# Chapter 4
# Inheritance, Interface and Package

## 1. Inheritance

Inheritance is the fundamental concept of object-oriented programming languages. Inheritance is the mechanism of deriving a new class from existing class. The existing class is known as the **superclass** or **base class** or **parent class** and the new class is called **subclass** or **derived class** or **child class** or **extended class**. The subclass inherits some of the members (fields, methods, and nested classes) from the superclass and can add its own members as well. A subclass does not inherit `private` members of its parent class. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass by using **super** keyword.

Inheritance uses the concept of code **reusability**. Once a super class is written and debugged, we can reuse the members in this class in other classes by using the concept of inheritance. Reusing existing code saves time and money and increases program's reliability.

An important result of reusability is the ease of distributing classes. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

**Note:** In Java, every class is implicitly a subclass of `Object` class. A superclass reference variable can refer to a subclass object.

### 1.1. Defining Subclasses

The keyword **extends** indicates that you are making a new class that derives from an existing class. For example,

```
public class Square
{
        protected int side;
        public int areaSquare()
        {
                return side * side;
        }
}
public class Cube extends Square
{
        public int areaCube()
        {
                return areaSquare() * side;
        }
}
public class MainProgram
{
        public static void main(String[]args)
        {
```

```
                Cube c1 = new Cube();
                c1.side = 5;
                System.out.println("Area = " + c1.areaCube());
        }
}
```

## 1.2. Using super Keyword

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. We can use **super** keyword in the following two ways:

- Using super to call superclass constructor
- Using super to access a member of the superclass that has been hidden by a member of a subclass

**Using super to call superclass constructor:**

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

```
super(parameter-list);
```

Here, parameter-list specifies any parameters needed by the constructor in the superclass. **super** always refers to the superclass immediately above the calling class. This is true even in multilevel inheritance [for example, **super.super…super(parameter-list)**]. Also, **super** must always be the first statement executed inside a subclass constructor. For example,

```
public class Square
{
        protected int side;
        public Square(int s)
        {
                side = s;
        }
        public int areaSquare()
        {
                return side * side;
        }
}
public class Cube extends Square
{
        public Cube(int s)
        {
                super(s); // calling superclass constructor
        }
        public int areaCube()
        {
                return areaSquare() * side;
        }
}
```

**Using super to access a member of the superclass**:
This form is applicable if member names of a subclass hide members by the same name in the superclass. The usage in this case has the following general form: **super.member** (this is true even in multilevel inheritance, for example, **super.super…super.member**).
For example,

```
public class Square
{
        protected int side;
        public int area()
        {
                return side * side;
        }
}
public class Cube extends Square
{
        public int area()
        {
                return super.area() * side;
        }
}
```

## 1.3. Method Overriding

The process of redefining the inherited method of the super class in the derived class is called method **overriding**. The signature and the return type of the method must be identical in both the super class and the subclass. The overridden method in the subclass should have its access modifier same or less restrictive than that of super class. For example, if the overridden method in super class is protected, then it must be either **protected** or **public** in the subclass but not **private** or **no-modifier**. The example below shows method overriding.

```
public class Square
{
        protected int side;
        public int area()
        {
                return side * side;
        }
}
public class Cube extends Square
{
        public int area()
        {
                return super.area() * side;
        }
}
```

**Note:** Like method overloading, method overriding is also an example of polymorphism.

## 1.4. 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. This is the example of *run-time polymorphism.*

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. 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:

```
public class Square
{
	protected int side;
	public Square(int s)
	{
		side = s;
	}
	public int area()
	{
		return side * side;
	}
}
public class Cube extends Square
{
	public Cube(int s)
	{
		super(s);
	}
	public int area()
	{
		return super.area() * side;
	}
}
public class MainProgram
{
	public static void main(String[]args)
	{
		Square s1 = new Square(3);
		Cube c1 = new Cube(5);
		Square s = s1;
		System.out.println(s.area());
		s = c1;
		System.out.println(s.area());
	}
}
```

## 1.5. Using final with Inheritance

We can also use the **final** keyword in inheritance in the following two ways.

- Using final to prevent overriding
- Using final to prevent inheritance

**Using final to prevent overriding:**

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. For example,

```
public class A
{
        public final void meth()
        {
                System.out.println("This is the final method");
        }
}
public class B extends A
{
        public void meth() // illegal
        {
                System.out.println("This is the method");
        }
}
```

**Using final to prevent inheritance:**

If we want to prevent a class from being inherited, we precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. For example,

```
public final class A
{
        // code
}
public class B extends A //illegal
{
        // code
}
```

## 1.6. Abstract Classes and Methods

If we declare a class as abstract, then it is necessary for you to subclass it so as to instantiate the object of that class i.e. we cannot instantiate object of abstract class without creating its subclass. The keyword **abstract** is used to create a class as an abstract class and it can contain abstract methods. Likewise, if you need your method to be always overridden before it can be used, you can declare the method as an abstract method using **abstract** keyword and without the method definition i.e. end it with semicolon. Remember, if you declare some method as an abstract method, you must declare your class as abstract. See the example skeleton below:

```
public abstract class MyClass
```

```
{
        public abstract void myMethod1();  // no definition
        public void myMethod2()
        {
                //code
        }
}
```

Here we cannot instantiate object of MyClass, if we need to instantiate, we must inherit class MyClass to some other class such that it overrides the abstract methods in class MyClass. If the derived class does not define all the abstract methods of the super class then the derived class again should be abstract.

## 1.7. The Object Class

The Object class is the ultimate ancestor – every class in Java extends Object. However, you never have to write **extends Object**. The ultimate superclass Object is taken for granted if no superclass is explicitly mentioned. Since every class in Java extends Object, it is important to be familiar with the services provided by the Object class. It has single default constructor **public Object()** and many methods like **clone()**, **toString()**, **finalize()**, **notify()**, **wait()**, **getClass()**, **hashCode()** etc.

# 2. Types of Inheritance

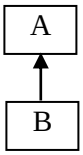
A class can inherit members from one or more classes and from one or more levels. On the basis of this concept, inheritance may take following different forms:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance

## 2.1. Single Inheritance

In single inheritance, a class is derived from only one existing class. The general form and figure are given below:

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

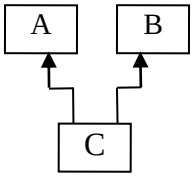



## 2.2. Multiple Inheritance

In this type, a class derives from more than one existing classes. **Java does not support multiple inheritance**. That is, a class cannot have more than one immediate superclass. However, Java provides an alternate approach known as **interfaces** to support the concept

of multiple inheritance. We will discuss this type of inheritance when we discuss interfaces.

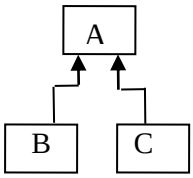
```
public class A
{
        // members of A
}
public class B
{
        // members of B
}
public class C extends A, B // not allowed in Java
{
        // own members of C
}
```




## 2.3. Hierarchical Inheritance

In this type, two or more classes inherit the properties of one existing class. The general form and figure are given below:

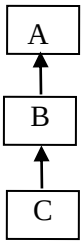
```
public class A
{
        // members of A
}
public class B extends A
{
        // own members of B
}
public class C extends A
{
        // own members of C
}
```




## 2.4. Multilevel Inheritance

The mechanism of deriving a class from another subclass is known as multilevel inheritance. The process can be extended to an arbitrary number of levels. For example,

```
public class A
{
        // members of A
}
public class B extends A
{
        // own members of B
}
public class C extends B
{
        // own members of C
}
```



# 3. Interfaces

In the Java programming, an *interface* is a reference type that can contain *only* constants and method declarations. There are no method bodies. Interfaces cannot be instantiated – they can only be *implemented* by classes or *extended* by other interfaces.
To use an interface, we write a class that *implements* the interface. When an instantiable class implements an interface, it should either provide method body for each of the methods declared in the interface or the class should be declared abstract.

## 3.1. Declaring Interface

The syntax for defining an interface is very similar to that for defining a class. To define an interface, we use the keyword **interface** instead of **class**. The general form of an interface definition is:

```
[access] interface InterfaceName [extends Interface1[, Interface2 [, …]]
{
        constant declarations;
        method declarations;
}
```

**Components:**

- **access**: It is either **public** or not used. When it is declared as public, the interface is available to any other code. When it is declared with no access specifier, the interface is only available to other members of the package in which it is declared.
- **InterfaceName**: It is the name of the interface, and can be any valid identifier.
- **constant declarations**: These are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.
- **method declarations**: Method declarations will contain only a list of methods without body statements. They end with semicolon after the parameter list. These methods are implicitly **abstract**.

**Example:**

```
public interface Area
{
        float PI = 3.1415f;
        float compute(float x, float y);
}
```

**Remember:** All the members in an interface are public implicitly so we can omit the public keyword there.

## 3.2. Extending Interfaces

Like classes, interfaces can also be extended. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. We can also extend several interfaces into a single interface. When an interface extends two or more interfaces they are separated by commas as given below.

```
public interface ItemConstants
{
        int code = 1001;
        String name = "Fan";
}
public interface ItemMethods
{
        void display();
}
public interface Item extends ItemConstants, ItemMethods
{
        ……
        ……
}
```

## 3.3. Implementing Interfaces

Interfaces are used as superclass whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. We use **implements** keyword to do this. Remember that a class can implement any number of interfaces. The general form is:

```
[access][modifier]class ClassName implements Interface1 [, Interface2[, …]]
{
        //definition of the class
        //all the methods in the interfaces must also be defined
}
```

**Example:**

```
public interface Area
{
        float PI = 3.1415f;
        float findArea(float x);
}
public class Circle implements Area
{
        public float findArea(float x)
        {
                return PI * x * x;
        }
}
public class InterfaceTest
{
        public static void main(String[]args)
        {
```

```
                Circle c1 = new Circle();
                float area = c1.findArea(3.2f);
                System.out.println("Area = " + area);
        }
}
```

**Note:** If a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface. In the above example we can write,

Area c1 = new Circle();

## 3.4. Using Interfaces to Achieve Multiple Inheritance

Interfaces can be used for multiple inheritance. In Java a class can extend only one class and if the situation comes where we need to gather the properties of two kinds of objects then Java cannot help us doing so by using classes. In this kind of situation, interface can be used to achieve multiple inheritance. Note that we can extend only one class but can implement any number of interfaces. The method signature in the class must match the method signature of the interface. For example,

```
public class Test
{
        protected int rollNumber;
        protected float part1, part2;
        public Test(int r, float p1, float p2)
        {
                rollNumber = r;
                part1 = p1;
                part2 = p2;
        }
        public void showDetail()
        {
                System.out.println("Roll Number: " + rollNumber);
                System.out.println("Marks Obtained");
                System.out.println("Part1 = " + part1);
                System.out.println("Part2 = " + part2);
        }
}

public interface Sports
{
        float sportWt = 6.0f;
        void showSportWt();
}

public class Result extends Test implements Sports
{
        private float total;
        public Result(int r, float p1, float p2)
        {
```

```
            super(r, p1, p2);
        }
        public void showSportWt()
        {
            System.out.println("Sports Wt = " + sportWt);
        }
        public void showTotal()
        {
            total = part1 + part2 + sportWt;
            System.out.println("Total score = " + total);
        }
}
public class MultipleInheritance
{
        public static void main(String[]args)
        {
            Result s1 = new Result(12, 27.5f, 56.0f);
            s1.showDetail();
            s1.showSportWt();
            s1.showTotal();
        }
}
```

# 4. Packages

A package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management. For example, **java.io** is a package in Java contains related types for providing input output facilities. Java has hundreds of pre-defined packages in its library. A package name must match directory name where the files are stored. For example, **java.lang** package must have its class files in the directory java/lang.

## 4.1. Creating Packages

We can also define our own packages to bundle group of classes, interfaces etc. To create a package, name the package using standard naming convention and write a package statement (there is only one package statement per file) with the package name at the top of the Java source file that is to be bundled in the package. For example, if we want to bundlee a Java source file **Student.java** in the package **tu.pmc.bca** then the first line in the file must have **package tu.pmc.bca;** as given below.

```
package tu.pmc.bca;
public class Student
{
        //class members
}
```

The class file **Students.class** must be in the directory tu/pmc/bca

## 4.2. Naming Convention for Packages

To get rid of the name collision especially class names, it is necessary for us to put our classes in the package so that when distributing the classes, we have no danger of ending up with the problem of having same name (since we can have same name outside the package). However, in these days of global market, we may distribute our package in any part of the world and to avoid the name collision the package name must be unique. So, to have unique name we follow the following conventions:

- Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.
- Companies use their reversed Internet domain name to begin their package names—for example, **com.example.orion** for a package named **orion** created by a programmer at example.com.
- Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, **com.company.region.project**).
- Packages in the Java language itself begin with java. or javax.
- In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

| **Legalizing Package Names** | |
|---|---|
| **Domain Name** | **Package Name Prefix** |
| clipart-open.org | org.clipart_open |
| free.fonts.int | int_.fonts.free |
| poetry.7days.com | com._7days.poetry |

## 4.3. Using Packages

There are two ways of accessing classes, interfaces etc. in another package. First, we can add the full package name in front of every class name. For example:

java.util.Date today = new java.util.Date();

This process is very cumbersome so the simpler and common approach is to use import keyword. Using this approach, you do not have to give the classes their full names. In this approach either you can import a specific class or the whole package. The import statement will be given at the top of the file just below the package statement. Here is the example for importing the package.

import java.util.*;

Now you can use

Date today = new Date();

Also, you can do

import java.util.Date;

When importing a number of packages, you need to pay attention to packages for a name conflict. For example, both the *java.util* and *java.sql* packages have a Date class. Suppose you write a program that imports both packages.

```
import java.util.*;
import java.sql.*;
```

If you now use the Date class, then you get a compile-time error:

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

The compiler cannot figure out which Date class you want. You can solve this problem by adding a specific import statement:

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

What if you really need both Date classes? Then you need to use the full package name with every class name.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Locating classes in packages is an activity of the compiler. The bytecodes in class files always use full package names to refer to other classes.

**Note:** The Java compiler automatically imports three entire packages for each source file: (1) the package with no name, (2) the java.lang package, and (3) the current package (the package for the current file).

## 4.4. Finding Packages and CLASSPATH

The java run-time system knows about packages in two ways. First, by default, the java run-time system uses the current working directory as its starting point. Thus, if our package is in the current directory, or a subdirectory of the current directory, it will be found. Second, we can specify a directory path or paths by setting the **CLASSPATH** environmental variable. We can set CLASSPATH environment variable in Windows shell. For example,

**set CLASSPATH=c:\users\dell\desktop\packagetest**

This class path is set until the shell exists. You can alternatively use advanced system setting of your Widows to set CLASSPATH environment variable.

# Exercises

1. Create a base class named Bonus. The class should contain two fields salesID of type String and salesAmount of type double. Include a getetBonus method that calculates a salesperson's bonus using *bonus* = *sales* * 0.05. Create a derived class named PremiumBonus from Bonus. The derived class's getPremiumBonus method should calculate the bonus using *bonus* = sales * 0.05 + (*sales* – 2500) * 0.01. Write a class with main method to create object of PremiumBonus class and use this object to find both bonus and premium bonus.
2. Write a Java program that uses interface to support multiple inheritance.
3. Assume that a bank maintains two kinds of account for its customers, one called savings account and the other current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level, a service charge is imposed.

   Create a class **Account** that stores customer name, account number and type of account. From this derive the classes **Curr_acct** and **Sav_acct** to make them more specific to their requirements. Include the necessary method in order to achieve the following tasks:
   a. Accept deposit from a customer and update the balance.
   b. Display the balance.
   c. Compute and deposit interest.
   d. Permit withdrawal and update the balance.
   e. Check for the minimum balance, impose penalty, if necessary, and update the balance.