Object-Oriented Programming in Java

by DR DANNY POO
www.DrDannyPoo.com

Object-Oriented Programming in Java

Community Edition

Danny Poo

PREFACE

Who Should Read This Book
This book teaches object-oriented programming in Java. Designed for readers with basic knowledge of Java programming, this book is a book crafted with the reader in mind. Complete source code is provided in every example, and where applicable, screen-shots showing the development of application programs are included to help readers in their practices.

Topics Covered
Topics covered include: object; class; single class inheritance; encapsulation; polymorphism; abstract class, abstract method, inner class, method overriding, multiple class inheritance and interface.

How This Book Is Organized
This book is organized into six chapters.

Chapter 1: Introduction
This chapter begins with an overview of object-oriented programming concepts.

Chapter 2: Class and Objects
Class and Object are the fundamental concepts of object-oriented programming. This chapter defines what a class and object is and provides a foundation for further discussion on object-oriented programming.

Chapter 3: Inheritance
Inheritance is an object-oriented mechanism for realizing software reuse. A subclass in single class inheritance can inherit properties from a superclass. When a subclass inherits properties from more than one superclass, we have multiple class inheritance. This chapter discusses concepts related to single class inheritance.

Chapter 4: Encapsulation
Encapsulation is the bringing together of data fields and methods into an object definition with the effect of hiding the internal workings of the data fields and methods from the users of the object. Any direct access and updates to the object's constituents is not permissible and changes to the data fields can only be carried out indirectly via a set of publicly available methods. This chapter focuses on data field encapsulation and class encapsulation.

Chapter 5: Polymorphism
The ability of objects of different subclass definition to respond to the same message is polymorphism. Polymorphism is only possible with dynamic binding – the capability of determining which method implementation to use for a method at runtime. This chapter explains the concept of polymorphism and its peripheral object-oriented programming concepts.

Chapter 6: Interface
Sometimes it is necessary to derive a subclass from several classes but the Java extends keyword does not allow for more than one parent class. With interfaces, multiple class inheritance is possible. This chapter explains and shows how to use the Java interface construct to realize multiple class inheritance.

Enjoy!

Dr Danny Poo
www.DrDannyPoo.com

About THe Author

Dr. Danny Poo brings with him 40 years of Software Engineering and Information Technology and Management experience. A graduate from the University of Manchester Institute of Science and Technology (UMIST), England, Dr. Poo is currently an Associate Professor at the Department of Information Systems and Analytics, National University of Singapore.

A well-known speaker in seminars, Dr. Poo has conducted numerous in-house training and consultancy for organizations, both locally and regionally. His notable teaching credentials include • Data Strategy • Data StoryTelling • Data Visualisation • Big Data Analytics • Machine Learning • Data Management • Data Governance • Data Architecture • Capstone Projects for Business Analytics • Software Engineering • Server-side Systems Design and Development • Information Technology Project Management • Health Informatics • Healthcare Analytics • Health Informatics Leadership.

Dr. Poo has also published extensively in conferences and journals on Software Engineering and Information Management.

Dr. Poo was the founding Director of the Centre for Health Informatics. This Centre provides courses to train healthcare professionals in Health Informatics. Dr. Poo is instrumental in developing curriculum and courses for this Centre. In particular, he has delivered numerous rounds of Health Informatics course since 2012 and has trained as many as 1000 healthcare and IT professionals on this subject. Besides, he teaches a course on Healthcare Analytics to healthcare professionals since it started in May 2015. To date, he has run fourteen 3-full-days-sessions of this course since May 2015. This course continues to receive great interest from participants.

Dr. Poo has authored a number of books including “Java Programming”, “Object-Oriented Programming”, “Graphical User Interface Programming in Java”, “Python Programming”, and “Learn to Program Enterprise JavaBeans 3.0”.

Dr. Poo has consulted for these companies • Deutsche Bank • Gemplus • Micron • NCR • PIL • PSA • Rhode-Schwarz • Standard Chartered Bank • Singapore Technologies Electronic • Monetary Authority of Singapore (MAS) • Infocomm Development Authority (IDA) • National Library Board (NLB) • Ministry of Manpower (MOM) • Nanyang Technological University (NTU) • Nanyang Polytechnic (NYP) • National University Hospital.

CHAPTER 1: INTRODUCTION

This book "**Object-Oriented Programming in Java**" looks at Java from an Object-Oriented Software Engineering perspective. It focuses on the underlying concepts of the object-oriented programming paradigm and shows how Java can be used to develop object-oriented applications. The concepts include:

1. Class and Object
2. Inheritance
3. Encapsulation
4. Polymorphism

In the process of delivering these concepts, this book will also discuss:

1. Method Overriding
2. Abstract Class
3. Interface

1.1 Object-Oriented Programming

The procedural approach to programming has been the predominant approach to creating software applications during the early days of computing. Modularization of code has been based on system processes where the steps in carrying out a task become the focus of the code design. For example, to develop a library application system, processes such as the checking in and out of books, making reservations of books, cataloging of books, etc. will be the focus of the approach. The analysis of these processes in terms of the procedural tasks and the production of a system whose representation is based on the procedural flow of the processes describe the procedural approach.

Object-oriented programming, on the other hand, models objects and their interactions in the problem space and the production of a system based on these objects and their interactions. Since the real-world problem domain is characterized by objects and their interactions, a software application developed using the object-oriented programming approach will result in the production of a computer system that has a closer representation of the real-world problem domain than would be the case if the procedural programming approach has been used.

1.2 Overview of the Object-Oriented Programming Concepts

At the core of object-oriented programming is the class. A **class** is a definition template from which objects are created. An **object** represents an entity in the real world that can be distinctly identified. Objects possess a unique identity, state and behavior. They interact with one another by sending messages. Such interactions exhibit object behavior.

Classes can also be related to one another in a hierarchical manner. A class in the lower part of a class hierarchy is derived from one or more classes higher up in the hierarchy. Classes lower in the hierarchy are known as **subclasses** while classes higher in the hierarchy are parent or **superclasses**.

In object-oriented programming, properties in superclasses are inherited by subclasses. This ability for subclasses to inherit properties from superclasses is known as **inheritance**. Although inheritance enhances software reuse, there are also issues to be considered. Must a subclass always take on a property from a superclass? What if a subclass requires its properties to have different implementation from its superclass?

Another concept accompanying the definition of classes is **encapsulation**. Encapsulation is the bringing together of data fields and methods into an object definition with the effect of hiding the internal workings of the data fields and methods from the users of the object. Encapsulation greatly enhances the maintainability of classes. Good software engineering practices encourage class encapsulation.

Another concept in object-oriented programming that has close relationship with subclass and superclass is **polymorphism**. An object of a subclass can be used by any code designed to work with an object of its superclass. The ability of objects of different subclass definition to respond to the same message (even when the message is for the superclass) is polymorphism (which means many forms). Polymorphism encourages generic programming and is great for software extension.

As we discuss through the above concepts, more facilities in object-oriented programming will be introduced. For example, subclasses can define method signatures that are the same as the method signatures in the superclass. The methods in the subclass are said to override the methods in the superclass. Method overriding is permissible in object-oriented programming.

Must classes always have instances? What if we intend to define classes merely for property definition with no intention to create instances from them? Object-oriented programming allows for this and such classes are labelled as **abstract classes**.

A construct in Java that is very similar to abstract class is the **interface**. An interface is a construct that contains only constants and abstract methods. It defines agreed-upon behavior that other objects use to interact with one another. It is a named collection of method definitions that do not have implementations. The implementation of the methods is provided by the classes that choose to implement the behavior.

All the above concepts are covered in this book. The next chapter begins with the discussion of class and objects. Progressively, this book will bring you through the journey of object-oriented programming. It will explain how the various concepts are inter-related with one another and how they will enable you as a software developer to build software applications that are reusable and maintainable.

CHAPTER 2: CLASS AND OBJECTS

Class and Object are the fundamental concepts of object-oriented programming; we therefore begin our discussion with an elaboration of what they are in this chapter.

2.1 Defining Class and Object

A class is a definition template for structuring and creating objects. It is a construct for defining objects of the same type. An object represents an entity in the real world that can be distinctly identified. It has a unique identity, state and behavior. The state of an object is represented by a set of data fields (Note: A data field is also known as the attribute or property of an object.) and the behavior is defined by a set of methods the object possesses. In Java, variables are used to define data fields while methods are used to represent the behavior of objects.

Objects interact with one another by sending messages. There are message-sending objects (or sender) and message-receiving objects (or receiver). A message can be accompanied by information known as parameters (or arguments).

Only valid messages are responded by the receiver. A valid message corresponds to a method which the receiver uses to fulfill his responsibility to the sender. A method is made up of a number of operations which together implements the method.

The set of methods a receiver uses to respond to the messages defines the behavior of the object.

One major difference between class and object is in the way data fields and methods are treated in classes and object. Since a class defines objects, data fields and methods are declared in classes and realized in objects with values in the data fields . On the other hand, objects are created instances of a class; each object has its own data fields (Note: Java allows for the declaration of class data fields (or class variables) which can also contain values) populated with values and methods. The values of the data fields represent the state of the objects.

2.2 A Java Class

Code 2.1 is a Java class named Circle (the line numbers on the left are not part of the code).

Code 2.1: Circle Class

```
class Circle {
    private double radius = 3.0;
    static int numberOfCircles = 0;

    public Circle() {}

    public Circle(double inRadius)  {
        radius = inRadius;
    }

    public double getRadius()  {
        return radius;
    }

    public void setRadius(double inRadius)  {
        radius = inRadius;
    }
```

```
    public double area() {
        return radius * radius * Math.PI;
    }
}
```

This class defines a data field, radius, of double type and five methods Circle(), Circle(double radius), getRadius(), setRadius(), and area().

2.2.1 Data Fields

Data fields declared in a class become part of the structure of an object when it is instantiated from the class. The data field in Circle class, radius, will be the only data field for any objects created from the class.

2.2.2 Methods

Circle(), Circle(double radius), getRadius(), setRadius(), and area() are the methods defined in the Circle class. Except for Circle() and Circle(double radius) methods, the rest of the methods form the set of available methods that other objects can invoke on the object. Only messages that conform to the requirements of these three methods are considered as valid messages. When invoked, getRadius() and area() return a double value while setRadius() does not.

2.2.3 Constructor

Circle() and Circle(double radius) are special methods known as constructors. When invoked, constructors create objects based on the class definition.

A constructor has exactly the same name as the defining class. It is designed to perform initializing actions such as initializing the data fields of objects. Constructors are overloaded making it easier to instantiate objects with different initial values.

A constructor does not have a return type and it may take in input parameters. A constructor that does not have any input parameters is known as a no-arg (or no-argument) constructor. Circle() is an example of a no-arg constructor.

2.2.4 Default Constructor

While it is good practice to define a constructor for every class, it is not imperative for a class to be declared with a constructor. If a class is declared without a constructor, a no-arg constructor with an empty body, like Line 5 in Code 2.1, is implicitly declared in the class. The no-arg constructor is the default constructor for every class declared without a constructor.

2.3 Creating Objects

A class is a blueprint that defines what an object's data and methods will be. An object is an instance created from a class. To create an instance is commonly referred to as **instantiation**.

2.3.1 The new Operator

In Code 2.2, we define a user class, CircleMain, to illustrate object creation using the Circle class of Code 2.1.

Code 2.2: CircleMain Class

```
class CircleMain {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        Circle.numberOfCircles++;
        System.out.println("The area of Circle c1 is " + c1.area());
        Circle c2 = new Circle(8.0);
        Circle.numberOfCircles++;
        System.out.println("The area of Circle c2 is " + c2.area());
        print(c1);
        print(c2);
        System.out.println("Number of Circle objects created = " +
                           Circle.numberOfCircles);
    }

    static void print(Circle c) {
        System.out.println("The area of the circle with radius " +
                           c.getRadius() + " is " + c.area());
    }
}
```

To create an object from a class, we make use of the new operator to invoke a constructor on the class: new ClassName(arguments);

A circle c1 is created in Line 3 using the no-arg constructor Circle(). This line

```
System.out.println("The area of Circle c1 is " + c1.area());
```

prints the area of Circle c1. Another circle c2 is created but this time we make use of the other constructor with a radius of 8.0 as its input argument. This line

```
System.out.println("The area of Circle c2 is " + c2.area());
```

prints the area of Circle c2. The following five lines are printed when CircleMain is run:

```
The area of Circle c1 is 28.274333882308138
The area of Circle c2 is 201.06192982974676
The area of the circle with radius 3.0 is 28.274333882308138
The area of the circle with radius 8.0 is 201.06192982974676
Number of Circle objects created = 2
```

We will explain how the last three lines are printed in Section 2.2.5.

Effectively, the user class instantiated two Circle objects, c1 and c2. c1 has a radius of 3.0 (the default radius defined in the Circle class) and c2 has a radius of 8.0, initialized via the second constructor. Both objects, c1 and c2, make use of the area() method defined in the Circle class to return the area of the respective circle. Methods defined in a class are therefore reused in objects. Such methods are instance methods and are applicable to all objects instantiated from the same class.

2.3.2 Object Reference Variable

Objects are accessed via object reference variables which contain object references. In Line 3 of Code 2.2,

```
Circle c1
```

defines an object reference variable c1. The class Circle defines the reference type of the reference variable. The variable c1 can therefore reference any instance of the Circle class. Earlier, we make use of

```
new Circle();
```

to create a Circle object. Combining,

```
Circle c1 = new Circle();
```

assigns the newly created Circle object's reference to the reference variable c1.

In general, to create an object and assigns its reference to a reference variable, we write:

```
ClassName objectRefVar = new ClassName();
```

2.3.3 Accessing Object Data Fields and Methods

The dot notation can be used to access the data fields of an instantiated object:

```
objectRefVar.dataField;
```

and its methods:

```
objectRefVar.method(arguments);
```

For example,

```
c1.radius;
```

accesses the radius of the object referenced by c1.

To access the area() method of Circle object c1, we write:

```
c1.area();
```

This method returns the area of the Circle object referenced by c1.

2.3.4 Instance Variables and Methods

An instance variable refers to a data field belonging to an object. For example, c1.radius is an instance variable of the object referenced by c1. Instance variables are not shared and are independent from one another. Each object from the same class has its own set of instance variables. That is, the radius of object c1 is distinct from the radius of object c2. The two radius variables occupy different memory location. Changes made to the radius variable of object c1 do not affect the radius variable of object c2.

An instance method refers to a method belonging to an object. For example, c1.area() is an instance method belonging to the object referenced by c1. Since instance methods are applicable only to objects, the objects must first be created before the instance methods can be called.

2.3.5 Passing Objects to Methods

In Code 2.2, object c1 and c2 are passed to the print() method. Essentially, passing an object to a method is equivalent to passing the reference to the object to the method. The object reference is used in the print() method to call the getRadius() and area() methods of the object. The printout from the code testifies which Circle object has been called in the two calls to print() method.

2.4 Class Variables and Methods

Class variables and methods refer to data fields and methods that are applicable to the class. To define a class variable, we use the static keyword:

```
static varType classVar;
```

A class (or static) variable can be shared among the instances of the class. It stores value for the variable in a common memory location accessible by objects from the same class. In Code 2.1, we define a static variable:

```
static int numberOfCircles;
```

which is incremented each time a Circle object is created in Code 2.2. In this way, we will be able to keep track of the number of Circle objects created. There is only one numberOfCircles variable defined in the Circle class and its value is shared among the objects from the same class. That explains why the value 2 is printed in Code 2.2.

Like class variables, class methods are methods that belong to a class and are not specific to an instance. In Code 2.2, two methods main() and print() have been declared as static. They are examples of class method. This means that when CircleMain class is instantiated, these two methods do not form part of the definition of the CircleMain object.

2.4.1 Constant

To declare a constant, we add a final keyword to a static variable declaration:

```
final static constantType constantName = value;
```

For example, we can declare PI, the ratio of the circumference of a circle to its diameter, as a constant in the following manner:

```
final static double PI = 3.141592653589793;
```

By convention, constants in Java are written in uppercase.

2.5 The this Keyword

Consider Code 2.3. The method setNumber() takes in a number argument to replace the number data field of the object. This code compiles and no error is reported. What do you think happens when main() in NumberMain is executed? Will it print 0 or 8?

Code 2.3: Number and NumberMain Class

```
class Number {
       private int number = 0;

    public Number() {
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        number = number;
    }
}

class NumberMain {
       public static void main(String[] args) {
              Number n1 = new Number();
              n1.setNumber(8);
              System.out.println(n1.getNumber());
       }
}
```

The above code prints 0 at the console. Why? The number variable is the input argument passed in to setNumber(). Assigning a variable to itself like:

```
number = number;
```

is permissible in Java but nothing happens since the value of number is not changed.

Our intention is to assign the input argument value to the number data field of the object. How do we achieve this? A quick solution is to rename the input argument to another name like:

```
public void setNumber(int n) {
     number = n;
}
```

2.5.1 Using this Keyword in Instance Method

Another solution is to make use of the this keyword. Java provides the this keyword to distinguish an object's data field from the formal parameter definition. We modify setNumber() as follows:

```
public void setNumber(int number) {
     this.number = number;
}
```

this.number (Note: Note that There is no need to use the this keyword for every instance variable. The this keyword is used within instance method when there is ambiguity in the use of variable names.) refers to the instance variable number of the object referred to during execution.

Since n1 is the object referred to in:

```
n1.setNumber(8);
```

The this keyword is replaced with n1 resulting in:

```
n1.number = number;
```

or

```
n1.number = 8;
```

thus updating the instance variable number of n1 to the value 8.

In like manner, a new object n2's call to setNumber() will result in the replacement of this to n2, resulting in the update of the instance variable number of n2 to the new value.

2.5.2 Using this Keyword in Constructor

The this keyword can also be used in constructors to invoke another constructor. Let us introduce another constructor in the Number class:

```
public Number(int number) {
    this.number = number;
}
```

This constructor is used to create an object with a set value provided in the formal parameter. We modify the no-arg constructor using the this keyword to call the above constructor:

```
public Number() {
    this(3);
}
```

Note the way this is used; there is no dot but parenthesis. When a new object is instantiated in the main() using the no-arg constructor:

```
Number n1 = new Number();
```

n1 will have data field number initialized to 3 instead of 0 as was the case previously. For another object instantiated via the other constructor:

```
Number n2 = new Number(8);
```

n2 will be initialized with the value 8 for its data field number.

2.6 Inner Class

Can a class be defined within a class? Yes, Java allows a class to be defined as an inner class. An inner class is a class defined within the scope of another class and is nested within a class. It supports the work of the outer class which contains it. It can reference the data and methods defined in the outer class in which it nests. With an inner class, you do not need to pass the reference of an object of the outer class to the constructor of the inner class. Inner classes, therefore, can make programs simple and concise.

2.6.1 Compiling an Inner Class

An inner class is compiled into a class named OuterClassName$InnerClassName.class. An inner class can be declared public, protected, or private (see Chapter 4) subject to the same visibility rules applied to a member of the class.

Consider the example in Code 2.4. InnerClass is defined as an inner class of OuterClass. When InnerClass is compiled, a class named OuterClass$InnerClass.class is produced.

The InnerClass is within the scope of OuterClass. That is why

```
outerData++;
m();
```

do not produce any errors when they are accessed within the method n() of InnerClass.

2.6.2 Static Inner Class

An inner class can also be declared static. Since an inner class is within the scope of an outer class, a static inner class can be accessed using the outer class name. However, a static inner class cannot access non-static members of the outer class.

Code 2.4: Inner Class

```
class OuterClass {
  private int outerData = 3;

  public void m() {
    InnerClass instance = new InnerClass();
  }

  public int getOuterData() {
       return outerData;
  }

  class InnerClass {
       private int innerData = 8;

    public void n() {
      outerData++;
      m();
    }

    public int getInnerData() {
        return innerData;
    }
  }

  public static void main(String[] args) {
       OuterClass oo = new OuterClass();
       OuterClass.InnerClass io = oo.new InnerClass();
       System.out.println("InnerClass object innerData = " + io.getInnerData());
       System.out.println("OuterClass object outerData = " + oo.getOuterData());
  }
}
```

2.6.3 Creating Inner Class Objects

Objects of an inner class are often created in the outer class. They can also be created from another class. If an inner class has been declared as a non-static class, an instance of the outer class must first be created before you can create an object for the inner class as follows:

```
OuterClass oo = new OuterClass();
OuterClass.InnerClass io = oo.new InnerClass();
```

If the inner class is static, use the following syntax to create an object for it:

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

CHAPTER 3: INHERITANCE

Software reuse is an important aspect of system development. It has many advantages. By reusing what has been developed, we can achieve simplified and well-tested code that results in software systems that are highly maintainable. At the programming level, software reuse can be achieved using the inheritance mechanism commonly found in object-oriented programming languages such as Java.

3.1 The Inheritance Mechanism

The inheritance mechanism in object-oriented programming allows you to derive new classes from existing classes. The derived class takes on the general properties (which include its data fields and methods) of the existing class. The inherited properties then form part of the derived class' definition.

3.1.1 Subclass and Superclass

A class C1 derived from a class C2 is a subclass of C2 and C2 is known as the superclass. A subclass is also known as a child class or extended class. A superclass is referred to as parent class or base class. A subclass inherits accessible data fields and methods from its superclass, and may also add new data fields and methods.

3.1.2 java.lang.Object Class

All new classes that you define derive from some existing classes, either explicitly or implicitly. In Java, all classes derive themselves implicitly from the java.lang.Object class. The latter is commonly known as the mother of all Java classes.

3.1.3 Downward Property Propagation

The inheritance mechanism permits only downward propagation of properties from superclasses to subclasses. There is no upward propagation of properties. Therefore, information specific to subclasses are unique to subclasses and are not propagated to superclasses.

3.2 Demonstrating Inheritance

Code 3.1 is a modified Circle class we first introduced in Chapter 2.

Code 3.1: Circle Class

```
class Circle {
    private double radius = 1.0;

    public Circle() {}

    public Circle(double radius)  {
        this.radius = radius;
    }

    public double getRadius()  {
        return radius;
    }
```

```
    public void setRadius(double radius)  {
        this.radius = radius;
    }

    public double area() {
        return radius * radius * Math.PI;
    }
}
```

Let us introduce a new class – the Cylinder class. This class is derived from the Circle class since Circle is more general than Cylinder. The Circle class is the superclass of Cylinder class and we express this relationship in Java using the extends keyword. Code 3.2 is the code for the Cylinder class.

The Cylinder class extends the Circle class in

```
class Cylinder extends Circle
```

This effectively makes Cylinder class a subclass of Circle class. All accessible properties of Circle class are now available to the Cylinder class as part of the latter's definition.

To show how inheritance works, we will create a user class – CylinderMain. Code 3.3 is the code for CylinderMain.

Code 3.2: Cylinder Class

```
class Cylinder extends Circle {
    private double length = 1.0;

    Cylinder() {}

    public double getLength() {
            return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double area() {
        return 2 * super.area() +
                   2 * getRadius() * Math.PI * length;
    }

    public double volume() {
            return super.area() * length;
    }
}
```

Code 3.3: CylinderMain Class

```
class CylinderMain {

    CylinderMain() {}

    public static void main(String[] args) {
        Cylinder cyd = new Cylinder();
        System.out.println("Cylinder length = " + cyd.getLength());
        System.out.println("Cylinder radius = " + cyd.getRadius());
            System.out.println("Cylinder volume = " + cyd.volume());
            System.out.println("Cylinder area   = " + cyd.area());
    }
}
```

When main() is executed, the following is printed on the console:

```
Cylinder length = 1.0                 ◄—— from Cylinder class
Cylinder radius = 1.0                 ◄—— from Circle class
Cylinder volume = 3.141592653589793   ◄—— from Cylinder class
Circle area     = 3.141592653589793   ◄—— from Circle class
```

How is that so? The printouts were produced by four System.out.println() statements in main(). cyd.getLength() refers to the getLength() method in Cylinder class. The getRadius() method has been inherited from the Circle class since Cylinder class does not define this method. Through this method, the radius of the cylinder is derived. The volume of the cylinder is produced by the volume() method in the Cylinder class (Code 3.2). Finally, the Cylinder class inherits from the Circle class the area() method so that it can produce the area of the circle in Code 3.3.

3.3 The super Keyword

In the previous chapter, the this keyword was used to refer to the instance in which this is used. In Java, the super keyword is used to refer to the superclass of the class in which super appears. It is used to call the superclass constructor or method.

3.3.1 Syntax

There are two forms:

```
super() or super(arguments)
```

The former is used to invoke the no-arg contructor of the superclass. The latter is used to invoke the superclass constructor that matches the arguments. There is a restriction on the use of the super() and super(arguments). These two statements must appear in the first line of the subclass constructor. This is the only way in which the superclass constructor can be invoked.

For example, suppose we want to change the radius of the Cylinder object to 3.0 instead of the default value 1.0 as defined in the Circle class. We can change this value in the Cylinder constructor as follows:

```
Cylinder() { super(3.0); }
```

super(3.0) will match with the constructor in Circle class (Code 3.1). The radius is replaced with the value 3.0.

3.3.2 Constructor Chaining

As we have discussed in Chapter 2, a constructor can invoke an overloaded constructor of the same class. With inheritance, a constructor may also invoke its superclass' constructor. Invocation of the superclass' constructors may be done explicitly or implicitly. An example of explicit invocation of a superclass' constructor using a super() statement has been given earlier:

```
Cylinder() { super(3.0); }
```

If any of the constructors in the subclass does not invoke the superclass constructor explicitly (with a super() statement), a super() statement will be placed within the subclass constructor implicitly by the Java compiler. Thus,

```
Cylinder() {}
```

will become:

```
Cylinder() { super(); }
```

If there are other statements in a subclass constructor, the super() statement will be placed before those statements. For example,

```
Cylinder() {

   // some statements

}
```

will become:

```
Cylinder() {
    super();
   // some statements

}
```

In any case, the construction of an instance of a class will invoke the superclass' constructors along the inheritance chain. This is known as constructor chaining.

Code 3.4: Alpha, Bravo, and Charlie Class

```
public class Charlie extends Bravo {

    public static void main(String args[]) {
            new Charlie();
    }

    public Charlie() {
            System.out.println("in Charlie's no-arg constructor");
    }
}

class Bravo extends Alpha {
    public Bravo() {
            this("in Bravo's overloaded constructor");
            System.out.println("in Bravo's no-arg constructor");
    }

    public Bravo(String s) {
            System.out.println(s);
    }
}

class Alpha {
    public Alpha() {
            System.out.println("in Alpha's no-arg constructor");
    }
}
```

Let us illustrate constructor chaining with an example. Code 3.4 contains three classes Alpha, Bravo and Charlie. Class Charlie is a subclass of class Bravo which in turn is a subclass of Alpha. main() is in class Charlie.

When the code is run, the following is produced at the console:

```
in Alpha's no-arg constructor
in Bravo's overloaded constructor
in Bravo's no-arg constructor
in Charlie's no-arg constructor
```

Following through the execution of the code beginning with main() in class Charlie, you will note that the super(); statement has been implicitly placed in Charlie(), Bravo() and Alpha() no-arg constructors to call their respectively superclass' constructor. The superclass of Alpha is java.lang.Object class.

3.3.3 Calling Superclass Methods

super can also be used to reference a method other than the constructor in a superclass as follows:

```
super.method(arguments);
```

For example, in the volume() method of the Cylinder class (Code 3.2), a reference is made on the area() method of the Circle class, a superclass of Cylinder. The volume() method can also be expressed as:

```
public double volume() {
   return super.area() * length;
}
```

3.4 Method Overriding

A subclass inherits data fields and methods from superclasses up in the inheritance hierarchy. The implementation of the methods that a subclass inherits from its superclasses is fixed and cannot be changed by the subclass. However, there are times when it is necessary for the subclass to modify the implementation of the methods defined in the superclasses.

Consider, for example, the surface area of a cylinder. We will make use of the same inheritance relationship we had earlier discussed in Code 3.1 and Code 3.2 where Cylinder is a subclass of Circle. The surface area of a cylinder is calculated using the following formula:

```
2 * area of Circle + 2 * radius * PI * length (or height) of cylinder
```

Although Cylinder inherits from the Circle class an area() method, the implementation does not meet the requirement. To resolve this problem, Java allows the subclass to modify the implementation of the method in the superclass by creating a method in the subclass which has the same name and formal parameters as its counterpart in the superclass. This ability is known as method overriding.

To illustrate, we will modify the Cylinder class; the modified class is shown in Code 3.5.

Code 3.5: The Modified Cylinder Class

```
class Cylinder extends Circle {
    private double length = 1.0;

    Cylinder() {}

    public double getLength() {
          return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double area() {
        return 2 * super.area() +
                   2 * getRadius() * Math.PI * length;
    }
```

```
    public double volume() {
            return super.area() * length;
    }
}
```

CylinderMain is also modified to reflect a call to the area() method in Cylinder class. The modified CylinderMain is shown in Code 3.6.

Code 3.6: The Modified CylinderMain Class

```
class CylinderMain {

    CylinderMain() {}

    public static void main(String[] args) {
        Cylinder cyd = new Cylinder();
        System.out.println("Cylinder length = " + cyd.getLength());
        System.out.println("Cylinder radius = " + cyd.getRadius());
            System.out.println("Cylinder volume = " + cyd.volume());
            System.out.println("Cylinder area   = " + cyd.area());
    }
}
```

When cyd.area() is called in main(), the area() method in the Cylinder class (in Code 3.5) is executed instead. The area() method in Cylinder class is said to overrides the area() method in Circle class. Note that within the area() method of the Cylinder class, the area() method of Circle is called via super.area(). In Code 3.5, we have added super to the area() method call so that it is the area() method of the Circle class that is invoked and not the area() method of the Cylinder class.

When main() completes its run, the following is produced on the console:

```
Cylinder length = 1.0
Cylinder radius = 1.0
Cylinder volume = 3.141592653589793
Cylinder area   = 12.566370614359172
```

CHAPTER 4: ENCAPSULATION

Using the dot notation to access the data fields of an object increases the dependency between the object and the calling objects. This effectively reduces the maintainability of the object since any change in the definition of the data fields will adversely affect the calling objects. Good software engineering dictates the use of encapsulation when classes are defined.

Encapsulation is the bringing together of data fields and methods into an object definition with the effect of hiding the internal workings of the data fields and methods from the users of the object.

Any direct access and updates to the object's constituents is not permissible and changes to the data fields can only be carried out indirectly via a set of publicly available methods.

In this chapter, we will discuss data field encapsulation and class encapsulation.

4.1 Access Modifiers: public, protected, private

To impose visibility constraint on a data field or method, we place an access modifier in front of the data field or method declaration.

For example, in Code 4.1, we have declared four class variables with access modifiers. These variables are accessible from within the class.

Code 4.1: Access Modifiers

```
class A {
    public    static int public_access;
    protected static int protected_access;
              static int default_access;
    private   static int private_access;

    public static void main(String[] args) {
        int i;
        i = public_access;
        i = protected_access;
        i = default_access;
        i = private_access;
    }
}
```

In Table 4.1, we explain the accessibility of each of the variables.

TABLE 4.1: Accessibility of Data Fields

Field	Access Modifier	Accessibility
`public_access`	`public`	Accessible from anywhere outside the class A. No restriction on accessible `public_access`
`protected_access`	`protected`	Accessible from within any class in the same package, and also from within any class that inherits directly or indirectly from class A.
`default_access`	*default (friendly)*	Accessible from within any class in the same package. This is also known as *package-private* or *package-access*.
`private_access`	`private`	Accessible only from within the same class. This is the most restrictive type of access.

4.1.2 Accessibility Effects

Table 4.2 summarizes the effects of the access modifiers on variables in general.

TABLE 4.2: Access Modifiers and their level of Accessibility

Keyword	Access
`public`	Access to class, data field or method is unrestricted and may be accessed from anywhere in the program. **(Most Accessible)**
default (friendly)	Access to class, data field or method is only allowed within any class in the *same package*.
`protected`	Access to data field or method is allowed from within any class in the same package, and also from within any class that inherits directly or indirectly from the class.
`private`	Access to data field or method is only allowed from within the same class. (**Least Accessible**)

Let us now add a user class B to our example (see Code 4.2). Class B will be making calls to access the data fields in Class A. We will assume that both classes A and B are located in the same directory. All data fields in class A are accessible by methods in class B except for data field private_access which produces a compilation error when accessed. private_access is accessible only from within class A.

Code 4.2: User Class

```
class B {
    public static void main(String[] args) {
        int i;
        i = A.public_access;
        i = A.protected_access;
        i = A.default_access;
        i = A.private_access; // Compilation error occurs here.
    }
}
```

4.2 Data Field Encapsulation

To prevent direct modifications of data fields through the object reference using the dot notation, declare the data field private, using the private access modifier. This is known as data field encapsulation. For example, when we define the Circle class (see Code 3.1) in Chapter 3, the data field radius was declared private:

```
private double radius = 1.0;
```

A private data field cannot be accessed by an object through a direct reference outside the class that defines the private field. To make a private data field accessible, provide a get method (also known as a getter or accessor method) to return the value of the data field:

```
public returnType getDataFieldName() {
    // statements
}
```

To enable a private data field to be updated, provide a set method (also known as a setter or mutator method) to set a new value.

```
public void setDataFieldName(dataType dataFieldValue) {
    // statements
}
```

For example, in the Circle class in Code 3.1, a getter and setter method was defined:

```
public double getRadius()  {
    return radius;
}

public void setRadius(double radius)  {
    this.radius = radius;
}
```

These two methods are used to access and update the radius data field in Circle objects. Note that in order for getRadius() and setRadius() to be accessible from the user class, these two methods must be declared public.

4.3 Class Abstraction

There are two aspects to a class: its implementation and its use. The implementation of a class is made up of a set of data fields and methods. Class abstraction is the separation of class implementation from the use of a class. This suggests that not all the class' data fields and methods are exposed to the clients to use.

When a class is defined, a description of the class on how it can be used is provided to the user. The set of methods in the description forms the interface to the class.

The collection of data fields and methods that is accessible from outside the class, together with a description of how they are expected to behave, serves as the class' contract to the user object.

4.4 Class Encapsulation

By bringing data fields and methods together and hiding their implementation from its users we achieve class encapsulation.

Class encapsulation on the Circle class in Code 3.1 is achieved by applying data field encapsulation on radius (by declaring it private) and providing a description of methods that a user object can call (by declaring the accessible methods: getRadius(), setRadius(), and area() public).

4.4.1 Encapsulating a Class

A stack is a common data structure used in programming. In Code 4.3, we define a Stack class with two data fields – an array to store the contents and an index pointing to the current element in the array. These two data fields are declared private; they cannot be accessed directly by users and their implementations are thus hidden from the users.
Two methods empty() and full() have also been declared private. Users of the class will not know how a Stack object determines if it is empty or full.

Users access the Stack object via two methods, push() and pop(), which have been declared public. push() returns a boolean value. If the value is true, the stack is not full and the integer is pushed into the stack. If the value is false, the stack is full and the push is unsuccessful. pop() returns an integer. If the value is not -1, the stack is not empty and the value returned is the current integer referenced in the stack. The two methods push() and pop() forms the Stack class' contract to the users.

Code 4.3: Stack Class

```
class Stack {
    private int content[] = new int[10];
    private int index     = 0;

    Stack() {}
```

```
    private boolean empty() {
        // returns true if array has no item
        return (index==0)?true:false;
    }

    private boolean full() {
        // returns true if array is filled
        return (index>9)?true:false;
    }

    public boolean push(int i) {
        // inserts an item i into array if not full
        if (!full()) {
            content[index++] = i;
            return true;
        } else
            return false;
    }

    public int pop() {
        // inserts an item i into array if not empty
        if (!empty())
            return content[--index];
        else
            return -1;
    }
 }
}
```

To use the Stack class, we implement StackMain as a user class. This class is given in Code 4.4. StackMain references the Stack object in two places using the push() and pop() methods. It is clear from the code that StackMain does not know how the number pushed into the stack is stored in the stack, neither does it know how the stack determines if it is empty or full.

Code 4.4: StackMain Class

```
class StackMain {

    public static void main(String[] args) {
        int number = 1;
        Stack s = new Stack();
        while (s.push(number)) { // pushed only if not full
            System.out.println("Number pushed is " + number);
            number++;
        }

        System.out.println();

        boolean thereAreItems = true;
        while (thereAreItems) {
            number = s.pop();
            if (number != -1) // Stack is empty when -1 is returned
                System.out.println("Number popped is " + number);
            else
                thereAreItems = false;
        }
    }
}
```

4.4.2 Enhanced Maintainability

By encapsulating and hiding the implementation of the Stack class, maintainability of the class is enhanced. Let us illustrate this by making a change to the way the Stack class stores its contents. Code 4.5 is the revised Stack class. In the revised Stack class, we have used a linked list instead of an array to store the integers. Accompanying this new implementation is the creation of a StackItem class (Code 4.6). This class serves as a linked list managing the items pushed and popped from the Stack object.

The data fields of the Stack class have been changed. Since they are declared private, they are not accessible to users. The data field max is an extra data field included to prevent endless execution on the code. It is really not part of the actual implementation of the revised Stack class.

As highlighted in Code 4.5, the implementation of the methods has been changed but the methods' signature remains the same. Since users interface with the Stack objects via push() and pop(), no change is required on the part of the users when calls are made on these two methods.

The user class StackMain remains the same as shown in Code 4.7. The use of class encapsulation has prevented unnecessary changes in the user class when the internal implementation of the Stack class is changed.

Code 4.5: Revised Stack Class

```
class RevisedStack {
    private StackItem top, temp;
    private int size;
    private int max = 10;

    RevisedStack() {
        top  = null;
        size = 0;
    }

    private boolean empty() {
        return (size == 0);
    }

    private boolean full() {
        return (size == max);
    }

    public boolean push(int i) {
        if (full())
               return false;
        else {
               temp = top;
               top  = new StackItem();
            top.setPrevious(temp);
            top.setItem(i);
            size = size + 1;
            return true;
        }
    }

    public int pop() {
        int i = 0;
        if (empty())
               return -1;
        else {
               i    = top.getItem();
            top  = top.getPrevious();
            size = size-1;
            return i;
        }
    }
```

```
}
```

Code 4.6: StackItem Class

```
class StackItem {
    private int item=0;
    private StackItem previous;
    public int getItem() {return item;}
    public void setItem(int x) {item=x;}
    public StackItem getPrevious() {return previous;}
    public void setPrevious(StackItem p) {previous=p;}
    StackItem() {previous=null;}
}
```

Code 4.7: StackMain Class

```
class StackMain {

    public static void main(String[] args) {
        int number = 1;
        RevisedStack s = new RevisedStack();
        while (s.push(number)) { // pushed only if not full
            System.out.println("Number pushed is " + number);
            number++;
        }

        System.out.println();

        boolean thereAreItems = true;
        while (thereAreItems) {
            number = s.pop();
            if (number != -1) // Stack is empty when -1 is returned
                System.out.println("Number popped is " + number);
            else
                thereAreItems = false;
        }
    }
}
```

The output from the run remains the same:

```
Number pushed is 1
Number pushed is 2
Number pushed is 3
Number pushed is 4
Number pushed is 5
Number pushed is 6
Number pushed is 7
Number pushed is 8
Number pushed is 9
Number pushed is 10

Number popped is 10
Number popped is 9
Number popped is 8
Number popped is 7
Number popped is 6
Number popped is 5
Number popped is 4
Number popped is 3
Number popped is 2
Number popped is 1
```

4.4.3 Bundling and Information Hiding

The design of the Stack class exhibits the two attributes of encapsulation: Bundling and Information Hiding. The data fields and methods have been bundled into the definition of a class. Implementation of the data fields using array or linked list have been hidden from the users using the private access modifier. Access to the data fields has been limited to two methods (push() and pop()) which have been declared public. The implementation of these methods is hidden from the user class (Information Hiding) i.e. the user class does not know how the methods are implemented. However, the user class is aware of how to call the two methods.

Bundling is the act of associating a set of methods with a set of data fields such that the methods are the only means that can affect the values of the data fields. Information Hiding refers to the hiding of the implementation of data fields and methods from their users. Information Hiding prevents direct access to data fields and limits access to the data fields to a set of predetermined and publicly available methods.

Bundling and Information Hiding result in a situation where users are aware and able to call the methods of objects but they do not know how the methods are implemented internally.

CHAPTER 5: POLYMORPHISM

When we covered Inheritance in Chapter 3, we discussed subclass and superclass. Another concept in object-oriented programming that has close relationship with subclass and superclass is polymorphism.

An object of a subclass can be used by any code designed to work with an object of its superclass. The ability of objects of different subclass definition to respond to the same message (even when the message is for the superclass) is polymorphism (which means many forms).

We will illustrate the concept of polymorphism in this chapter.

5.1 Illustrating Polymorphism with Geometric Shapes

Triangles and rectangles are geometric shapes. When they are placed in an inheritance hierarchy, triangles and rectangles belong to classes that are subclasses of a more general class, GeometricShape. Figure 5.1 illustrates the relationship among Triangle, Rectangle and GeometricShape class. GeometricShape is the superclass of Triangle and Rectangle class.

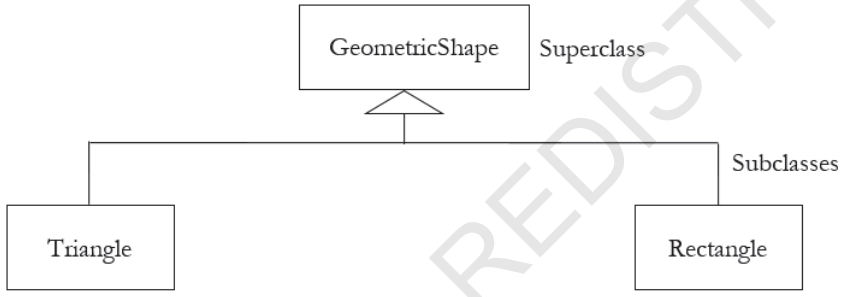

FIGURE 5.1: GeometricShape and its Subclasses

5.1.1 The Triangle Class

We begin with the description of the Triangle class (see Code 5.1).

Code 5.1: Triangle Class

```
class Triangle extends GeometricShape {
    private double base;
    private double side1;
    private double side2;
    private double height;

    public Triangle() {
        this(2.0, 2.0, 3.0, 5.0);
    }

    public Triangle(double base, double height,
                    double side1, double side2) {
        this(base, height, side1, side2, "green");
    }

    public Triangle(double base, double height, double side1,
```

```
                    double side2, String colour) {
        super(colour);
        this.base   = base;
        this.side1  = side1;
        this.side2  = side2;
        this.height = height;
    }

    public String identity() {
        return super.getColour() + " Triangle";
    }

    public double getBase() { return base; }

    public void setBase(double base) { this.base = base; }

    public double getHeight() { return height; }

    public void setHeight(double height) { this.height = height; }

    public double getSide1() { return side1; }

    public void setSide1(double side1) { this.side1 = side1; }

    public double getSide2() { return side2; }

    public void setSide2(double side2) { this.side2 = side2; }

    public double calculateArea() {
        return (0.5 * base * height);
    }

    public double calculatePerimeter() {
        return base + side1 + side2;
    }

    public String toString() {
        return "Triangle -> base=" + base + " height=" + height +
               " side1=" + side1 + " side2=" + side2;
    }
}
```

The Triangle class extends the GeometricShape class. Any accessible data fields and methods defined in the GeometricShape class are inherited by the Triangle class. The calculateArea() and calculatePerimeter() method implements the method for calculating the area and perimeter of a triangle respectively. The identity() method returns the identity of the object referenced. The colour field distinguishes the object and is thus used as the distinguishing identity. The toString() method is implemented to print the data fields of a Triangle object. This method overrides the superclass' toString() method found in java.lang.Object.

5.1.2 The Rectangle Class

The code for Rectangle class is given in Code 5.2.

Code 5.2: Rectangle Class

```
class Rectangle extends GeometricShape {
    private double breadth;
    private double length;

    public Rectangle() {
        this(1.0, 1.0);
    }
```

```
    public Rectangle(double breadth, double length) {
        this(breadth, length, "blue");
    }

    public Rectangle(double breadth, double length, String colour) {
       super(colour);
       this.breadth = breadth;
       this.length  = length;
    }

    public String identity() {
        return super.getColour() + " Rectangle";
    }

    public double getBreadth() {
        return breadth;
    }

    public void setBreadth(double breadth) {
        this.breadth = breadth;
    }

    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double calculateArea() {
        return breadth * length;
    }

    public double calculatePerimeter() {
        return 2 * (breadth * length);
    }

    public String toString() {
        return "Rectangle -> breadth=" + breadth +
               " and length=" + length;
   }
}
```

The Rectangle class also extends the GeometricShape class. Any accessible data fields and methods defined in the GeometricShape class are also inherited by the Rectangle class. The calculateArea() and calculatePerimeter() method implements the method for calculating the area and perimeter of a rectangle respectively. The identity() method returns the identity of the object referenced. The colour field distinguishes the object and is thus used as the distinguishing identity. The toString() method is implemented to print the data fields of a Rectangle object. This method overrides the superclass' toString() method found in java.lang.Object.

5.1.3 The GeometricShape Class

The code for GeometricShape class is given in Code 5.3.

The GeometricShape class is defined as an abstract class. In designing classes, we should ensure that a superclass contains common features of its subclasses. However, there are times when superclasses while containing common features of subclasses may be too abstract for them to have any instances. In such situation, the superclass is defined as abstract class so that no instances can be created from the class. The colour data field is the only common feature among the subclasses, that is why the GeometricShape class defines only one data field. One of the constructors in the GeometricShape class has been declared protected. This form of access

modifier is used when it is necessary for a data field or method in a class to be accessed only within the class and its subclasses. To facilitate method overriding of the calculateArea() and calculatePerimeter() method, we have declared them abstract in the GeometricShape class. Note that the implementation of these methods has been provided in the subclasses Triangle and Rectangle.

Code 5.3: GeometricShape Class

```
abstract class GeometricShape {

    private String colour = "red";

    GeometricShape() {}

    protected GeometricShape(String colour) {
        this.colour = colour;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public abstract double calculateArea();
    public abstract double calculatePerimeter();
    public abstract String identity();
}
```

5.1.4 The User Class: GeometricShapeMain Class

Finally, we create a user class to visualize the effects of polymorphism. GeometricShapeMain is the user class whose code is given in Code 5.4.

Code 5.4: GeometricShapeMain Class

```
import java.util.LinkedList;

class GeometricShapeMain {
    static void display(GeometricShape shape) {
        System.out.println();
        System.out.println(shape.toString());
        System.out.println("The area of " + shape.identity() +
                           " is " + shape.calculateArea());
        System.out.println("The perimeter is " +
                           shape.calculatePerimeter());
    }

    public static void main(String[] args) {
        // Rectangle -> breadth=9.0, length=4.0
        GeometricShape shape1 = new Rectangle(9.0, 4.0);

        // Triangle -> base=6.0, height=4.0, side1=5.0, side2=5.0
        GeometricShape shape2 = new Triangle(6.0, 4.0, 5.0, 5.0);

        LinkedList shapeList = new LinkedList();
        shapeList.add(0, shape1);
        shapeList.add(1, shape2);

        for (int i = 0; i<shapeList.size(); i++)
            display((GeometricShape)shapeList.get(i));
    }
}
```

GeometricShapeMain creates two geometric objects – shape1 (a rectangle) and shape2 (a triangle). These two objects are added into a linked list, shapeList. The next two lines display information about the object in the list.

Note the use of generic shape in the display() method. The shape.calculateArea() and shape.calculatePerimeter() methods are bound to the respective subclass methods at runtime through dynamic binding. Since the first object encountered is a Rectangle object, the Rectangle calculateArea() and calculatePerimeter() methods are invoked in the first round of the loop in the for statement. The second object invokes the calculateArea() and calculatePerimeter() methods of the Triangle class in the subsequent loop since it is a Triangle object. As you can see from the outputs:

```
Rectangle -> breadth=9.0 and length=4.0
The area of blue Rectangle is 36.0
The perimeter is 72.0

Triangle -> base=6.0 height=4.0 side1=5.0 side2=5.0
The area of green Triangle is 12.0
The perimeter is 16.0
```

the same message – calculateArea() and calculatePerimeter() – has been used in the display() method. calculateArea() and calculatePerimeter() are said to be polymorphic. Depending on the type of the object, the appropriate subclass of GeometricShape has been called to respond to the message. This demonstrates the ability of objects of different subclass definition to respond to the same message. That is polymorphism in action.

5.2 Abstract Class

Abstract classes are very much like the regular classes in definition – they have data and methods. However, unlike regular classes, abstract classes cannot create instances using the new operator, even though abstract classes do have their own constructor methods.

An abstract method is a method signature without implementation (i.e. without the braces {..}). The implementation of abstract methods is provided by subclasses. A class that has an abstract method must be declared as an abstract class via the abstract keyword. All subclasses of an abstract class must implement (or override) any abstract methods defined in the superclass unless the subclass is defined as an abstract class itself.

By declaring calculateArea() and calculatePerimeter() abstract in the GeometricShape class, subclasses of GeometricShape must implement the methods. If any of the subclasses of GeometricShape does not implement any of these methods, an error will be reported during compilation. To avoid such situation, dummy methods for calculateArea() and calculatePerimeter() could be implemented in GeometricShape as shown in Code 5.5.

Code 5.5: Dummy Methods

```
abstract class GeometricShape {

    private String colour = "red";

    GeometricShape() {}

    protected GeometricShape(String colour) {
        this.colour = colour;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }
```

```
    public double calculateArea( return 0.0; );
    public double calculatePerimeter( return 0.0; );
    public abstract String identity();
}
```

5.3 Dynamic Binding

In Code 5.3, calculateArea() and calculatePerimeter() have been declared abstract. By so doing, we are informing Java that these methods will be implemented (or overridden) by the subclasses and that the Java Virtual Machine should use the implementation of the methods in the subclasses at runtime. This capability of determining which method implementation to use for a method at runtime is known as dynamic binding.

Dynamic binding in Java works as follows: When a method is invoked, the JVM searches for an implementation of the method from the most specific subclass (the one lowest in the class hierarchy) and works its way up the hierarchy until an implementation is found and the first-found implementation of the method is invoked.

Polymorphism is only possible with dynamic binding.

CHAPTER 6: INTERFACE

The concept of inheritance was first introduced in Chapter 3 but the kind of inheritance that we discussed was single inheritance. That is, a subclass can only inherit from one superclass. Sometimes it is necessary to derive a subclass from several classes but the Java extends keyword does not allow for more than one parent class. With interfaces, multiple class inheritance is possible.

In this chapter, we will discuss the interface construct in Java.

6.1 The Interface Construct

An interface is a construct that contains only constants and abstract methods. It defines agreed-upon behavior that other objects use to interact with one another. It is a named collection of method definitions that do not have implementations. The implementation of the methods is provided by the classes that choose to implement the behavior.

6.2 Interface Definition

An interface is declared in the following manner:

```
modifier interface InterfaceName {
    // constant declarations
    // method signatures
}
```

An interface definition is similar to a class definition except that it uses the interface keyword. All methods in an interface are abstract methods. Only the contract definition is specified in an interface. The implementation is provided by the subclass that implements it. An interface can also include constant declaration. An example of an interface is given below:

```
interface I
{
  int a = 5;  // constant declaration
  int getA(); // method with no implementation
  int getB();
}
```

6.2.1 Interface Declaration and Interface Body

There are two components in an interface definition: interface declaration and interface body.

Interface Declaration ☛ `interface I`
Interface Body ☛
```
{
  int a = 5;
  int getA();
  int getB();
}
```

An interface declaration specifies the name of the interface and other superinterfaces that the interface extends. For example,

```
interface I extends J, K, L
```

where J, K and L are themselves interfaces. J, K and L are the superinterfaces of interface I. Unlike a class, an interface can extend from any number of interfaces. Like a class, an interface inherits all the constants and methods from its superinterfaces.

An interface declaration can also include the public access specifier:

```
public interface I extends J, K, L
```

An interface with the public access specifier is accessible by any class in any package. Omitting it, the interface is said to be accessible only to classes in the same package as the interface.

An interface body contains constant and method declarations:

```
{
  int a = 5;   // constant
  int getA();  // method
  int getB();  // method
}
```

A constant is a variable with a value that cannot be changed. Classes implementing an interface inherit the constants declared in the interface.

A method declaration is the contract part of a method. Declaration of the getA() method above ends with a semicolon (;) without the braces ({}) because the implementation part of the method is not required in interfaces. All methods declared are, by default, considered as public and abstract. The use of such modifiers on methods is therefore unnecessary. The use of the private and protected access modifier is not allowed on data fields and methods in an interface.

6.2.2 Compilation of Interface

Java creates a bytecode file when an interface is compiled, just like a regular class. No objects can be instantiated from an interface using the new operator.

6.2.3 Implementing Interface

An interface defines an agreed-upon behavior that other objects use to interact with one another. Any objects that want to assume the agreed-upon behavior must implement all the methods declared in the interface; for example,

```
class Y extends X implements I {
  ...
}
```

A class that implements an interface has to provide the implementation part of the methods in the interface as well as those in its superinterfaces. If the class does not implement all the methods in the interfaces, it must be declared as abstract. The method signature in the implementing class must match the method signature declared in the interfaces (A method signature consists of the name of the method, and the number and type of formal parameters).

For example, Class Y extends Class X and implements interface I in Code 6.1. Since interface I has two superinterfaces J and K, Class Y has to implement methods from interface I, J and K too.

Code 6.1: Implementing Interface

```
class ImplementingInterface {
  public static void main(String argv[]) {
      Y y = new Y();
      System.out.println(y.getQ());
      System.out.println(y.getR());
      System.out.println(y.getS());
  }
}

class X {
  private String a = "I am A from Class X";
  X() {}
  protected String getQ() { return a; }
}

class Y extends X implements I {
  Y() {}
  public String getQ() { return "Hi, " + super.getQ(); }
  public String getR() { return "Hello, " + c; }
  public String getS() { return "Yup, " + e; }
}

interface I extends J, K {
  String a = "I am A from interface I";
  String getQ();
}

interface J {
  String c = "I am C from interface J";
  String getR();
}

interface K {
  String e = "I am E from interface K";
  String getS();
}
```

The output from Code 6.1 is:

```
Hi, I am A from Class X
Hello, I am C from interface J
Yup, I am E from interface K
```

Note that the method getQ() in Class Y overrides method getQ() in Class X. The method signature of getQ() in Class Y and Class X must be the same.

6.3 Understanding the Use of Interface

To illustrate the use of interface, let us consider Code 6.2.

Defined in Class X are two variables a and b. Two assessor methods getA() and getB() provide the means for accessing these two private data fields. Class Z is a subclass of Class X; the former inherits the two methods getA() and getB() from Class X. Class Z implements interface Y. The implementation of the methods in interface Y – in this case, getA() – has to be provided by Class Z. By implementing the interface, Class Z is essentially signing a contract and agrees to provide the implementation for all the methods defined in the interface – in this case, implementing the getA() method. The output from Code 6.2 is:

```
X's a = 2; X's b = 3
Z's a = 9; Z's b = 3
```

The first line of the output prints the value of a and b in object x. The data field b of object z has been inherited from Class X and thus has the same value 3. However, the value of a for object z is not the same as that of object x even though there is a variable a defined in Class X. Note that the implementation of getA() from which object z produces the value of a is different from the getA() method of object x:

```
public int getA() {
    return (super.getA()*2) + a;// using a from Class X and Interface Y
}
```

This produces a value 9 as shown in the output. Why? The getA() method in Class Z is an implementation of the method getA() in interface Y. Essentially, Class Z determines how the value of a is produced. There is thus no ambiguity as to which method to use to generate the value of variable a.

Code 6.2: Interface

```
class Interface {
  public static void main(String[] args) {
      X x = new X();
      Z z = new Z();
      System.out.println("X's a = " + x.getA() +
                         "; X's b = " + x.getB());
      System.out.println("Z's a = " + z.getA() +
                         "; Z's b = " + z.getB());
  }
}

class X {
  private int a = 2;
  private int b = 3;
  X() {}
  protected int getA() { return a; }
  protected int getB() { return b; }
}

// let Z decides how a should be displayed
class Z extends X implements Y {
  Z() {}
  public int getA() {
    return (super.getA()*2) + a; // using a from Class X and Interface Y
  }
}

interface Y {
  public int a = 5;
  public int getA();
}
```

6.4 What and How in the Use of Interface

The use of interface dictates what needs to be done (the agreed-upon behaviour) and does not specify how they are done. When two similar methods are available from the superclasses, it is the subclass that decides how the implementation of method is to be achieved. This approach resolves the conflict in method selection found in Multiple Class Inheritance. For example, Class Z inherits getA() method from both Class X and Interface Y. Since Class Z chooses the implementation, it decides how getA() is implemented. The getA() method in Class Z is said to override the getA() method of Class X. Note that interface Y declares a constant a of value 5, this constant (Note: A constant is a variable with a value that cannot be changed.) is also inherited by Class Z.

We can see from the above example that interface provides a means for separating the implementation from the contract and since implementation is always determined at coding time, no ambiguity can ever exist to cause indeterminate execution of methods that produces errors.

6.5 Application of Interface

In this section, we will discuss the application of two interfaces found in the Java 2 Standard Edition API. The two interfaces are given in Table 6.1:

TABLE 6.1: Comparable and Comparator Interfaces

Interface	Package	Method	Description of Method
Comparable	java.lang	public int **compareTo** (object o)	Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
Comparator	java.util	public int **compare** (object o1, object o2) public boolean **equals** (Object obj)	Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. Indicates if object obj is equal to this Comparator

An object that implements the Comparable interface is said to have the capability to compare itself to another object of the same type. The comparison is done via the method compareTo() . This method has to be provided by the implementing class. We will look at the compare() method of the Comparator interface later in our discussion.

6.5.1 Sales Person Application

An employee of a company has three attributes: name, age, and basic salary. A sales person is a specialized type of employee. Like all other employees, a sales person has the three attributes plus an additional attribute: commission. There are five sales persons and their data are given in Table 6.2.

TABLE 6.2: Sales Person Data

Name	Age	Basic Salary	Commission
John	25	1500	2300
Mary	18	3000	3000
Jack	15	600	6000
Billy	40	4000	1500
Kitty	32	6800	400

Develop an application to provide for three sort methods on sales persons based on the natural ordering of their age, name and wage. Display the sorted lists in the following manner:

```
Before Sorting :
John    :25     :1500   :2300
Mary    :18     :3000   :3000
Jack    :15     :600    :6000
Billy   :40     :4000   :1500
Kitty   :32     :6800   :400
```

```
Sort by Age :
Jack    :15     :600    :6000
Mary    :18     :3000   :3000
John    :25     :1500   :2300
Kitty   :32     :6800   :400
Billy   :40     :4000   :1500

Sort by Name :
Billy   :40     :4000   :1500
Jack    :15     :600    :6000
John    :25     :1500   :2300
Kitty   :32     :6800   :400
Mary    :18     :3000   :3000

Sort by Wage (BasicSalary + Commission) :
John    :25     :1500   :2300
Billy   :40     :4000   :1500
Mary    :18     :3000   :3000
Jack    :15     :600    :6000
Kitty   :32     :6800   :400
```

6.5.2 SalesPerson and Employee Class

We begin with the definition of a SalesPerson class as a subclass of an abstract class Employee (see Code 6.3). Class Employee is defined as an abstract class (Line 1) because no object could be instantiated from Employee. SalesPerson class inherits from Employee three attributes (name, age, and basicSalary) and three methods (getName(), getAge(), and getBasicSalary()).

Code 6.3: Employee Class

```
abstract class Employee {
  private String name;
  private int    age;
  private int    basicSalary;

  public Employee() {}
  public Employee (String name, int age, int basicSalary) {
    this.name        = name;
    this.age         = age;
    this.basicSalary = basicSalary;
  }

  public String getName() { return this.name; }
  public int getAge() { return this.age; }
  public int getBasicSalary() { return this.basicSalary; }
}
```

Code 6.4: A Comparable SalesPerson Class

```
class SalesPerson extends Employee implements Comparable {
  private int commission;
  public SalesPerson() {}
  public SalesPerson (String name, int age,
                      int basicSalary, int commission) {
    super(name, age, basicSalary);
    this.commission = commission;
  }

  public int getCommission() { return this.commission; }

  public int compareTo(Object o) {
    if (this.getAge() < ((SalesPerson) o).getAge())
      return -1;
    else if (this.getAge() > ((SalesPerson) o).getAge())
```

```
      return 1;
    return 0;
  }
}
```

To sort the list of five sales persons, we need to have a means for comparing SalesPerson objects and determining their natural ordering based on some comparative criteria. To make a SalesPerson comparable, we extend the SalesPerson class by implementing the Comparable interface highlighted in Table 6.1 (see Code 6.4). Hence, a SalesPerson object is not only an Employee object but also a comparable SalesPerson object .

To implement the Comparable interface, SalesPerson has to provide the implementation for compareTo() method. Our criteria for comparing two SalesPerson objects in the compareTo() method will be based on the age attribute.

6.5.3 Sort by Age: The main() Method 1

InterfaceExample class (Code 6.5) is the user class and it includes the main() method. We begin by instantiating the five SalesPerson objects and storing them in the array personArr[]. Their values are printed before the sort begins. The objects are sorted via the Arrays.sort() method which sorts the specified array of objects in ascending order. The Arrays Class is in the java.util package, hence the import java.util.Arrays; statement. A requirement of the Arrays.sort() method is that all elements in the array must implement the Comparable interface. This requirement is satisfied since all SalesPerson objects implement the Comparable interface.

Code 6.5: InterfaceExample Class

```
import java.util.Arrays;
class InterfaceExample {
  public static void main(String[] args) {
    SalesPerson personArr [] = new SalesPerson[5];
    personArr[0] = new SalesPerson("John" , 25, 1500, 2300);
    personArr[1] = new SalesPerson("Mary" , 18, 3000, 3000);
    personArr[2] = new SalesPerson("Jack" , 15, 600, 6000);
    personArr[3] = new SalesPerson("Billy", 40, 4000, 1500);
    personArr[4] = new SalesPerson("Kitty" , 32, 6800, 400);

    System.out.println("Before Sorting : ");
    for (int i = 0; i < personArr.length; i++) {
      System.out.println(personArr[i].getName() + "\t:" + personArr[i].getAge() +
                         "\t:" + personArr[i].getBasicSalary() +
                         "\t:" + personArr[i].getCommission());
    }
    System.out.println("\nSort by Age : ");
    Arrays.sort(personArr);

    for (int i = 0; i < personArr.length; i++) {
      System.out.println(personArr[i].getName() + "\t:" + personArr[i].getAge() +
                         "\t:" + personArr[i].getBasicSalary() +
                         "\t:" + personArr[i].getCommission());
    }

    System.out.println("\nSort by Name : ");
    Arrays.sort(personArr, new SortPersonByName());

    for (int i = 0; i < personArr.length; i++) {
      System.out.println(personArr[i].getName() + "\t:" + personArr[i].getAge() +
                         "\t:" + personArr[i].getBasicSalary() +
                         "\t:" + personArr[i].getCommission());
    }

    System.out.println("\nSort by Wage (BasicSalary + Commission) : ");
    Arrays.sort(personArr, new SortPersonByWage());
```

```
    for (int i = 0; i < personArr.length; i++) {
      System.out.println(personArr[i].getName() + "\t:" + personArr[i].getAge() +
                         "\t:" + personArr[i].getBasicSalary() +
                         "\t:" + personArr[i].getCommission());
    }
  }
}
```

6.5.4 Sort by Name: The main() Method 2

The next part of the application requires the SalesPerson objects to be sorted by their names. We have already implemented the Comparable interface to sort SalesPerson objects by their age. Since there is only one compareTo() method that can be implemented, we have a problem in implementing sort based on name and wage.

To solve this problem, we define a comparator that is capable of comparing two objects and determine their natural order based on some criteria. To achieve this, we define SortPersonByName as a class that implements the Comparator interface (see Code 6.6). To implement the Comparator interface requires SortPersonByName class to implement the compare() method. SortPersonByName class is thus a comparator by definition. Its compare() method determines the order of objects by examining the names of the two objects o1 and o2.

Code 6.6: SortPersonByName Class

```
import java.util.Comparator;
class SortPersonByName implements Comparator {
  public int compare(Object o1, Object o2) {
    SalesPerson p1 = (SalesPerson) o1;
    SalesPerson p2 = (SalesPerson) o2;
    return p1.getName().toLowerCase().compareTo(p2.getName().toLowerCase());
  }
}
```

The SortPersonByName comparator is passed in as a parameter to the Arrays.sort() method as shown in Code 6.5. The sort() method sorts the specified array of SalesPerson objects according to the order induced by the specified SortPersonByName comparator. In this case, the objects are ordered by names.

Earlier, we mentioned that the implementing class must implement all the methods in an interface. In Code 6.6, only the compare() method of the Comparator interface was implemented but not the equals() method as stated in Table 6.1. Why is this acceptable? The equals() method is implemented by SortPersonByName class implicitly since by default (Note: All classes including user defined classes, by default, are subclasses of the Object class.) , SortPersonByName class is a subclass of the Object class (the "mother" of all classes). A check on the Object class API reveals that this class belongs to the java.lang package and it has

```
boolean equals(Object obj)
```

as one of its methods. As a subclass of the Object class, SortPersonByName inherits this method. Therefore, the equals() method of the Comparator interface is actually implemented by the SortPersonByName class implicitly.

6.5.5 Sort by Wage: The main() Method 3

In like manner, we define the SortPersonByWage comparator to sort the SalesPerson objects in wage order. The wage of a SalesPerson is the sum of basic salary and commission. Code for SortPersonByWage comparator is given in Code 6.7. SortPersonByWage is used in Arrays.sort() method as shown in Code 6.5.

Code 6.7: SortPersonByWage Class

```
import java.util.Comparator;
class SortPersonByWage implements Comparator {
  public int compare(Object o1, Object o2) {
    SalesPerson p1 = (SalesPerson) o1;
    SalesPerson p2 = (SalesPerson) o2;
    int wage1 = p1.getBasicSalary() + p1.getCommission();
    int wage2 = p2.getBasicSalary() + p2.getCommission();
    return ((wage1 < wage2) ? -1 : (wage1 > wage2) ? 1 : 0);
  }
}
```

6.5.6 The Output

When the InterfaceExample application is run, the following output is produced on the console:

```
Before Sorting :
John    :25 :1500   :2300
Mary    :18 :3000   :3000
Jack    :15 :600    :6000
Billy   :40 :4000   :1500
Kitty   :32 :6800   :400

Sort by Age :
Jack    :15 :600    :6000
Mary    :18 :3000   :3000
John    :25 :1500   :2300
Kitty   :32 :6800   :400
Billy   :40 :4000   :1500

Sort by Name :
Billy   :40 :4000   :1500
Jack    :15 :600    :6000
John    :25 :1500   :2300
Kitty   :32 :6800   :400
Mary    :18 :3000   :3000

Sort by Wage (BasicSalary + Commission) :
John    :25 :1500   :2300
Billy   :40 :4000   :1500
Mary    :18 :3000   :3000
Jack    :15 :600    :6000
Kitty   :32 :6800   :400
```

6.6 The Serializable Interface

The five SalesPerson objects in the previous Employee example only exist during program execution. When the program ends, the objects are destroyed. Whatever state the objects might be in is lost forever.

One way to retain the values of objects beyond program execution is to store the state of the objects into secondary storage (such as a hard-disk). Java provides a facility for this purpose. It is known as serialization.

Classes are serializable if they implement the java.io.Serializable interface which is available in the java.io package. The Serializable interface does not have any methods or data fields. By implementing the java.io.Serializable interface, a class is identifying to the Java Virtual Machine that it is serializable. All subclasses of a serializable class are themselves serializable.

6.6.1 Demonstrating the use of Serialization Interface

To demonstrate the concept of serialization, we will serialize the SalesPerson Class (see Code 6.4). SerializableExample is the user class (see Code 6.8). There are 6 steps in this code:

1. Create objects
2. Print the state of the created objects before serialization
3. Serialize objects
4. Initialize array
5. Read objects into array
6. Print the state of objects

Code 6.8: SerializableExample Class

```
import java.io.*;
class SerializableExample {
  public static void main(String[] args) {
    SerSalesPerson personArr [] = new SerSalesPerson[5];

    System.out.println("\nCreating objects ...");
    // create five objects and store their references in array
    personArr[0] = new SerSalesPerson("John" , 25, 1500, 2300);
    personArr[1] = new SerSalesPerson("Mary" , 18, 3000, 3000);
    personArr[2] = new SerSalesPerson("Jack" , 15, 600, 6000);
    personArr[3] = new SerSalesPerson("Billy", 40, 4000, 1500);
    personArr[4] = new SerSalesPerson("Kitty"  , 32, 6800, 400);

    System.out.println("\nPrinting objects before serialization ...");
    // print the objects' values
    for (int i = 0; i < personArr.length; i++) {
      System.out.println(personArr[i].getName() + "\t:" +
                         personArr[i].getAge() +
                          "\t:" + personArr[i].getBasicSalary() +
                          "\t:" + personArr[i].getCommission());
    }

    System.out.println("\nSerializing objects ...");
    // serialize the SerSalesPerson objects into file "record.dat"
    try {
      ObjectOutputStream objOut = new ObjectOutputStream(
                                  new BufferedOutputStream(
                                  new FileOutputStream("record.dat")));
      for (int i = 0; i < personArr.length; i++) {
        objOut.writeObject(personArr[i]);
      }
      objOut.close();
    } catch (NotSerializableException e) {
        System.err.println(e);
    } catch (IOException e) {
        System.err.println(e);
    }

    System.out.println("\nInitializing array ...");
    // initializes array
    for (int i = 0; i < personArr.length; i++) {
      personArr[i] = null;
    }

    for (int i = 0; i < personArr.length; i++) {
      System.out.println("personArr[" + i + "] = " + personArr[i]);
    }


    System.out.println("\nReading objects into array ...");
    // read the objects from file and assign them into array
    try {
```

```
      ObjectInputStream objIn = new ObjectInputStream(
                                new BufferedInputStream(
                                new FileInputStream("record.dat")));
      for (int i = 0; i < personArr.length; i++) {
        personArr[i] = (SerSalesPerson) objIn.readObject();
      }
      objIn.close();
    } catch (ClassNotFoundException e) {
        System.err.println(e);
    } catch (IOException e) {
        System.err.println(e);
    }

    System.out.println("\nPrinting objects ...");
    // print the objects' values
    for (int i = 0; i < personArr.length; i++) {
      System.out.println(personArr[i].getName() + "\t:" +
                         personArr[i].getAge() +
                          "\t:" + personArr[i].getBasicSalary() +
                          "\t:" + personArr[i].getCommission());
    }
  }
}
```

Code 6.9: SerEmployee Class

```
import java.io.Serializable;
abstract class SerEmployee implements Serializable  {
  private String name;
  private int    age;
  private int    basicSalary;

  public SerEmployee() {}
  public SerEmployee (String name, int age, int basicSalary) {
    this.name        = name;
    this.age         = age;
    this.basicSalary = basicSalary;
  }

  public String getName() { return this.name; }
  public int getAge() { return this.age; }
  public int getBasicSalary() { return this.basicSalary; }
}
```

Code 6.10: SerSalesPerson Class

```
class SerSalesPerson extends SerEmployee
                     implements Comparable {
  private int commission;

  public SerSalesPerson() {}
  public SerSalesPerson (String name, int age,
                         int basicSalary, int commission) {
    super(name, age, basicSalary);
    this.commission = commission;
  }

  public int getCommission() { return this.commission; }
  public int compareTo(Object o) {
    if (this.getAge() < ((SerSalesPerson) o).getAge())
      return -1;
    else if (this.getAge() > ((SerSalesPerson) o).getAge())
      return 1;
    return 0;
  }
}
```

The serializable Employee class has been named SerEmployee to avoid a clash with the Employee class used in the previous example. SerEmployee is given in Code 6.9. Similarly, the serializable SalesPerson class has been named SerSalesPerson to avoid a clash with the SalesPerson class used in the previous example. SerSalesPerson is given in Code 6.10.

The output from the code is as follows:

```
Creating objects ...

Printing objects before serialization ...
John    :25 :1500   :2300
Mary    :18 :3000   :3000
Jack    :15 :600    :6000
Billy   :40 :4000   :1500
Kitty   :32 :6800   :400

Serializing objects ...

Initializing array ...
personArr[0] = null
personArr[1] = null
personArr[2] = null
personArr[3] = null
personArr[4] = null

Reading objects into array ...

Printing objects ...
John    :25 :1500   :2300
Mary    :18 :3000   :3000
Jack    :15 :600    :6000
Billy   :40 :4000   :1500
Kitty   :32 :6800   :400
```

The serialized objects have been written into a file named "record.dat". It contains the state of the five objects.

The abstract class SerEmployee implements the interface Serializable. Therefore, it is a serializable class. All subclasses of Employee class are thus serializable. So, we now have a SerSalesPerson that is comparable and serializable.

From the above, we can identify one advantage of Interface, that is, Interface can be used to incrementally extend the functionalities of classes.

6.7 Interface and Abstract Class

Abstract class was introduced earlier in Chapter 5. One salient characteristic of an abstract class is that no object can be instantiated from it. Likewise, we do not instantiate objects from an interface.

Much like an abstract class, an interface is made up of a set of abstract methods. Can we therefore not replace interface with abstract class in our example above? The answer is "No". While interface is similar to an abstract class, the two are not the same.

Suppose we define Comparable as an abstract class, then instances of SalesPerson Class would have to inherit from Employee and Comparable classes and this would contribute to a Multiple Class Inheritance situation – a situation not permissible in Java. This explains why we need to define Comparable as an interface.

6.8 Changes in Interface

One limitation of interface is that once it is defined and implemented by classes, the interface cannot be changed. Making changes to an interface will break all implementing classes since the implementing classes no longer implement the new interface. An interface therefore cannot grow when new behaviour has to be added.

From the above, it is essential for an interface to be considered and specified fully in the beginning. However, this is not always possible. Maintenance on old implementing classes or creating new interfaces is thus inevitable.

6.9 Uses of Interface

Interfaces are meant for defining a set of pre-determined behaviour that is to be implemented by any class in the class hierarchy. Uses of interfaces include:

1. Capturing behaviour without forcing class relationship
2. Providing a place for defining methods that classes are expected to implement
3. Provides a front for objects to publish the constant and method definitions without revealing their classes.