Vehicle
{
  public void drive()
  {
     System.out.println("Driving a Car");
  }
}
```

In Listing 4, we create the `Vehicle` class, which contains the `start()` method, and a subclass, `Car`, which contains the `drive()` method Next, we create a new class, `Aircraft`, which contains the `fly()` method. The `Aircraft` class extends the `Vehicle` class, as shown in Listing 5 (you can find the Aircraft.java file on the CD in the code\chapter5 folder):

▶ Listing 5: The Vehicle Class and the Aircraft Subclass

```
class Aircraft extends Vehicle
{
  public void fly()
  {
     System.out.println("Flying a plane");
  }
}
```

Listing 5 shows the `Aircraft` class that is derived from the `Vehicle` class. Figure 7 shows the inheritance hierarchy of this inheritance relationship:



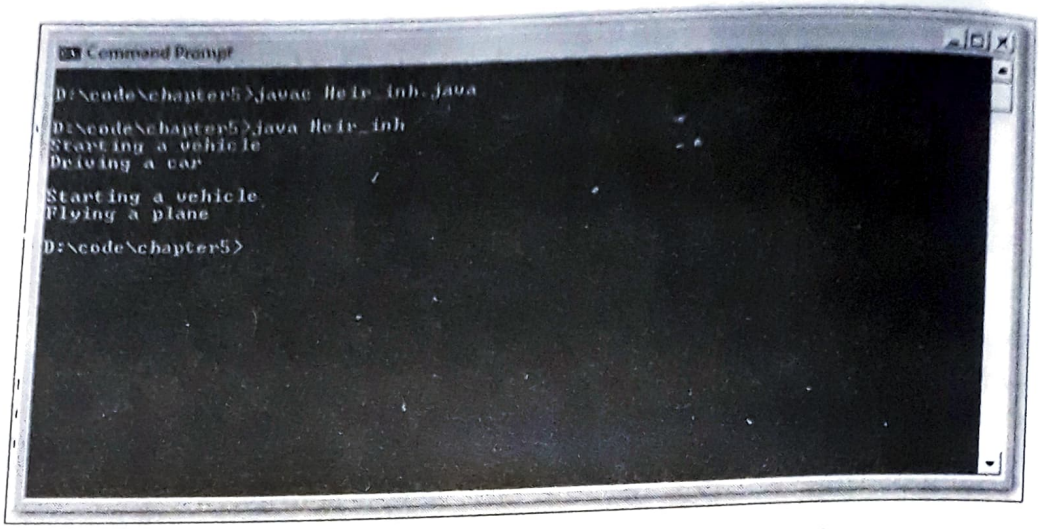
▲ Figure 7: Representing the Inheritance Hierarchy Formed by Hierarchical Inheritance

Figure 7 displays the inheritance hierarchy of the hierarchical inheritance relationship. To understand the implementation of the hierarchical inheritance relationship, let's create the `Heir_inh` class, which implements this type of relationship, as shown in Listing 6 (you can find the Heir_inh.java file on the CD in the code\chapter5 folder):

▶ Listing 6: The Heir_inh Class

```
public class Heir_inh {
  public static void main(String[] args)
  {
     Car c = new Car(); c.start(); c.drive();
     System.out.println();
     Aircraft j = new Aircraft (); j.start(); j.fly();
  }
}
```

Listing 6 shows the creation of the instances of the Car and Aircraft classes. When we execute the Heir_inh class, the output appears as shown in Figure 8:



▲ Figure 8: Executing the Heir_inh.java File

TEST YOUR KNOWLEDGE

Q2. Write a program for implementing hierarchical inheritance.

Ans.

```
class Super
{
     public void msgsuper() { System.out.println("Executing
        Superclass"); }
}
class Sub1 extends Super {
 public void msgsub1() { System.out.println("Executing Subclass one");
}
}
class TestYourKnowledge2 extends Super {
 public void msgsub2() { System.out.println("Executing Subclass two");
}
 public static void main(String[] args)
 {
     Sub1 c = new Sub1();
     c.msgsuper();
     c.msgsub1();
     System.out.println();
     c.msgsuper();
     TestYourKnowledge2 j = new TestYourKnowledge2 ();
     j.msgsub2();
 }
}
```

Execution:

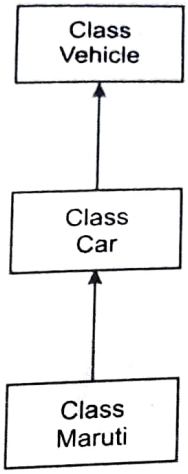
```
D:\code\chapter5>javac TestYourKnowledge2.java
D:\code\chapter5>java TestYourKnowledge2
```

Output:

```
Executing Superclass
Executing Subclass one
Executing Superclass
Executing Subclass two
```

Implementing Multilevel Inheritance

To understand multilevel inheritance, take the example of a class, Maruti, which is derived from the Car class, which in turn is derived from the Vehicle class. This type of relationship is called multilevel inheritance. Figure 9 shows multilevel inheritance:



▲ **Figure 9: Representing Multilevel Inheritance**

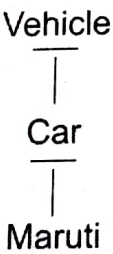
Figure 9 shows multilevel inheritance. Here, the Car class is derived from the `Vehicle` class and the Maruti class is derived from the Car class.

It is possible to implement multilevel inheritance in Java, as shown in Listing 4, where we created the `Vehicle` class and its subclass `Car`. Here, let's create a new class `Maruti`, which is the subclass of the `Car` class. Listing 7 shows the declaration of the `Maruti` class, which extends the `Car` class and adds the `own()` method (you can find the Maruti.java file on the CD in the code\chapter5 folder):

▶ **Listing 7: The Maruti Subclass of the Car Class**

```
class Maruti extends Car
{
  public void own()
  {
     System.out.println("Owing a Maruti");
  }
}
```

In Listing 7, the Maruti class is created from the Car superclass and contains the own() method. The inheritance relationship between the Vehicle, Car, and `Maruti` classes form an inheritance hierarchy, as shown in Figure 10:



▲ **Figure 10: Representing the Inheritance Hierarchy among the Vehicle, Car, and Maruti Classes**

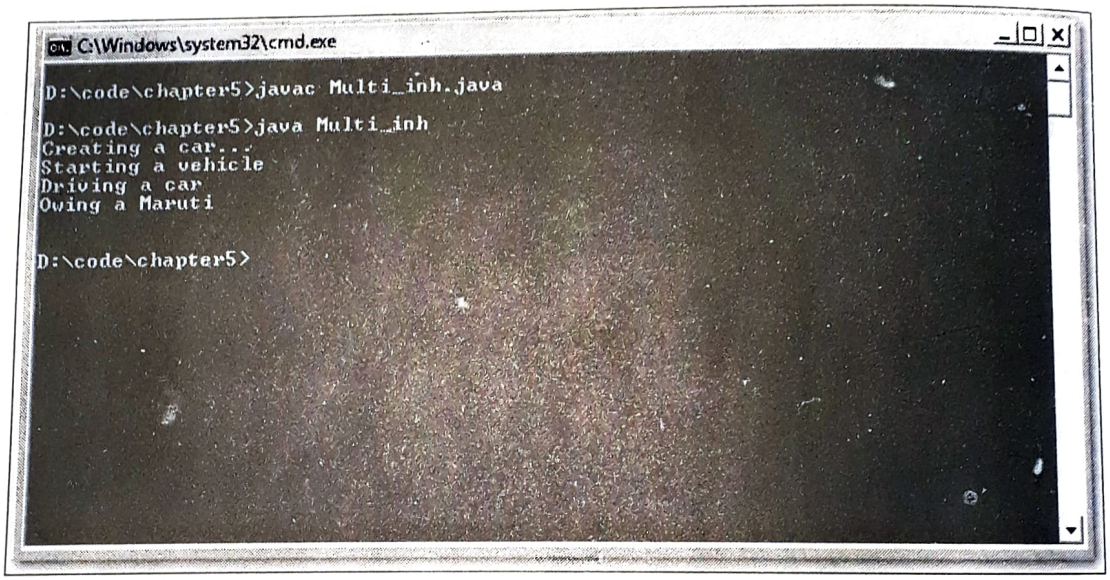
Figure 10 shows the inheritance hierarchy of the Vehicle, Car, and Maruti classes, where the Car class is the subclass of the Vehicle class and the Maruti class is the subclass of the Car class.

In Listing 8, the `Multi_inh` class is declared, which creates the instance of the Maruti class (you can find the Multi_inh.java file on the CD in the code\chapter5 folder):

▶ **Listing 8: The Multi_inh Class**

```
public class Multi_inh {
  public static void main(String[] args) {
      System.out.println("Creating a Car...");
      Maruti c = new Maruti();
      c.start();
      c.drive();
      c.own();
      System.out.println();
  }
}
```

In Listing 8, the `Multi_inh` class creates the instance of the derived class (Maruti) and then invokes the methods of the Car and Vehicle superclasses. Figure 11 shows the output when the `Multi_inh` class is compiled and executed:



▲ **Figure 11: Executing the Multi_inh.java File**

Figure 11 displays the output of the `Multi_inh` class. In the figure, the methods of the `Vehicle` and `Car` classes are executed by the object of the `Maruti` class.

Java does not support multiple inheritance, wherein a base class can extend more than one superclass at a time. Therefore, we shall not discuss multiple inheritance.

TEST YOUR KNOWLEDGE

Q3. Write a program for implementing multilevel inheritance.

Ans.

```
class Book {
  public void buy() { System.out.println("Purchaged a book"); }
}
```

```
class Title extends Book {
 public void name() { System.out.println("Java Programming"); }
}
class TestYourKnowledge3 extends Title
{
 public void auth() { System.out.println("Khalid Mugal"); }
 public static void main(String[] args) {
     System.out.println("Shopping Schedule");
     TestYourKnowledge3 a = new TestYourKnowledge3();
 a.buy(); a.name();a.auth();
  }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge3.java
D:\code\chapter5>java TestYourKnowledge3
```

Output:

```
Shopping Schedule
Purchased a book
Java Programming
Khalid Mugal
```

Now that you have a fairly good idea of the various forms of inheritance in Java, let's move on and discuss how private, protected, and public members of a superclass are inherited in a subclass.

3 Identifying Inheritance and Member Accessibility

In this section, the main objective is to test the behavior of inherited members according to their access modifiers. During inheritance, the behavior of private, protected and public members are different from one another. In the following sections, we describe the inheritance of the public, private, and protected members of a superclass.

Using the public Access Modifier

The public access modifier is the least restrictive of all the access modifiers. In other words, a public variable can be accessed within the class in which it is declared or within its subclass. In addition, a public variable is also accessible from any class inside and outside the package it is declared. If the members of a superclass are public, then they can be accessed from a subclass either inside or outside a package without any difficulty.

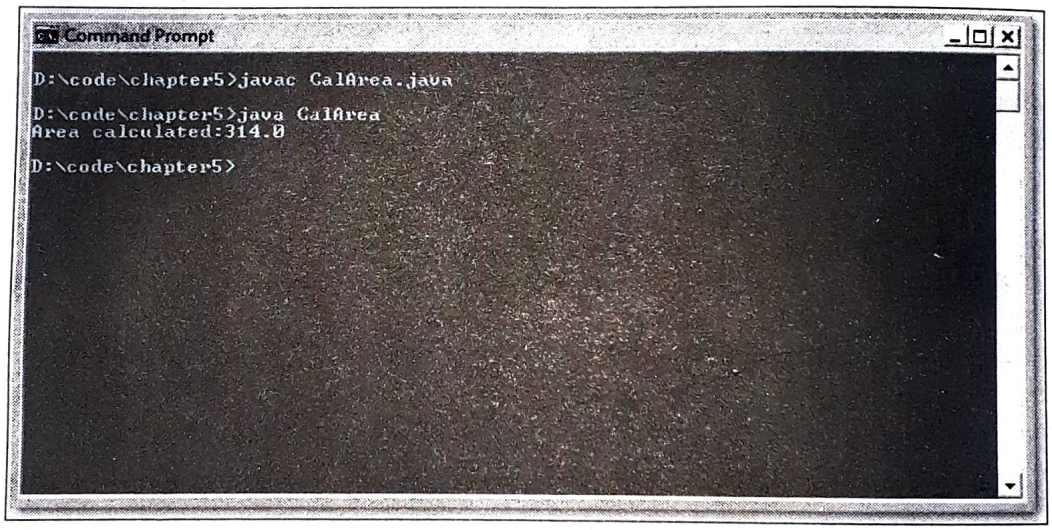
For more information about a package, please refer to Chapter 6, Multithreading and Packages in Java.

To understand this better, let's consider the `MyData` superclass, which contains a public member called radius, which is accessed from its subclass, `CalArea`. Listing 9 shows the radius public member in the `CalArea` class (you can find the CalArea.java file on the CD in the code\chapter5 folder):

▶ **Listing 9: The CalArea Class**

```
class MyData {
  public double radius ;
}
class CalArea extends MyData {
  public double calArea(double r) {
      return 3.14*r*r;
  }
  public static void main(String[] args) {
      CalArea num = new CalArea();
      num.radius = 10;
      double Result = num.calArea(num.radius);
      System.out.println("Area calculated:" +Result);
  }
}
```

In Listing 9, the `CalArea` subclass inherits the public members of the `MyData` superclass. The `radius` public member is initialized with a value 10 and passed as an argument of the `calArea()` method. The successful execution of Listing 9 indicates that you can access a public member of a superclass from a subclass. Figure 12 shows the output when the `CalArea` class is executed:



▲ **Figure 12: Executing the CalArea.java File**

In Figure 12, the output of the `CalArea` class is shown after accessing a public member of the `MyData` superclass.

TEST YOUR KNOWLEDGE

Q4. Write a program to inherit the public members in a subclass.

Ans.

```
class pub_mem {
  public String intro ;
}
class TestYourKnowledge4 extends pub_mem {
  public String getIntro(String str) {
      intro = str;
      return intro;
  }
  public static void main(String[] args) {
```

```
        TestYourKnowledge4 obj = new TestYourKnowledge4();
        String name = obj.getIntro("user");
        System.out.println("You're:" +name);
    }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge4.java
D:\code\chapter5>java TestYourKnowledge4
```

Output:

```
You're: user
```

Using the protected Access Modifier

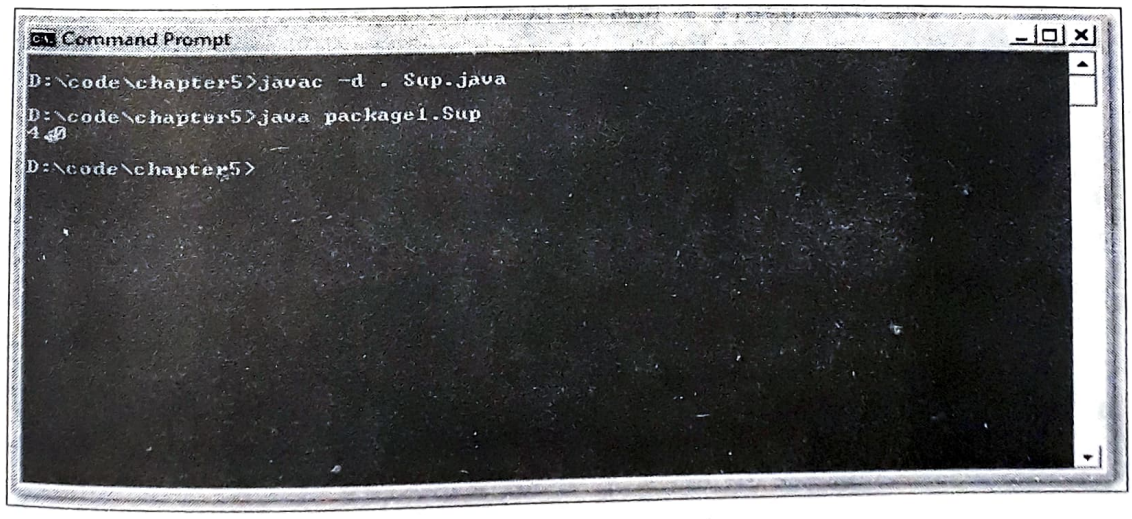
Protected members are those members that are accessible from all the classes of a particular package and only from the subclasses in another package. In other words, the non-subclasses of another package cannot access the protected members declared in a Java class.

Let's consider a class, `Sup1`, in `package1`, which contains a member, `radius`, with the value 4.0. Listing 10 shows the `Sup` class, which accesses the members of the `Sup1` class, and both these classes are in the same package (you can find the Sup.java file on the CD in the code\chapter5 folder):

▶ **Listing 10: The Sup Class**

```
package package1;
class Sup1 {
  double radius = 4.0 ;
}
public class Sup {
  public static void main(String s[]) {
      Sup1 ob = new Sup1();
      System.out.println(ob.radius);
  }
}
```


Executing the `Sup` class displays the output, as shown in Figure 13:



▲ **Figure 13: Executing the Sup.java File**

Figure 13 shows the output of the Sup class, where the members of the Sup1 class are accessed without inheriting the Sup class.

While discussing protected members, we first explained how to access the protected members of a class from another class declared within the same package. Now, let's discuss how to access the protected members of a superclass from a subclass within another package.

Consider a situation in which the MySup class is declared in package1, as shown in Listing 11 (you can find the MySup.java file on the CD in the code\chapter5 folder):

▶ **Listing 11: The MySup Class**

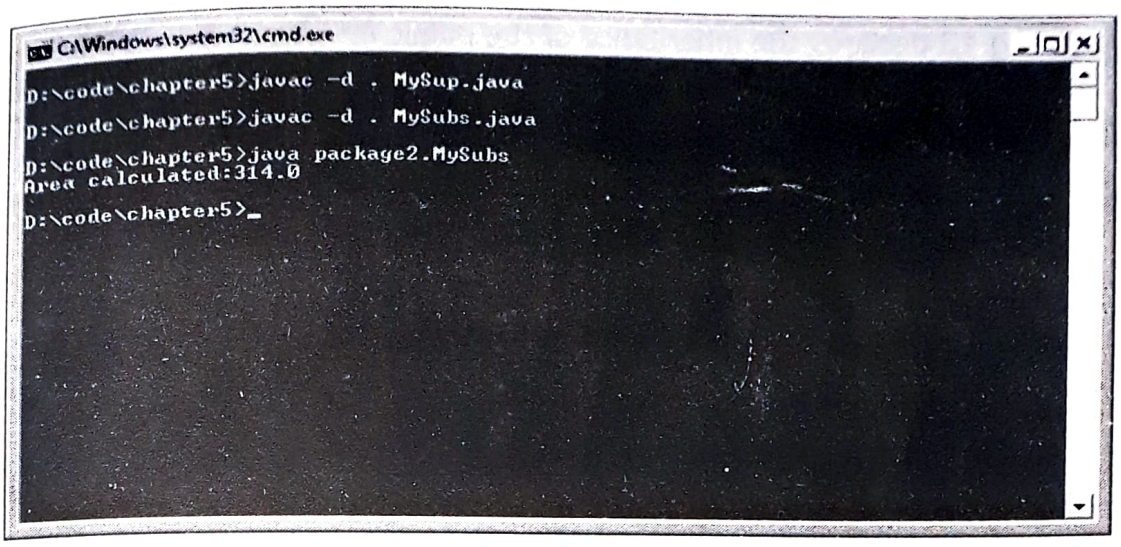
```
package package1;

public class MySup
{
  protected double radius ;
}
```

In Listing 11, the radius member is declared protected. Therefore, it can be accessed only from a subclass in another package, which is the MySubs class declared in package2, in our case. Listing 12 shows how to access a protected member from the MySubs class, declared in the package2 package (you can find the MySubs.java file on the CD in the code\chapter5 folder):

▶ **Listing 12: The MySubs Class**

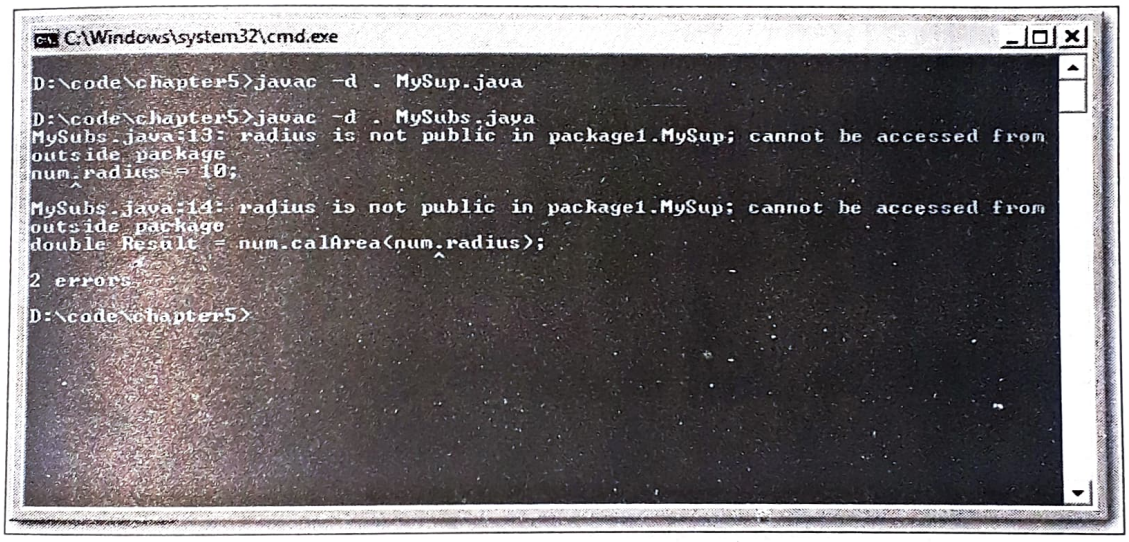
```
package package2;
import package1.*;

class MySubs extends MySup
{
  public double calArea(double r)
  {
      return 3.14*r*r;
  }

  public static void main(String[] args)
  {
      MySubs num = new MySubs();
      num.radius = 10;
      double Result = num.calArea(num.radius);
      System.out.println("Area calculated:" +Result);
  }
}
```

In Listing 12, the MySubs class is declared in the package2 package and this class is a subclass of the MySup class declared in package1. Here, an instance of the MySubs class is created and then a value 10 is assigned to the protected variable, radius. The calArea() method is then invoked by the subclass instance, and the value of the radius variable is passed as an argument. Finally, the result obtained from the calArea() method is displayed, as shown in Figure 14:



▲ Figure 14: Executing the MySubs.java File

Figure 14 displays the output on execution of the `MySup` class from the package1 package and the `MySubs` class from the package2 package. Alternatively, if the access modifier of the `MySup` class is changed to the `default` (`package`) accessibility, then the code does not compile but produces error messages, as shown in the Figure 15:



▲ Figure 15: Displaying an Error when a Member is Accessed Outside a Package

Figure 15 displays an error message as a result of accessing a default superclass member outside the package.

Using the private Access Modifier

The private access modifier is the most restrictive of all the access modifiers. Private members are not accessible outside their class and they cannot be accessed even by their subclasses, either from the current package or from another package. Therefore, private members cannot be inherited.

To understand this more clearly, let's first declare a public member, which can easily be accessed by a subclass and then change the member from public to private. Now, accessing a private member from a subclass will produce an error message.

Listing 13 declares the inheritance of the public member, `name` of the `MySuperPublic` class (you can find the MySubPublic.java file on the CD in the code\chapter5 folder):

▶ **Listing 13: The MySubPublic.java File**

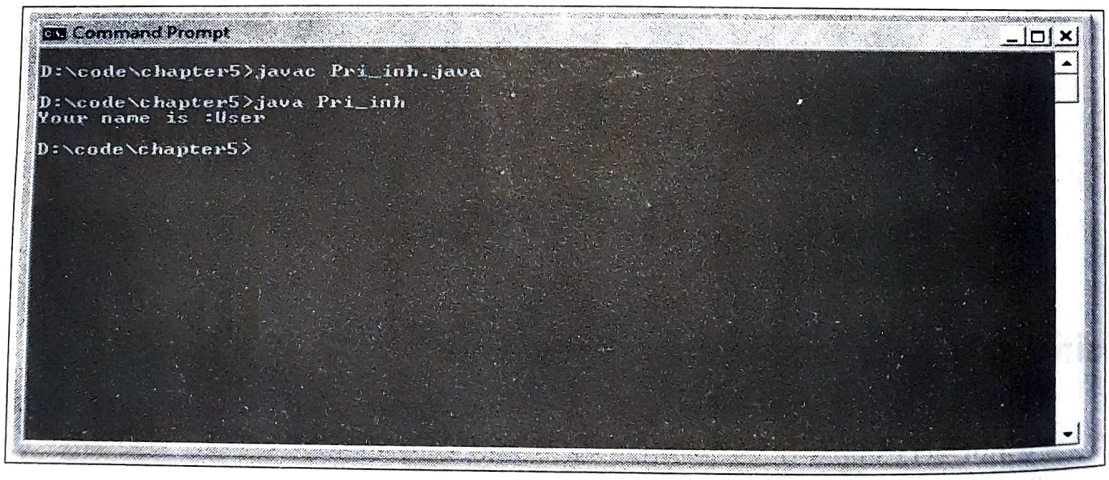
```
class MySuperPublic {
  public String name = "User";
}

class MySubPublic extends MySuperPublic
{
  public void displayName()
  {
      System.out.println ("Your name is:" + name);
  }
}
```

Now, let's create the `Pri_inh` class, which creates an instance of the `MySubPublic` class. Listing 14 declares the `Pri_inh` class (you can find the Pri_inh.java file on the CD in the code\chapter5 folder):

▶ **Listing 14: The Pri_inh.java Class**

```
public class Pri_inh
{
  public static void main(String[] args)
  {
      MySubPublic c = new MySubPublic();
      c.displayName();
  }
}
```

In Listing 14, the Pri_inh class creates an instance of the `MySubPublic` class to access the public member variable, `name`. Figure 16 shows the execution of the `Pri_inh` class:



▲ **Figure 16: Executing the Pri_inh.java File**

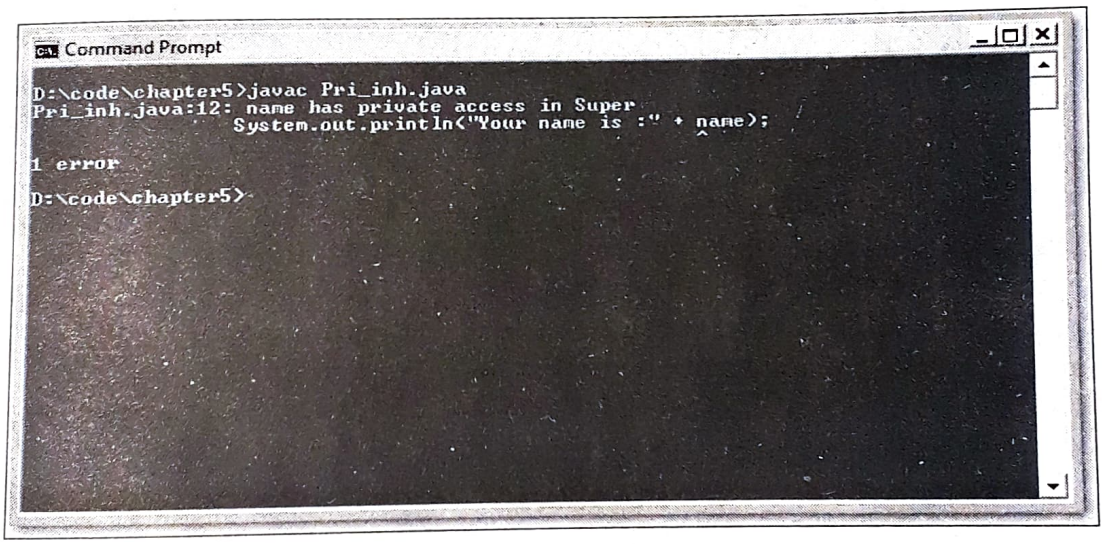
In Figure 16, the public members of the superclass are successfully accessed and display the message, "Your name is :User" .

If the modifier of the String `name` is changed from public to private, then the code also changes, as shown in Listing 15:

▶ **Listing 15: The MySubPublic Class**

```
class MySuperPublic {
  private String name = "User";
}
class MySubPublic extends MySuperPublic
{
  public void displayName()
  {
      System.out.println("Your name is :" + name);
  }
}
public class Pri_inh
{
  public static void main(String[] args)
  {
      MySubPublic c = new MySubPublic();
      c.displayName();
  }
}
```

In Listing 15, the access modifier of the `name` variable is declared `private`. Compilation of the `Pri_inch` class displays an error message, as shown in Figure 17:



▲ **Figure 17: Displaying an Error Message on Inheriting a Private Member**

Next, let's discuss how to refer to a subclass object from a superclass.

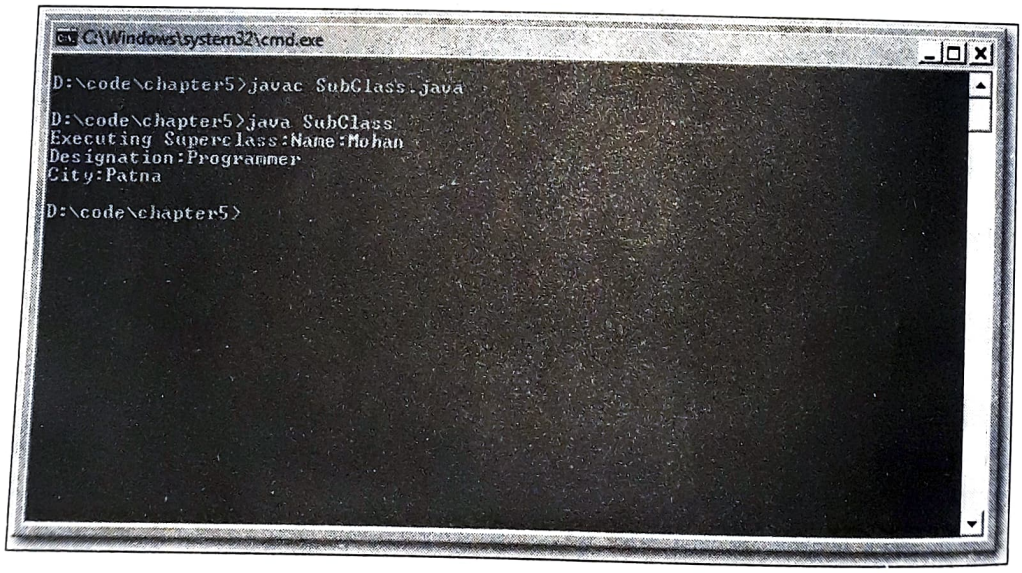
5.4 Referencing Subclass Objects

A reference variable of a superclass can access the members of the subclass. In Java, a reference variable does not contain an object; rather, it holds the object's reference. To understand this better, let's declare the `SuperClass` class, which contains two variables, `name` and `desg`, and a method, `show()`, which displays the values of the variables. The `SubClass` is the subclass of the `SuperClass` class, which contains the `address` field and the `show()` method. Listing 16 shows the declaration of the fields and methods in the `SuperClass` and `SubClass` classes (you can find the SubClass.java file on the CD in the code\chapter5 folder):

▶ Listing 16: The SubClass.java File

```
class SuperClass
{
 String name = "Mohan";
 String desg = "Programmer";
 public void show()
 {
     System.out.println("Name:" + name);
     System.out.println("Designation:" + desg);
 }
}
public class SubClass extends SuperClass
{
 String address = "Patna";
 public void show()
 {
     System.out.println("City:" + address);
 }
 public static void main(String s[])
 {
        SuperClass obj1 = new SuperClass();
        SubClass obj2 = new SubClass();
    System.out.print("Executing Superclass:");
    obj1.show();
    obj1 = obj2;
    obj1.show();
 }
}
```

In Listing 16, the objects of the `SuperClass` and `SubClass` classes are created inside the `main()` method. After the `show()` method of the `SuperClass` superclass is called, the reference of the `SuperClass` superclass is assigned to the reference of the `SubClass` subclass. Finally, the `show()` method of the subclass is called by the new reference object, `obj1`. Figure 18 shows the output on executing Listing 16:



▲ Figure 18: Executing the SubClass.java File

In Figure 18, the `obj2` subclass reference object is assigned to the `obj1` superclass reference and the method of the subclass is invoked by using the `obj1` object.

TEST YOUR KNOWLEDGE

Q5. Write a program to access the superclass reference in its subclass.

Ans.

```
class Super
{
  String name = "Gopi";
  String desg = "Cook";
  public void show()
  {
      System.out.println("Name:" + name);
      System.out.println("Designation:" + desg);
  }
}
public class TestYourKnowledge5 extends Super
{
  String address = "Delhi";
  public void show()
  {
      System.out.println("City:" + address);
  }
  public static void main(String s[])
  {
      Super ob1 = new Super();
      TestYourKnowledge5 ob2 = new TestYourKnowledge5();
      System.out.println("Executing Super:");
      ob1.show();
      ob1 = ob2;
      ob1.show();
  }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge5.java
D:\code\chapter5>java TestYourKnowledge5
```

Output:

```
Executing Super
Name: Gopi
Designation: Cook
City = Delhi
```

Now, we describe the use of super keyword in Java.

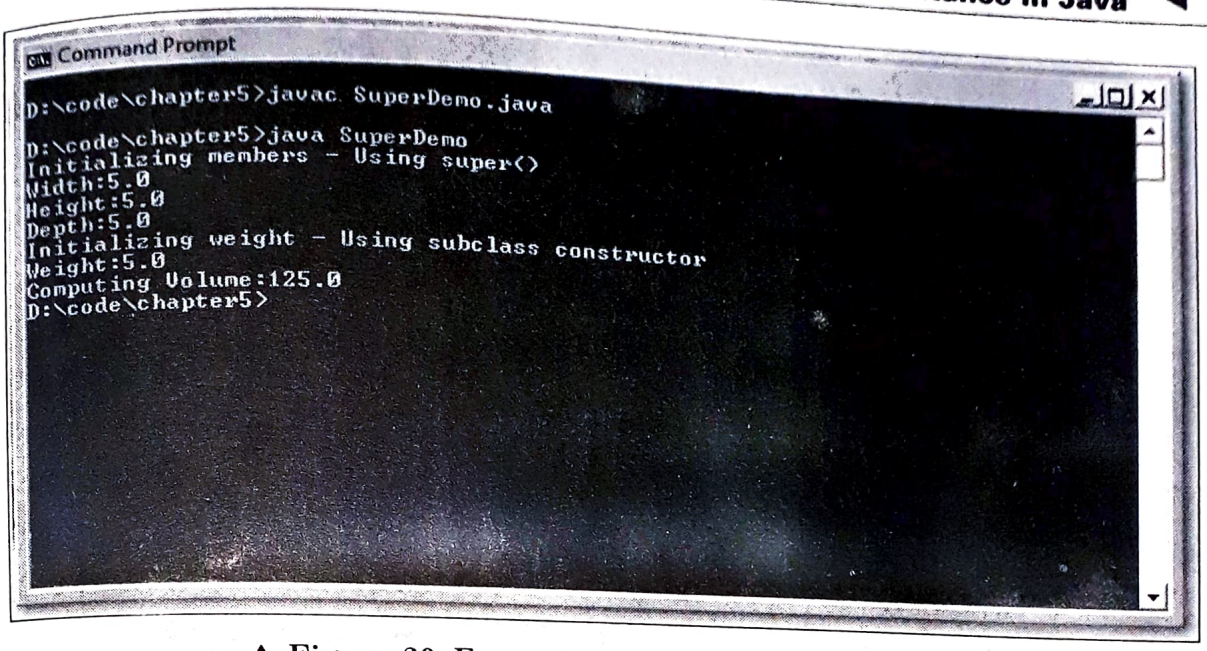
5.5 Using the super Keyword

The `super` keyword is used to access the members of a superclass in a subclass. Unlike the `this` keyword, the `super` keyword cannot be used to refer to the current object of a class but is instead used to invoke the overridden method and hidden variables in a subclass. In other words, the `super` keyword is used whenever you need to refer to a superclass from a subclass.

```
    double height;
    double depth;

    //constructor - I (superclass)

    Dim(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    Double volume()
    {
        return width * height * depth;
    }
}
class NewDim extends Dim
{
    double weight;
    //constructor-I subclass
    NewDim(double w, double h, double d, double m)
    {
        //Initializing superclass members from the subclass using the
        //super() call
        super(w,h,d);
        weight = m;
    }
}
public class SuperDemo
{
    public static void main(String s[])
    {
        System.out.println("Initializing members - Using super()");
        NewDim obj = new NewDim(5,5,5,5);
        System.out.println("Width:" + obj.width);
        System.out.println("Height:" + obj.height);
        System.out.println("Depth:" + obj.depth);
        System.out.println("Initializing weight - Using subclass constructor ");
        System.out.println("Weight:" + obj.weight);
        System.out.print("Computing Volume:");
        Double res = obj.volume();
        System.out.print(obj.volume());
    }
}
```

Listing 18 demonstrates the implementation of the `super()` constructor. The `super()` constructor is a superclass constructor used to call the members of a superclass from a subclass. The `super()` constructor is called in the `NewDim` class, which is a subclass of the `Dim` class and used to initialize the superclass members that are inherited in a subclass. As a result, while initializing the subclass object, the `NewDim(5,5,5,5)` subclass constructor is invoked, which also initializes the `superclass` and `subclass` members.

Figure 20 shows the output on executing Listing 18:



▲ **Figure 20: Executing the SuperDemo.java File**

Figure 20 displays the output on execution of the SuperDemo class. In Listing 18, the `super()` superclass constructor is called to initialize the members of the superclass.

The `final` keyword is used extensively in Java for various purposes, and particularly in inheritance. Therefore, let's discuss the `final` keyword in the next section.

5.7 Using the final Keyword

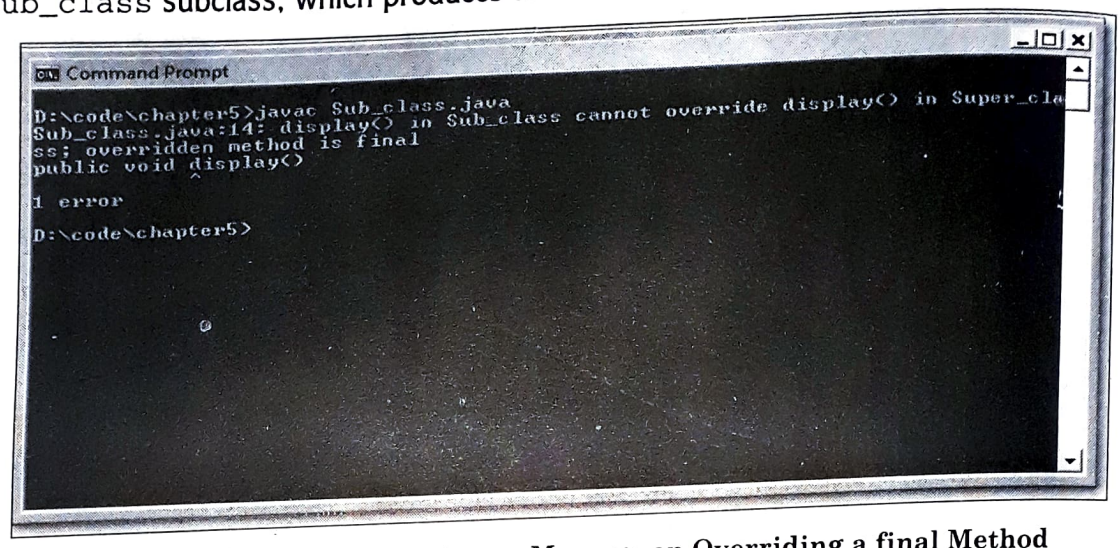
The `final` keyword is used for more than one purpose in Java programming. It can be used with classes and members. When the `final` keyword is used with a variable, the `final` variable is treated as a constant and you cannot modify its value. Similarly, if a reference variable is declared as final, then its reference cannot be changed. In addition, when it is used with a method, then the method cannot be overridden in a subclass. Moreover, using the `final` keyword with a class prevents the class from being inherited. In other words, you cannot create a subclass from the `final` class. In this section, we discuss the use of the final keyword with a class. Listing 19 shows how the final keyword prevents a method to be overridden in a subclass (you can find the Sub_class.java file on the CD in the code\chapter5 folder):

▶ **Listing 19: The Sub_class.java File**

```
class Super_class
{
  final void display()
  {
     System.out.println("Invoking superclass method");
  }
}
class Sub_class extends Super_class
{
  public void display()
  {
     System.out.println("Invoking subclass method");
  }
  public static void main(String s[])
```

```
    {
        System.out.println("Invoking members ");
        Sub_class obj = new Sub_class();
        obj.display();
    }
}
```

In Listing 19, the final `display()` method of the `Super_class` superclass is overridden in the `Sub_class` subclass, which produces an error message, as shown in Figure 21:



▲ **Figure 21: Displaying an Error Message on Overriding a final Method**

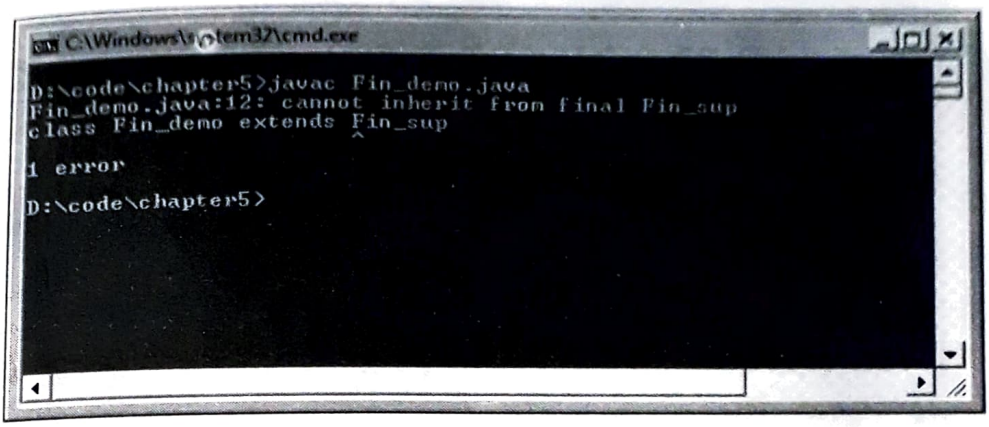
In Figure 21, an error message is displayed as the `Sub_class` subclass overrides the `final` method, that is, display(), of the Super_class superclass.

As stated earlier, the `final` keyword, when used with a class, prevents the class from creating a subclass. This feature is shown in Listing 20 (you can find the Fin_demo.java file on the CD in the code\chapter5 folder):

▶ **Listing 20: The Fin_demo.java File**

```
final class Fin_sup
{
 final void display()
 {
     System.out.println("Invoking superclass method");
 }
}
class Fin_demo extends Fin_sup
{
 public static void main(String s[])
 {
     System.out.println("Invoking members ");
     Fin_demo obj = new Fin_demo();
     obj.display();
 }
}
```

In Listing 20, the `Fin_sup` final superclass is extended, which produces an error message, as shown in Figure 22:



▲ **Figure 22: Displaying an Error Message on Inheriting a final Class**

In Figure 22, an error message is displayed when a final class is inherited or its subclass is created.

The concepts of the abstract and final classes are fairly distinct feature in Java. We have already discussed the final class in the previous section. Now, let's explore abstract classes in the next section.

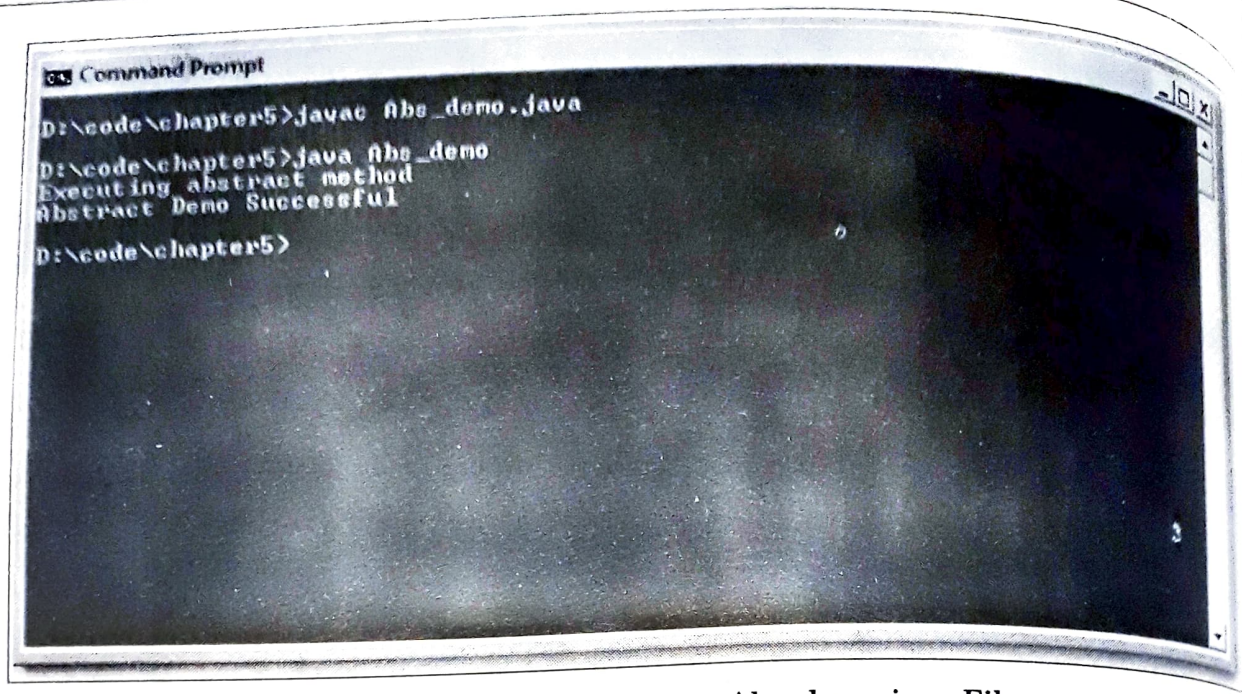
5.8 Declaring abstract Classes

The `abstract` keyword is used with a class to indicate that the class cannot be instantiated. In other words, its object cannot be created. If an `abstract` class contains `abstract` methods, then the implementation of the `abstract` methods is provided in its subclass. Here, let's create an abstract class, `Abs_ super`, which contains the `display()` `abstract` method, whose implementation is provided in its subclass, `Abs_demo`. Listing 21 shows the inheritance concept in an abstract class (you can find the Abs_demo.java file on the CD in the code\chapter5 folder):

▶ **Listing 21: The Abs_demo.java File**

```
abstract class Abs_super
{
 abstract public void display();
}
class Abs_demo extends Abs_super
{
 public void display()
 {
     System.out.println("Executing abstract method");
 }
 public static void main(String s[])
 {
     Abs_demo obj = new Abs_demo();
     obj.display();
     System.out.println("Abstract Demo Successful");
 }
}
```

Listing 21 shows the implementation of the `display()` abstract method of the `Abs_super` class in the `Abs_demo` subclass. Figure 23 shows the execution of the `Abs_demo` class:



▲ **Figure 23: Executing the Abs_demo.java File**

Figure 23 shows the output on implementing the `display()` abstract method.

TEST YOUR KNOWLEDGE

Q7. Write a program to implement an abstract method of an abstract class.

Ans.

```
abstract class abs
{
 abstract public void abs();
}

class TestYourKnowledge7 extends abs
{
 public void abs()
 {
     System.out.println("Abstract method implemented!");
 }
 public static void main(String s[])
 {
     TestYourKnowledge7 obj = new TestYourKnowledge7();
     obj.abs();
 }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge7.java
D:\code\chapter5>java TestYourKnowledge7
```

Output:

```
Abstract method implemented!
```

Q8. Write a program to compute the areas of a rectangle and square by using an abstract class.

Ans.

```
abstract class Cal
{
  int length;
  int bredth;
  abstract int area();
}
class Rectangle extends Cal
{
  int area()
  {
      return length * bredth;
  }
}
class Square extends Cal
{
  int area()
  {
      length = bredth;
      return length * bredth;
  }
}
class TestYourKnowledge8
{
  public static void main(String s[])
  {
      System.out.print("Calculating area of rectangle:");
      Rectangle ob1 = new Rectangle();
      ob1.length = 5;
      ob1.bredth = 7;
      System.out.println(ob1.area());
      System.out.print("Calculating area of square:");
      Square ob2 = new Square();
      ob2.length = 5;
      ob2.bredth = 7;
      System.out.println(ob2.area());
  }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge8.java
D:\code\chapter5>java TestYourKnowledge8
```

Output:

```
Calculating area of rectangle:35
Calculating area of square:49
```

Next, let's discuss the concept of interfaces, which is important for understanding how to extend multiple interfaces in Java.

5.9 Working with Interfaces in Java

An interface in Java is similar to an abstract class. Interfaces do not provide any method implementations and are therefore abstract by nature. As a result, you cannot create the subclass of an interface. An abstract class can define both abstract and non-abstract methods, but interfaces can have only abstract methods.

Java does not provide the facility of multiple inheritance (a subclass inheriting from more than one superclass). Instead, it provides an interface that permits multiple interface inheritance. Interfaces are declared by using the `interface` keyword, and contain constant and method declarations only. Therefore, it is not possible to create an object of an interface. The `implements` keyword is used to implement an interface in a class. The class that implements an interface is known as an implementation class. An implementation class is required to define all the methods of the interface that is implemented by the class.

In the next and subsequent sections, we define and implement an interface. We also discuss how to use reference variables of an interface, extend an interface from other interface and access its variables.

Defining Interfaces

Declaring an interface is similar to declaring a class, the only difference being that the class keyword is used while declaring a class, and the interface keyword is used while declaring an interface. The syntax of an interface is shown in the following code snippet:

```
<access modifier> interface <interface name> extends  <interface clauses>
{
//Interface body
<constant declarations>
<method prototype declarations>
.........
}
```

The rules for declaring an interface are as follows:

- There is no need to explicitly specify the public and abstract keywords while declaring a method in an interface. In other words, all interface methods are implicitly public and abstract.
- An interface declares only constants and not instance variables.
- The variables declared in an interface must be public, static, and final.
- Interface methods must not be `static`, final, or `native`.
- An interface can extend one or more interfaces.
- An interface cannot implement another interface or class.
- An interface must be declared by using the interface keyword.

As stated earlier, the definition of an interface includes two components, the interface declaration component and the interface body component. The interface declaration component consists of declaring various interface attributes, such as the name of the interface or whether or not it extends another interface. The interface body component consists of constants and method declarations in the interface. The following code snippet shows an interface definition:

```
//declaring an interface
public interface Predator
{
  //interface body
  // declaring abstract methods
  void catchPrey();
  void eatPrey();
}
```

As it is not essential to specify the public and abstract keywords while declaring the methods in an interface; therefore, methods in the preceding code snippet are declared without using these keywords.

Implementing Interfaces

A class can implement an interface by using the `implements` keyword. If a class implements more than one interface, then they must be separated by a comma. Interface methods are implicitly public so that they can be implemented outside the interface. A class can partially implement an interface if it is an abstract class. Listing 22 declares an interface, which is implemented by a class (you can find the IntImpl.java file on the CD in the code\chapter5 folder):

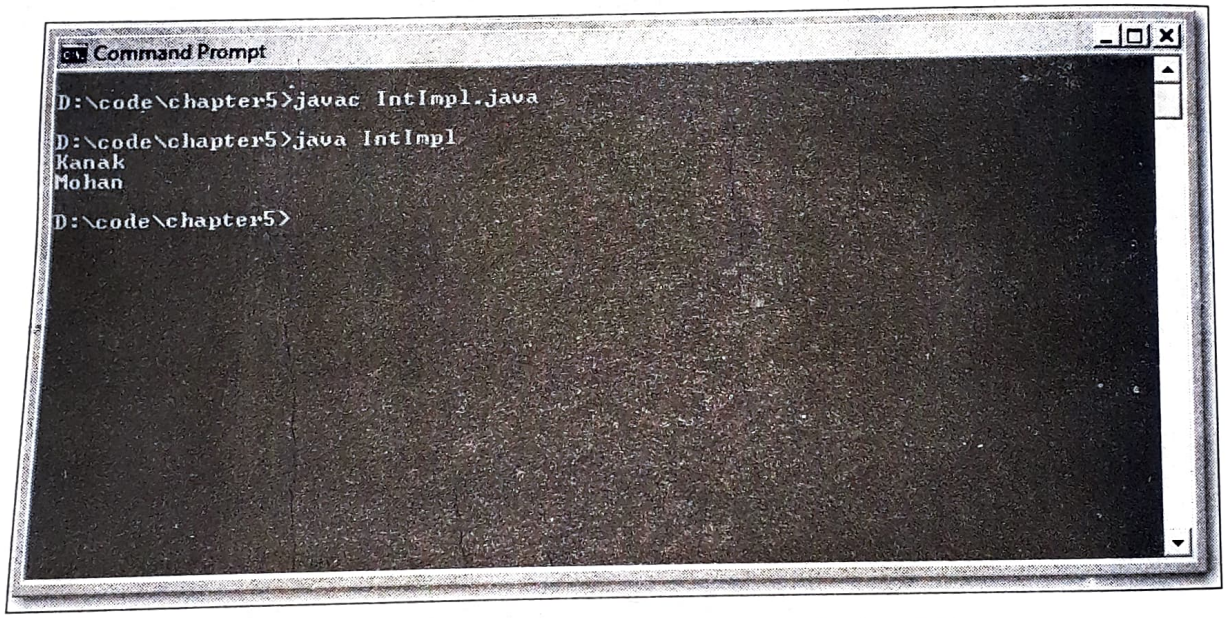
▶ **Listing 22: Implementing an Interface**

```
interface Demo
{
  public void push(Object item);
  Object pop();
}

public class IntImpl implements Demo
{
  Object [] stArray;
  int tos;
  IntImpl(int capacity)
  {
      stArray = new Object[capacity];
      tos = -1;
  }

  public void push(Object item)
  {
      stArray[++tos] = item;
  }

  public Object pop()
  {
      Object ob = stArray[tos];
      tos--;
      return ob;
  }

  public static void main(String s[])
  {
      IntImpl obj = new IntImpl(3);
      obj.push("Mohan");
```

```
        obj.push("Kanak");
        System.out.println(obj.pop());
        System. out.println(obj.pop());
    }
}
```

In Listing 22, we create the `Demo` interface, which contains the `push()` and `pop()` methods. Next, the `IntImpl` class is declared, which implements the `Demo` interface. This class declares the stArray array and the `tos` as well as capacity integer variables. A constructor of the `IntImpl` class is invoked with the value 3. This means that the `stArray` can contain 3 members of Object type.

In addition, the `tos` integer variable is assigned the -1 value so that it can be used to insert a value at the first position of the stack, when incremented. The `push()` method takes an argument of the `Object` type and the element associated with the argument is pushed inside the array at an incremented value of the `tos` variable. The `pop()` method first extracts an element based on the `tos` position and then decrements the value of the `tos` variable. Finally, the `pop()` method returns the extracted element. Figure 24 shows the execution of the `IntImpl` class:



▲ **Figure 24: Executing the IntImpl.java File**

Figure 24 displays the output after invoking the `push()` and `pop()` methods.

TEST YOUR KNOWLEDGE

Q9. Write a program to calculate the area by using an interface.

Ans.

```
interface  Cal
{
  public int area();
}
class Rectangle implements Cal
{
  public int length = 0;
  public int bredth  = 0;
  public int area()
  {
```

```
        return length * bredth;
    }
}
class Square implements Cal
{
  public int length = 0;
  public int bredth  = 0;
  public int area()
  {
      length = bredth;
          return length * bredth;
  }
}
public class TestYourKnowledge9
{
  public static void main(String s[])
  {
      System.out.print("Calculating area of rectangle:");
      Rectangle ob1 = new Rectangle();
ob1.length = 5;
      ob1.bredth = 7;
      System.out.println(ob1.area());
      System.out.print("Calculating area of square:");
      Square ob2 = new Square();
      ob2.length = 5;
      ob2.bredth = 7;
      System.out.println(ob2.area());
}
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge9.java
D:\code\chapter5>java TestYourKnowledge9
```

Output:

```
Calculating area of rectangle:35
Calculating area of square:49
```

Using Reference Variables of an Interface

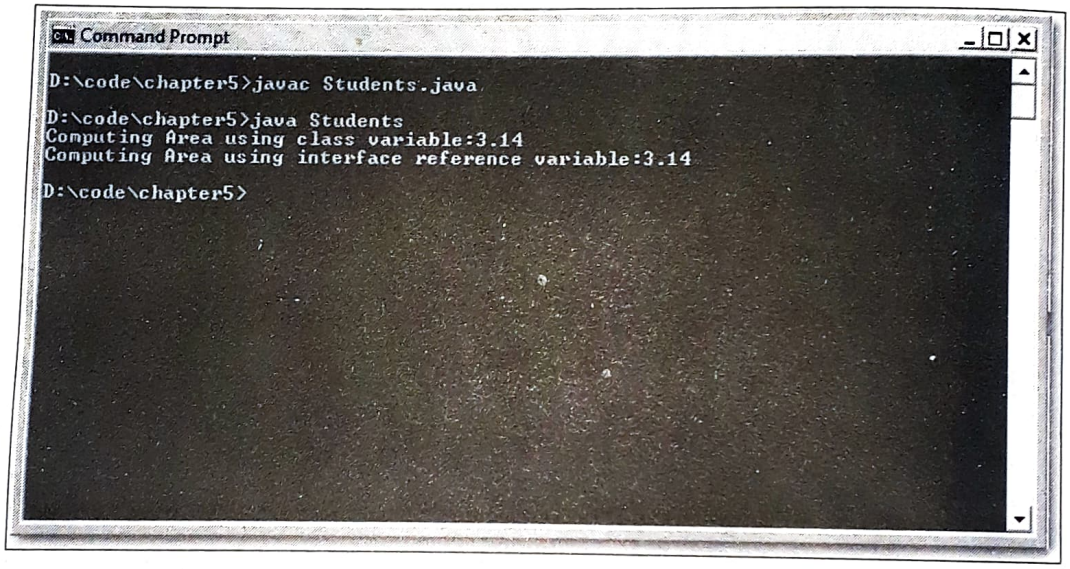
Similar to creating a class reference variable, you can also create an interface reference variable. In addition, by using the reference variable you can easily access the methods from the classes that implement the interface. Listing 23 shows how to use the variables declared in an interface (you can find the Students.java file on the CD in the code\chapter5 folder):

▶ **Listing 23: The Students.java File**

```
interface Cal
{
  double radius = 0;
  double Area(double radius);
}
public class Students implements Cal
{
```

```
    public double Area(double radius)
    {
        return 3.14*radius*radius;
    }
    public static void main(String s[])
    {
        Students obj = new Students();
        double res = obj.Area(1.0);
        System.out.print("Computing Area using class variable:");
        System. out.println(res);
        Cal intref;
        intref = obj;
        double res1 = obj.Area(1.0);
        System.out.print("Computing Area using interface reference
        variable:");
        System.out.println(res1);
    }
}
```

In Listing 23, the `Cal` interface is declared, which is implemented by the `Students` class. The methods of this class are then referenced separately by using the `obj` object and the `intref` interface variable. Figure 25 shows the execution of the `Students` class:



▲ Figure 25: Executing Students.java File

Figure 25 displays the output of the `Students` class, in which an interface variable is declared and initialized with the reference variable of the class implementing the interface. By using this interface variable, the methods of the interface are also invoked.

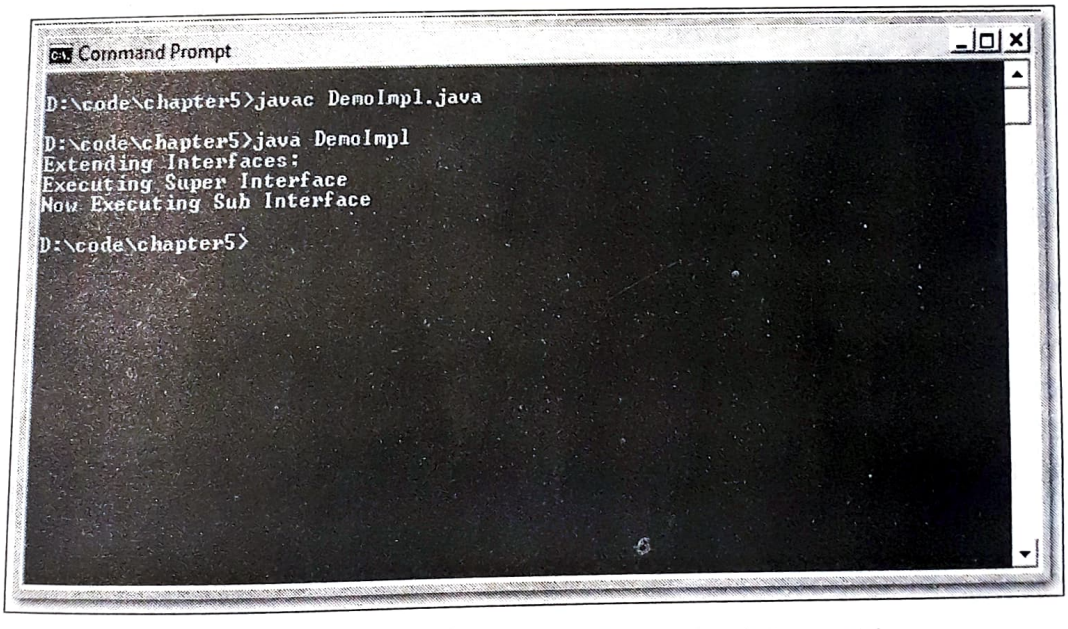
Extending Interfaces

An interface can extend another interface by using the `extends` keyword. Unlike the implementation of a class, which does not permit multiple inheritance, an interface can extend more than one interface. In the inheritance hierarchy, the interface formed by extending an interface is called a subinterface and the extended interfaces are called superinterfaces. Listing 24 shows how to extend more than one interface from an interface (you can find the DemoImpl.java file on the CD in the code\chapter5 folder):

▶ **Listing 24: The DemoImpl.java File**

```
interface Super
{
  public void mesgfromSuper();
}
interface Sub extends Super
{
  public void mesgfromSub();
}
class Demo implements Sub
{
  public void mesgfromSuper()
  {
      System.out.println("Executing Super Interface");
  }
  public void mesgfromSub()
  {
      System.out.println("Now Executing Sub Interface");
  }
}
public class DemoImpl
{
  public static void main(String s[])
  {
      Demo obj = new Demo();
      System.out.println("Extending Interfaces:");
      obj.mesgfromSuper();
      obj.mesgfromSub();
  }
}
```

In Listing 24, an interface is declared by extending a previously declared interface. Here, the class implementing the subinterface provides the implementation of both the superinterface and subinterface. Figure 26 shows the execution of the DemoImpl.java file:



▲ **Figure 26: Executing the DemoImpl.java File**

Figure 26 displays the output on executing the subinterface and superinterface in a class.

TEST YOUR KNOWLEDGE

Q10. Write a program for demonstrating interface extension.

Ans.

```
interface One
{
  public void mesg1();
}
interface Two extends One
{
  public void mesg2();
}
class TestYourKnowledge10 implements Two
{
  public void mesg1()
  {
      System.out.println("Executing SuperInterface");
  }
  public void mesg2()
  {
      System.out.println("Now Executing SubInterface");
  }
  public static void main(String s[])
  {
      TestYourKnowledge10 obj = new TestYourKnowledge10();
      System.out.println("Extending Interfaces:");
      obj.mesg1();
      obj.mesg2();
  }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge10.java
D:\code\chapter5>java TestYourKnowledge10
```

Output:

```
Extending Interfaces:
Executing SuperInterface
Now Executing SubInterface
```

■ Accessing Interface Variables

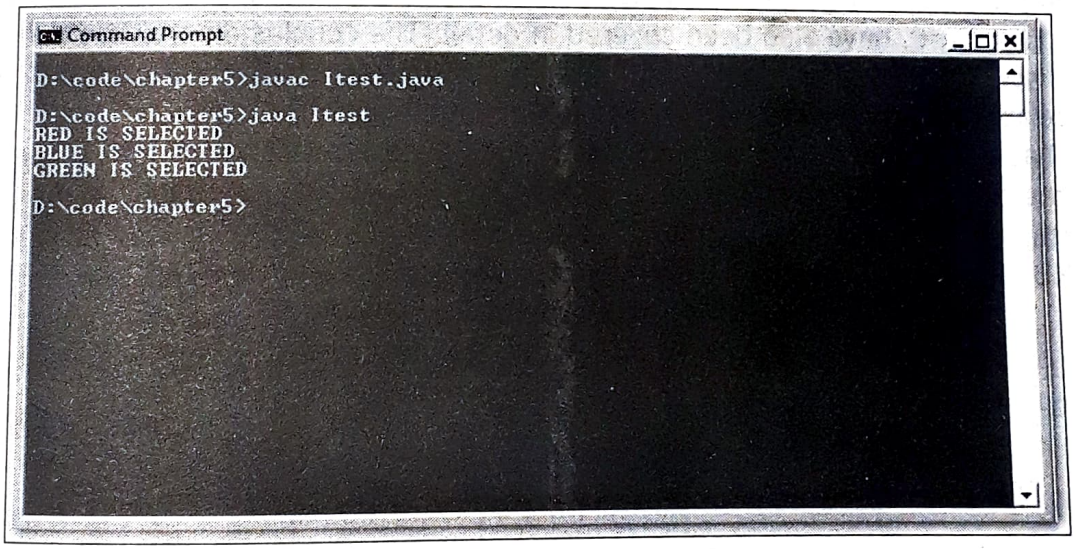
Similar to declaring a header file in other programming languages, Java also permits the declaration of a set of constants inside an interface, which can be accessed by any field or method similar to an ordinary variable. Listing 25 shows how to access interface variables within a Java class (you can find the Itest.java file on the CD in the code\chapter5 folder):

▶ **Listing 25: The Itest.java File**

```
interface Set
{
  int red = 1;
  int blue = 2;
```

```
    int green = 3;
}
class SetImpl implements Set {
    void select(int colour)
    {
        switch(colour) {
            case red:   System.out.println("RED IS SELECTED");
            break;
            case blue: System.out.println("BLUE IS SELECTED");
            break;
            case green: System.out.println("GREEN IS SELECTED");
            break;
        }
    }
}
public class Itest {
    public static void main(String s[]) {
        SetImpl ob = new SetImpl();
        ob.select(1);
        ob.select(2);
        ob.select(3);
    }
}
```

In Listing 25, an interface containing a set of integers is declared and later the class implementing this interface uses these integers one by one inside a switch statement. The value of the integer is passed to the method argument that is called inside the `main()` method. Figure 27 shows the execution of the Itest.java class:



▲ **Figure 27: Executing the Itest.java File**

Figure 27 displays the output on executing of the Itest.java class, which implements the interface variable.

TEST YOUR KNOWLEDGE

Q11. Write a program to implement interface variable implementations.

Ans.

```
interface Constants
{
    double PI = 3.14;
```

```
}
public class TestYourKnowledge11 implements Constants
{
public static void main(String s[])
{
      double radius = 10;
      System.out.println("Calculating Area: " + PI * radius * radius);
  }
}
```

Execution:

```
D:\code\chapter5>javac TestYourKnowledge11.java
D:\code\chapter5>java TestYourKnowledge11
```

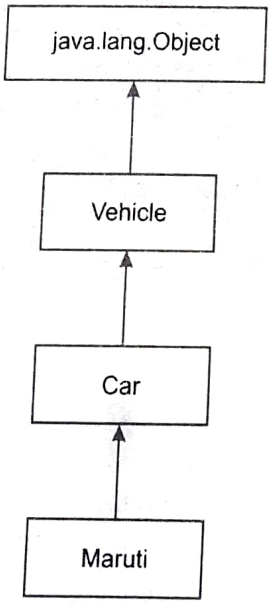
Output:

```
Calculating Area: 314.0
```

Now that you know something about interfaces, next we discuss class hierarchy in Java, which is nothing but an orderly arrangement of inherited classes.

5.10 Exploring Class Hierarchy in Java

We have already discussed the concept of inheritance in detail. The various inheritance implementations and the behavior of classes with access modifiers, such as `final` and `abstract`, have also been covered in detail. The conclusion that we can draw from the whole discussion is that Java only provides single or linear inheritance implementation. The relationship between a superclass and its subclass forms an inheritance hierarchy. The classes that are higher in this hierarchy are called generalized classes and those that are lower down the order are called specialized classes. The `java.lang.Object` class is always at the top of this inheritance hierarchy. All classes are directly or indirectly inherited from this class. Figure 28 shows the Java class hierarchy for a particular inheritance instance:



▲ Figure 28: Java Inheritance Hierarchy with the Object Class at the Top

Figure 28 is a representation of the inheritance hierarchy in Java. Note the `java.lang.Object` class, which is always at the top of such hierarchies.

With this, we come to the end of the theoretical part of the chapter. What follows is a brief summary of the chapter and a few questions related to the various topics discussed in the chapter.

Summary

In this chapter, we discussed inheritance, which is a powerful OOP feature. This chapter describes the various forms of inheritance in Java programming such as single, hierarchical, and multilevel. This chapter also discusses the inheritance of `public`, `protected`, and `private` modifiers. In addition, we learn how to refer a subclass variable, and use the `super` keyword and the `super ()` constructor call.

The chapter explains the use of the `final` and `abstract` keywords and discusses interfaces. It describes how to extend interfaces and access interface variables. It also explores class hierarchy in Java.

Review Questions

Now, let's solve the following types of questions related to this chapter:

- True or False
- Multiple Choice
- Short Answers
- Debugging

True or False

State True or False for the following:

1. Java permits multiple inheritance. **False**
2. The private members of a superclass can be inherited in a subclass. **False**
3. Protected members can be inherited by any classes outside a package. **False**
4. The public members of a superclass cannot be inherited in any class outside a package. **False**
5. The super keyword is used for accessing superclass members from a subclass. **True**
6. The super ( ) constructor call is used for initializing members of a superclass in a subclass constructor. **True**
7. A final class can be inherited. **False**
8. The final method of a superclass can be overridden in the subclass. **False**
9. Java provides multiple interface inheritance. **True**
10. An interface can contain a method body. **False**
11. An abstract class can be declared final. **False**
12. The java.lang.Object class is always at the top of an inheritance hierarchy. **True**

■ Multiple Choice Questions

Answer the following multiple choice questions:

Q1. Which of the following forms of inheritance are provided in Java?

A. Single
B. Multiple
C. Hierarchical
D. Multilevel

Ans. Options A, C and D are correct

Q2. Which of the following members can be inherited?

A. public
B. protected
C. Only A
D. Only B
E. Both A and B
F. None of the above

Ans. Option E is correct.

Q3. A private member:

A. Can be inherited
B. Cannot be inherited
C. Can't say

Ans. Option B is correct.

Q4. The final method:

A. Can be overridden
B. Cannot be overridden
C. Can be overridden but in its subclass
D. None of the above

Ans. Option B is correct.

Q5. The instance of an abstract class:

A. Can be created
B. Can be created but only in its subclass
C. Cannot be created
D. None of the above

Ans. Option C is correct.

Q6. Consider the following program:

```
class S
{
 public void display()
     {
     System.out.println ("Hello from Super class");
     }
}
class Sb extends S
{
 public static void main(String s[])
 {
     Sb obj = new Sb();
     obj.display();
 }
}
```

What is the result when the preceding program is compiled and executed?

A. The program will not compile
B. The program will compile but not execute
C. The program will execute successfully and print "Hello from Super class"
D. None of the above

Ans. Option C is correct.

Q7. Consider the following program:

```
package package1;
class Super
{
    private int value = 4;
}
class Sub extends Super
{
    public static void main(String s[])
    {
        Sub obj = new Sub();
        System.out.println ("Accessing value from superclass:" +
        obj.value);
    }
}
```

What is the result when the preceding program is compiled and executed?

A. The program will not compile
B. The program will compile but not execute
C. The program will execute successfully and print the value 4
D. None of the above

Ans. Option A is correct.

Q8. The String class:

A. Can be extended
B. Cannot be extended.
C. Can't say

Ans. Option B is correct.

Q9. The following declaration:

```
abstract class AbsDemo {
 abstract void display() {System.out.println("The class is an abstract
    class");
}
```

A. Is correct
B. Is not correct
C. Can't say

Ans. Option B is correct.

Q10. The following declaration:

```
abstract class AbsDemo {
    abstract void display();
}
```

A. Is correct

B. Is not correct

C. Can't say

Ans. Option A is correct.

Q11. Consider the following program:

```
interface test {
    void test();
}
class Impltest implements test {
    public void test() {
        System.out.println("Displaying a method from interface");
    }
    public static void main(String s[])  {
        Impltest ts = new Impltest( );
        ts.test();
    }
}
```

What is the result when the preceding code is compiled and executed?

A. The program will not compile

B. The program will compile but not execute

C. The program will execute successfully and print "Displaying a method from interface"

D. None of the above

Ans. Option C is correct.

Q12. The class at the top of the inheritance hierarchy is:

A. java.lang.Object

B. java.util.List

C. java.util.Vector

D. None of the above

Ans. Option A is correct.

Short Answer Questions

Answer the following short answer questions.

Q1. Why Java does not support multiple inheritance?

Ans. Java does not support multiple inheritance to eliminate code complexity and confusion. Instead, it provides an interface that permits multiple interface implementations. As a result, the interface can extend more than one interface at a time. The class that provides the implementation of the interface also implements the methods of the inherited interfaces.

Q2. What is the difference between an interface and an abstract class?

Ans. An interface only defines the method prototype and not the method body; therefore, an interface is abstract by nature. An abstract class can provide implementations for some of its method which an interface cannot do.

Q3. Why is a final class called a complete class?

Ans. When a class is declared final, it cannot be inherited. As a result, its subclass cannot be declared. The final class does not need any other class to implement its methods. In addition, the methods of a final class cannot be overridden. Therefore, the final class is called a complete class.

Q4. What is the difference between importing and extending a class?

Ans. When we extend a class from its superclass, an inheritance relationship develops between the two. However, while importing a class, such a relationship does not exist. In addition, the inherited members are visible only in the subclass. In contrast, while importing a class, its members are visible in any class.

Q5. How can one prevent a class from being inherited?

Ans. A class can be declared final to prevent it from being inherited. The syntax for declaring a final class is:

```
final class NotInherited {
// members
}
```

Q6. What is overriding?

Ans. Overriding is a process of declaring a method in a subclass with the same signature as that declared in its superclass. The subclass provides its own implementation of the method. The following example represents the method of overriding:

```
class Super {
     public void methodofsuperclass()  {
          ...
     }
}
class Subclass extends Super {
     public void methodof superclass() {
          ....
     }
}
```

Q7. How can one prevent a method from being overridden?

Ans. To prevent a specific method from being overridden in a subclass, the final modifier is used with method declaration. The following example represents the final Notoverriden method:

```
public final void Notoverriden() {
 // Method body
}
```

Q8. What is a marker interface?

Ans. A marker interface is an in empty interface without any body. These are also called tag or ability interfaces. Some examples of marker interfaces are java.lang.Cloneable, java.io.Serializable, and java.util.EventListner.

Q9. What are the differences between overriding and overloading?

Ans. Method overriding is the declaration of a superclass method with the same signature (name and parameters) in a subclass. Only the non-final methods of a supeclass are overridden in a subclass. Method overloading occurs when a method name is the same but the parameters are different. A method can be overloaded in a class and a subclass.

Q10. What is the role of the protected access modifier?

Ans. The protected access modifier is used to ensure that the members of a class are accessed by the classes in the current package and only by the subclass in other package.

Q11. What are the differences between the this() and super() constructor calls?

Ans. The this() constructor is used to invoke a constructor of the same class, whereas the super() constructor call can be used to invoke a superclass constructor. The super() constructor call is used to initialize the members of a superclass from a subclass. In addition, the super() constructor call is used to reference the object of the superclass whereas the this() constructor call is used to reference the current object.

Q12. What are constants in interfaces?

Ans. An interface can contain constants, which are defined by field definitions in the form of public, static or final members. They are initialized during interface declaration and accessed by the classes implementing the interface.

Debugging Exercises

Q1. What will happen when you compile and execute the following program?

```
class MySuper {
 public MySuper(String str)    {
     System.out.println("One");
 }
}
public class MySub extends MySuper {
 public MySub(String str)  {
     System.out.println("Two");
 }
 public static void main(String args[]) {
     new MySub("Bye");
 }
}
```

Ans. The program will lead to a compilation error because no superclass constructor is declared here.

Q2. **What will happen when you compile and execute the following program?**

```
class MySuper {
 public final void MyMethod()  {
     System.out.println("Hello");
 }
}
public class MySub extends MySuper {
 public final void MyMethod()  {
     System.out.println("Hi");
 }
 public static void main(String args[]) {
     new MySub().MyMethod();
 }
}
```

Ans. The program will lead to a compilation error because the final method is overridden.

Q3. **What is the result if the following code snippet is compiled and executed?**

```
class P{}
class Q extends P{}
```

Ans. The code snippet will compile successfully but not execute because the main() method is not present.

Q4. **What is the result if the following program is compiled and executed?**

```
class MySuper
{
 public static void Method(MySuper a)
 {
 System.out.println("Method in super class");
 }
}
class MySub extends MySuper
{
 public static void SubMethod(MySuper a)
 {
     if(a instanceof MySub) {
         System.out.println("object of MySub");
     } else if(a instanceof MySuper)
     {
         System.out.println("object of MySuper");
     }
     else {
         System.out.println("object not of MySub or MySuper");
     }
 }
 public static void main(String arg[]){
     MySuper a = new MySub();
     SubMethod(a);
 }
}
```

Ans. The program compiles successfully and displays "object of MySub" as the output.

Q5. What is the result when the following program is compiled and executed?

```
class MySuper {
 public static void MyMethod()
 {
 System.out.println("Static method in super class");
 }
}
public class MySub extends MySuper {
 public static void MyMethod() {
     System.out.println("Static method in subclass");
 }
 public void MyMethod() {
     System.out.println("Non-static method in subclass");
 }
 public static void main(String args[]) {
     MySuper s = new MySub();
     MyMethod();
 }
}
```

Ans. The program will not compile successfully because a static method cannot be overloaded.

Q6. What is the result when the following program is compiled and executed?

```
class A {
 A()
 {
     System.out.println ("A");
 }
}
public class B extends A {
 public static void main(String args[]) {
     B b1 = new B();
     super();
 }
 B() {
     System.out.println("B");
 }
}
```

Ans. The program will not compile successfully because the super () call must be the first statement in the subclass constructor.

Q7. What will be the output after overloading the constructor of the Test class, created in the following program?

```
class Test {
 Test() {
     System.out.println("Default constructor in Test class");
 }
 Test(float f) {
 System.out.println("Parameterized constructor in Test class");
 }
}
public class MyTest extends Test {
 public static void main(String args[]) {
```

```
        new MyTest();
    }
}
```

Ans. The program compiles successfully and displays "Default constructor in Test class" as an output.

Q8. What is the output when the following program is compiled and executed?

```
class test {
 super(){System.out.println("Executing super");}
 public static void main(String args[])
 {
     test ob = new test();
        ob.super();
 }
}
```

NCS859

Ans. The program will not compile successfully because the super () call has not been used correctly.

Q9. What is the result when the following program is compiled and executed?

```
final class MyBase {
 public void MyMethod() {
     System.out.println("Final class");
 }
}
public class MySub extends MyBase {
 public void MyMethod() {
 System.out.println("Subclass extends the Final Base class");
}
public static void main(String arg[]) {
 MyBase obj = new MyBase();
 obj.MyMethod();
 }
}
```

Ans. The program will not compile successfully because the final class cannot be inherited.

Q10. What is the result when the following program is compiled and executed?

```
class MySuper {
 MySuper () { disp();  }
 void disp() { System.out.println("SuperClass"); }
}
class MySub extends MySuper {
 double i =  Math.ceil(8.4f);
 public static void main(String[] args) {
     MySuper obj = new MySub();
     obj.disp();
 }
 void disp() { System.out.println(i); }
}
```

Ans. The program compiles successfully and displays 0.0, followed by 9.0 as an output.

Q11. What result will you get on compiling and executing the following program?

```
class Super {
    public void display()  {
        System.out.println("SUBCLASS");
    }
}
class Sub extends Super {
    public static void main(String s[]) {
        Sub.display();
    }
}
```

Ans. The program will not compile successfully because a non-static method of superclass is accessed in a static method or class.

Q12. What will happen if you compile and execute the following program?

```
class Super {
    public static void display()  {
        System.out.println("SUBCLASS");
    }
}
class Sub extends Super {
    public static void main(String s[])  {
        Sub obj = new Sub();
        obj.display();
    }
}
```

Ans. The program compiles successfully and displays "SUBCLASS" as an output.