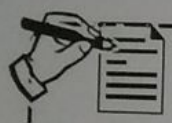
Inheritance and Polymorphism

8.1 Inheritance Basics

A child inherits the property of his parent. He can acquire new properties or modify the inherited one. Similarly, inheritance is a feature of OOP, which allows making use of the existing class without making changes to it. This can be achieved by deriving a new class (Derived class) from the existing class (Base class). The derived class inherits all the properties of the base class. More properties can be added to the derived class, if needed. Therefore, the complexity of the derived class may grow as the level of inheritance grow. There is no limit to the level of inheritance.



Note :

The mechanism of deriving a new class from an old one is called **inheritance**. The old class is called the **base class** or **super class** or **parent class** and the new one is called the **subclass** or **derived class** or **child class**.

For example, scooter is a class in itself. It is also a member of two wheeler class. Two wheelers class in turn is a member of an automotive class as shown in figure 8.1.

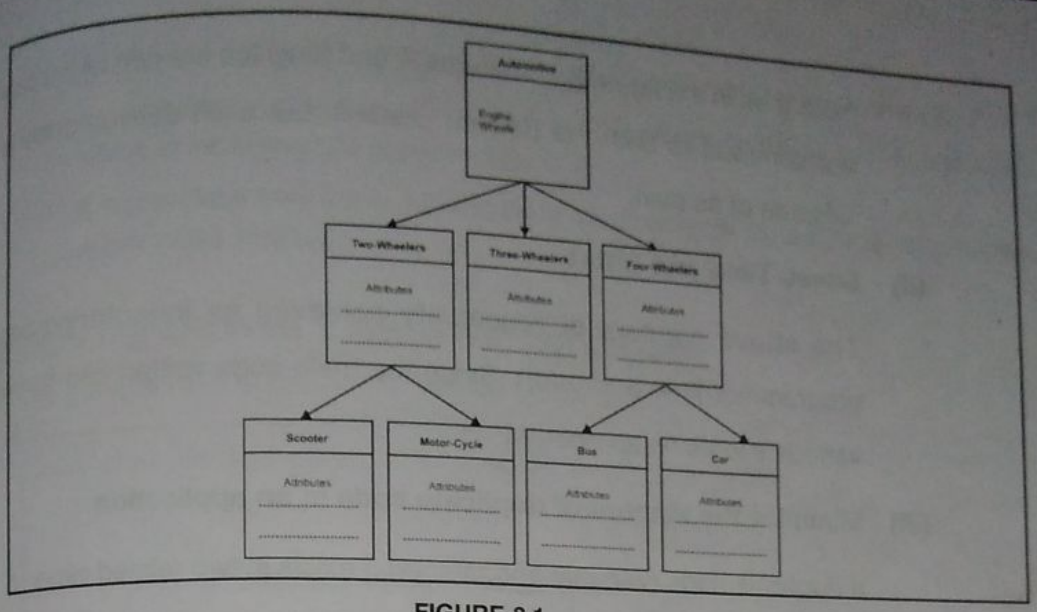


FIGURE 8.1

8.1.1 Benefits of Inheritance

Main benefits of inheritance are :

(i) **Reusability :** Main advantages of inheritance is **reusability.** Reusability means that we can add additional features of an existing class without modifying it as shown in figure 8.2.

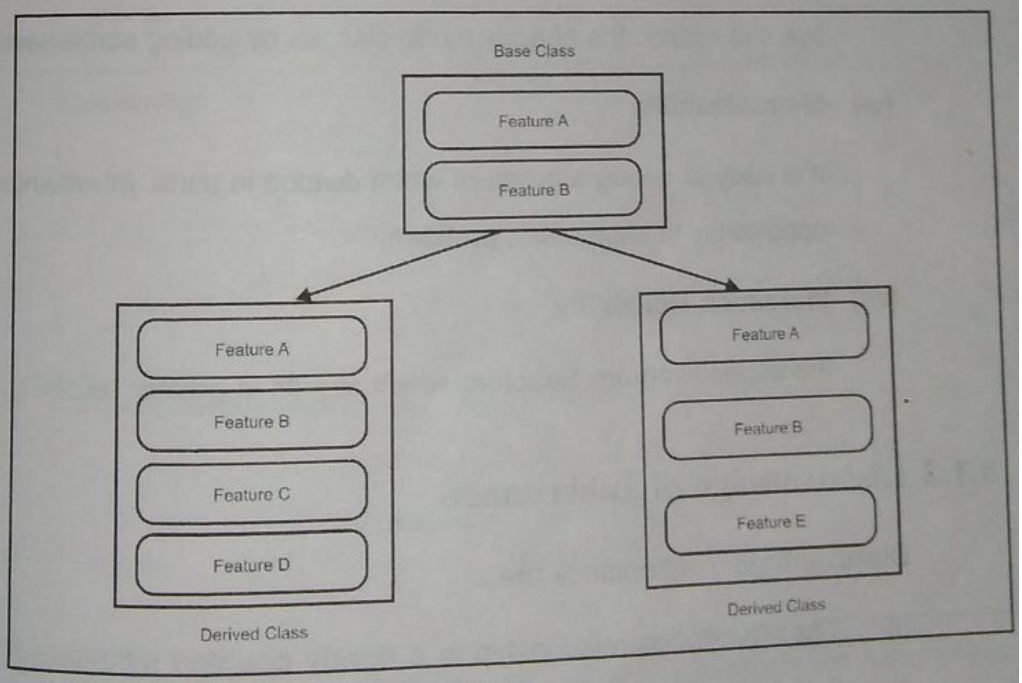


FIGURE 8.2

Note than in the figure 8.2, features A and B, which are part of the base cl
are common to both the derived classes, but each derived class also
features of its own.

(ii) Saves Time and Effort

The above concept of reusability achieved by inheritance saves
programmer time and effort. Since the main code written can be reus
various situations as needed.

(iii) Minimize the amount of duplicate code in an application

If duplicate code (variable and methods) exists in two related classes, w
refactored that hierarchy by moving that common code up to the co
superclass.

(iv) Better organization of code

Moving of common code to superclass results in better organization of

(v) Extendability

We can extend the already made classes by adding some new feature

(vi) Maintainability

It is easy to debug a program when divided in parts. Inheritance provi
opportunity to capture the problem.

(vii) Increased Reliability

Increases Program Structure which results in greater reliability.

8.1.2 Disadvantage of Inheritance

Disadvantage of inheritance are :

(i) The inheritance relationship is a **tightly coupled** relationship, ther
tight bounding between parent and child.

(ii) As classes are tightly coupled, they cannot work independently of each other.

(iii) If we change code of parent class it will get affects to the all the child classes which is inheriting the parent code.

(iv) If super class method is deleted code may not work as subclass may call the super class method.

8.2 Defining Derived Classes (or Subclasses)

The general form of defining a derived class is

```
class derived_class_name extends base_class_name
{
        ...
        ... // members of derived class
        ...
}
```

The keyword **extends** signifies that the properties of the derived class are extended to the base class. Therefore, now the derived class contains all its own members as well as those of the base class.

The definition of derived class is similar to a normal class definition except for the use of extends keyword and base class name.

Example 8.1

```
class student
{
        .....
        .....    // members of base class
        .....
}

class test extends student
{
        .....
        .....    // members of derived class
        .....
}
```

In this example, **test** is the name of derived class and **student** is the name of base class. A derived class gains all the non-private members of its base class.

```
class inheritance
{
        public static void main (String arg [])
        {
                derived d = new derived ( );
                // Accessing base class methods using derived class object
                d.get_ab (10, 20);
                d.add2 ( );

                // Accessing derived class methods
                d.get_c (30);
                d.add3( );
        }
}
```

Output :

```
a = 10
b = 20
Sum of two numbers = 30
a = 10
b = 20
c = 30
Sum of three numbers = 60
```

In derived class, we have not declared variables **a** and **b**, but we have used them in derived class methods. This is possible just because of inheritance. Also it is clear from this example that derived class object can access base class methods as well as derived class methods.

8.3 Access Control

The access control determines how a member can be accessed. The access specifiers provided by java are **public, private, friendly** (or default) and **protected**.

(a) public Access

A variable or method declared as **public** has the widest possible visibility and accessible everywhere. Public member of a class is visible or accessible to its own class, its derived classes and outside the class.

public variable or method can be declared as :

```
public int x ;
public void sum ( ) {.....}
```

(b) friendly Access

If we did not specify the access control modifier, the member has accessibility known as **friendly level of access**. The difference between friendly access and public access is that the public modifier makes the member visible to all classes regardless of their package while friendly access makes members visible only inside the package. (A package is a group of related classes stored separately.)

(c) protected Access

protected modifier makes the members accessible not only to classes and subclasses in the same package but also to subclasses in other packages. Hence we can say the access level lies between public and friendly access.

(d) private

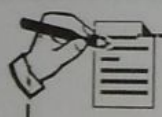
The private members of a class can be accessed only within their own class. They cannot be inherited by subclasses and are therefore inaccessible in subclasses.

(e) private protected Access

In Java, we can use two access control modifiers as

```
private protected int number ;
```

The access level provided by such modification is between protected and private access.



Note :

Derived classes do not access to the private member of base classes.

What do we do if the private data needs to be inherited by a derived class? One way to solve this problem is by making the member **public**. Of course, making the member public also makes it available to all other code, which may not be desirable. Fortunately, Java allows you to create a **protected member** in this case.

A protected member is created using the **protected** access modifier. A protected member of the base class becomes a protected member of the derived class and is, therefore, accessible by the derived class. The Table 8.1 shows the visibility provided by various access modifiers.

TABLE 8.1

Access Modifier → / Access Location ↓	public	protected	friendly (default)	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same packages	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

8.4 Types of Inheritance

Java supports three types of inheritance :

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance

Java does not support **multiple inheritance** directly because multiple inheritance supports multiple base classes and in java, the classes cannot be inherited from more than one class. However, you can implement multiple inheritance through **interfaces.**

8.4.1 Single Inheritance

In single in heritance, there is only one base class and one derived class as shown in the figure 8.3.

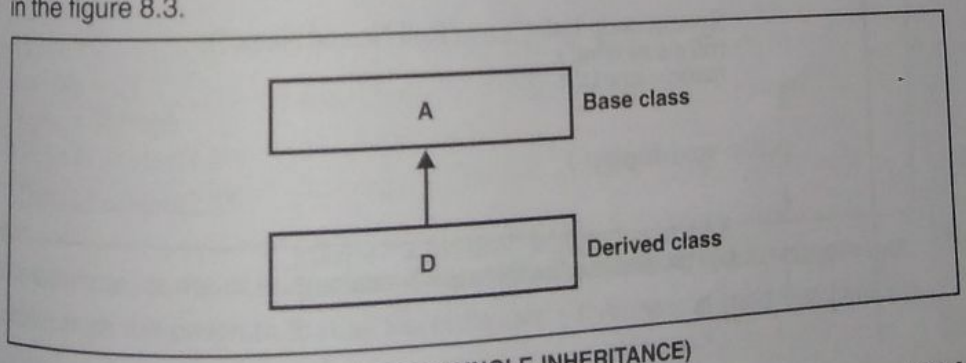


FIGURE 8.3 (SINGLE INHERITANCE)

Example 8.3

Figure 8.4 and following program illustrate single inheritance.

FIGURE 8.4

```
import java.util.Scanner;

//Base class
class student
{
      int roll;
      String name;

      //Create Scanner Object
      Scanner s=new Scanner(System.in);

      public void getdata( )

      {
            System.out.println("Enter Roll No and Name");
            roll = s.nextInt();
            name = s.next();
      }

      public  void display( )

      {
            System.out.println("Roll No = " + roll);
            System.out.println("Name = " + name);
      }
}
```

```
// Derived class
class test extends student

{
    int sub1, sub2;
    public void readdata( )

    {
            getdata( );                    //base class method
            System.out.println("Enter the Marks of two subjects");
            sub1 = s.nextInt();
            sub2 = s.nextInt();
    }

    public void putdata( )

    {
            display( );                    //base class method
            System.out.println("Marks of subject1 " + sub1);
            System.out.println("Marks of subject2 " + sub2);
    }
}

class singleinheritance

{
    public static void main(String args[])

    {
            test obj = new test( );
            obj.readdata( );
            obj.putdata( );
    }
}
```

Output :

```
Enter Roll No and Name
21
Shivank
Enter the Marks of two subjects
80
85
Roll No = 21
Name = Shivank
Marks of subject1 80
Marks of subject2 85
```

In this example, name of base class is **student** and name of derived class is **test.** It is clear from the program that all the non-private members of base class are the members of derived class.

8.4.2 Multilevel Inheritance

In multilevel inheritance, a class is derived from an already derived class as shown in figure 8.5.

FIGURE 8.5

The class A is a base class for the derived class B. Class B is a base class for the derived class C. Therefore, the class B is known as **intermediate base class.**

Example 8.4

Figure 8.6 and following program illustrate multilevel inheritance.

FIGURE 8.6

Jul 15, 2021

```
import java.util.Scanner;

//Base class
class student
{
    int roll;
    String name;

    //Create Scanner Object
    Scanner s=new Scanner(System.in);

    public void getdata( )

    {
        System.out.println("Enter Roll No and Name");
        roll = s.nextInt();
        name = s.next();
    }

    public void display( )

    {
        System.out.println("Roll No = " + roll);
        System.out.println("Name = " + name);
    }
}

// Intermediate base class
class test extends student

{
    protected int sub1, sub2; //private will not work
    public void readdata( )

    {
        getdata( );          //base class method
        System.out.println("Enter the Marks of two subjects");
        sub1 = s.nextInt();
        sub2 = s.nextInt();
    }

    public void putdata( )

    {
        display( );          //base class method
        System.out.println("Marks of subject1 " + sub1);
        System.out.println("Marks of subject2 " + sub2);
    }
}
```

```
//Derived class
class total extends test

{
    int total_marks;
    public void compute()

    {
        total_marks = sub1 + sub2;
        System.out.println("Sum of Marks = " + total_marks);
    }
}

class multilevelinheritance

{
    public static void main(String args[])

    {
        total obj = new total( );
        obj.readdata( );
        obj.putdata( );
        obj.compute( );
    }
}
```

Output :

```
Enter Roll No and Name
21
Shivank
Enter the Marks of two subjects
80
85
Roll No =21
Name = Shivank
Marks of Subject1 80
Marks of Subject2 85
Sum of Marks = 165
```

8.4.3 Hierarchical Inheritance

In hierarchical inheritance, two or more classes are derived from one base class as shown in figure 8.7.

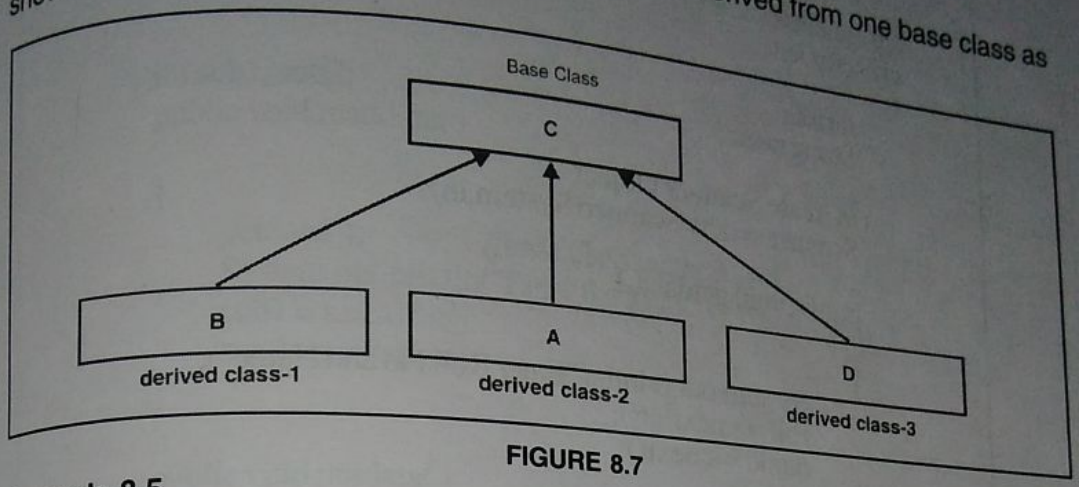


FIGURE 8.7

Example 8.5

Figure 8.8 and following program illustrate the hierarchical inheritance.

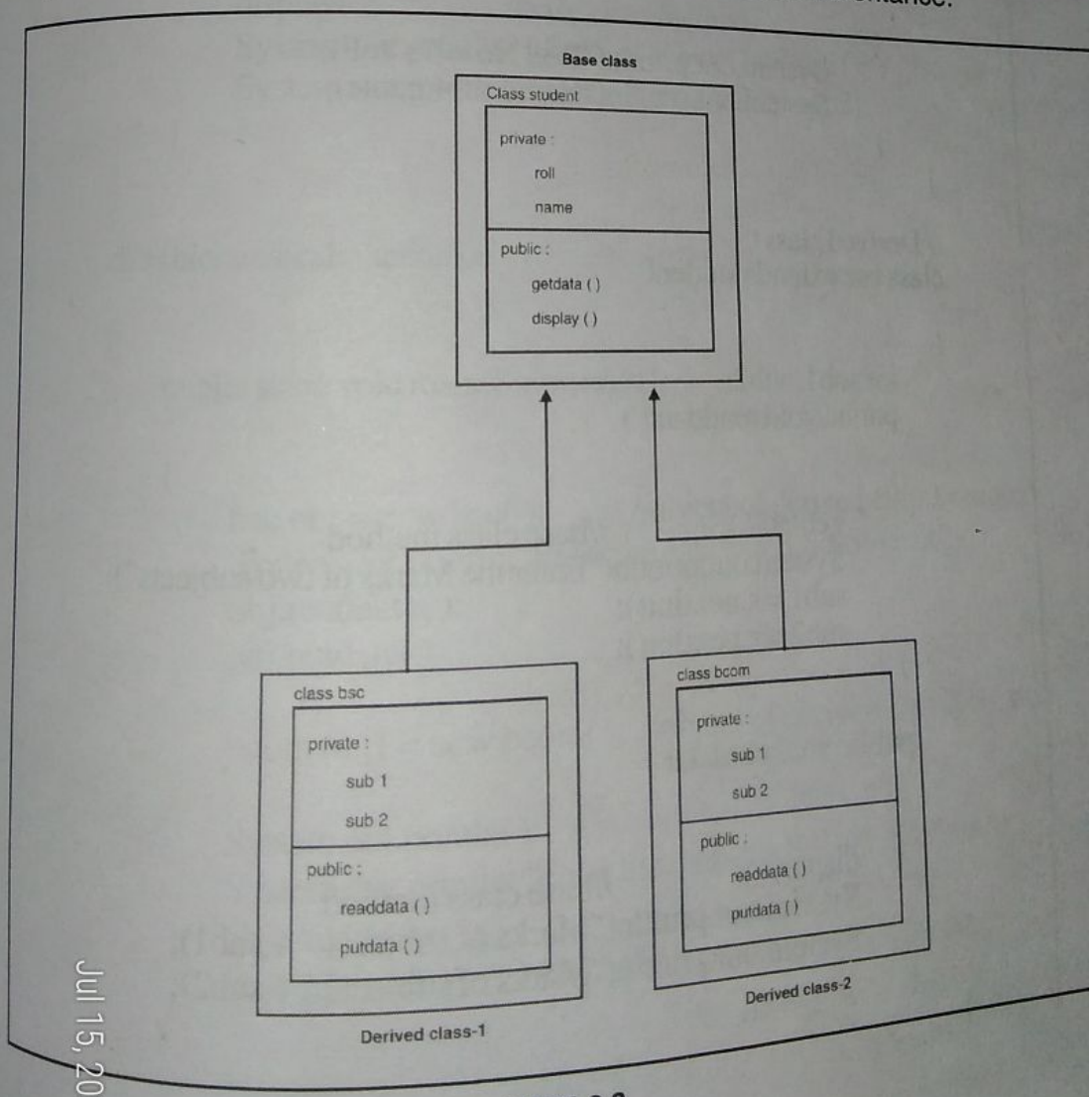


FIGURE 8.8

```
class bike extends vehicle
{
  void condition( )
  {
    System.out.println("Good condition ");
  }
}

class finalmethoddemo
{
  public static void main(String args[ ])
  {
    bike obj = new bike( );
    obj.condition( );
  }
}
```

Output :

Compile time error

.10 Polymorphism

Polymorphism refers to the ability of one thing to take many (Poly) distinct forms (Morphism). Types of Polymorphism is shown in figure 8.9.

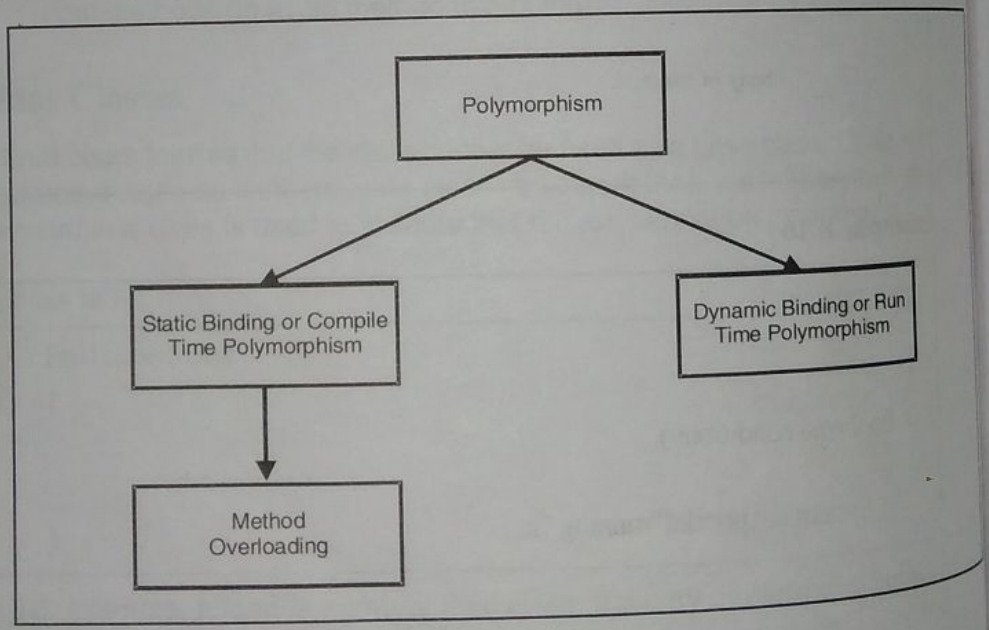


FIGURE 8.9

Binding means connecting the method call to the code to be executed in response to the call. These are of two types.

1. **Static Binding or Compile Time Polymorphism**

   Static Binding means that the code associated with the method call is linked at compile time. Static Binding is also known as **early binding** or compile time polymorphism. Method overloading is an example of static binding.

   For example, when an **overloaded method** is called, the compiler matches the arguments passed with the format arguments (or signature) of the various methods with the same name.

   Once the match is found, it associates that code of the method with the call. Therefore, there is no confusion and appropriate method is linked at compile time. This is called static binding.

2. **Dynamic Binding or Run Time Polymorphism**

   Dynamic binding means that the code associated with the method call is linked at run time. Dynamic binding is also known as **late binding** or run time polymorphism.

   For example, consider the following figure 8.10.

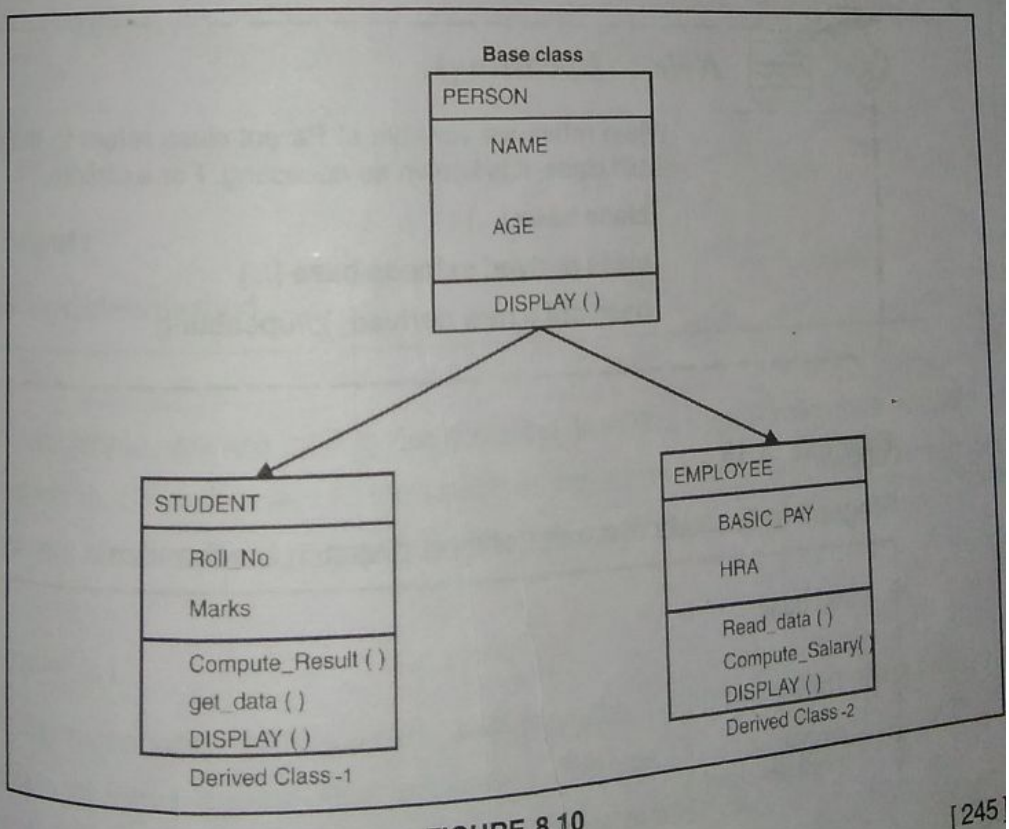


FIGURE 8.10

In this example, there is one base class, namely PERSON and two derived cla... namely STUDENT and EMPLOYEE. A exactly same method DISPLAY( ) without a... argument is available in base class as well as in both derived class. If, we call t... method through the same base class reference, then compiler will get confused ... to which method to refer to for each call. So, compiler does not attach the metho... any call and leaves the job to the run-time-system to decide the method to be calle... So calling of the appropriate method takes place at run-time and hence it is call... **dynamic binding**.

Overridden methods allow Java to support runtime polymorphism (or dynam... binding).

8.10.1 Runtime Polymorphism or Dynamic Method Dispatch

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call t... an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Note : (Upcasting)

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example,

```
class base {...}
class derived extends base {...}
base obj = new derived( );//upcasting
```

Example 8.16

Program to illustrate dynamic method dispatch mechanism.

```
class base

{
  void display( )
```

```
  {
    System.out.println("Base class Method ");
  }
}

class derived extends base
{
  void display( )
  {
    System.out.println("Derived class Method ");
  }
}

class runtimepoly
{
  public static void main(String args[])
  {
    base obj = new derived( );// upcasting
    obj.display( );
  }
}
```

Output :

Derived class method

In this example, we are calling the **display( )**method by the reference variable parent class. Since it refers to the subclass object and sub class method overrid the parent class method, therefore, subclass method is invoked at runtime.

Example 8.17

Consider a scenario, Bank is a class that provides method to get the rate of inter But rate of interest may differ according to banks. For example, SBI, PNB AXIS banks could provide 8%, 9% and 7% rate of interest.

Interfaces

1 Introduction

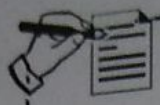
Java does not support **multiple inheritance**. But, a large number of real-life applications require the use of multiple inheritance. Therefore, java provides an alternate approach known as **interfaces** to support the concept of multiple inheritance.

The interface in java is also a mechanism to achieve **fully abstraction**. Interface is a collection of abstract data members such as methods. All the members of an interface are implicitly **public** and **abstract**. The method defined in an interface do not have their implementation and only specify the parameters they will take and the type of values they will return. An interface is always **implemented in a class**. The class implementing an interface must define all the methods contained inside an interface, otherwise Java compiler given an error message. Interface in Java is equivalent to an abstract base class. You cannot instantitate an object through an interface, but you can offer a set of functionalities that is common to several different classes.

2 What is an Interface?

Interface looks like class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default **abstract** (only method signature, no body). Also, the variables declared in an interface are **public**, **static** & **final** by default.

Interfaces are used for **abstraction**. Since methods in interfaces does not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not support multiple inheritance, using interface we can achieve this as a class can implement more than one interfaces.



Note :

The java compiler adds automatically public and abstract keywords before the interface method and public, static and final keywords before data members.

9.3 Reasons to use Interface

There are three reasons to use interface.

1. It is used to achieve fully abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

9.4 Defining an Interface

Interfaces are syntactically similar to classes.

Syntax is :

```
Interface interfacename
{
    variables declaration;
    methods declaration;
}
```

Where

interface is keyword

interfacename is any valid java identifier.

Methods are declared using only their return type and signature. They are, abstr methods. We know now that in an interface, no method can have an implementat All **variables declared** in the interface definition are **constant**, therefore, they sho be initialized with a **constant value**.

Example 9.1

```
Interface student
{
    int num = 40;
    void display( );
}
```

In this example, student is the name of interface. It contains one constant **num** having value equal to 40 and one method **display( )**. Interface variables are public, static and final by default and methods are public and abstract as shown in figure 9.1.

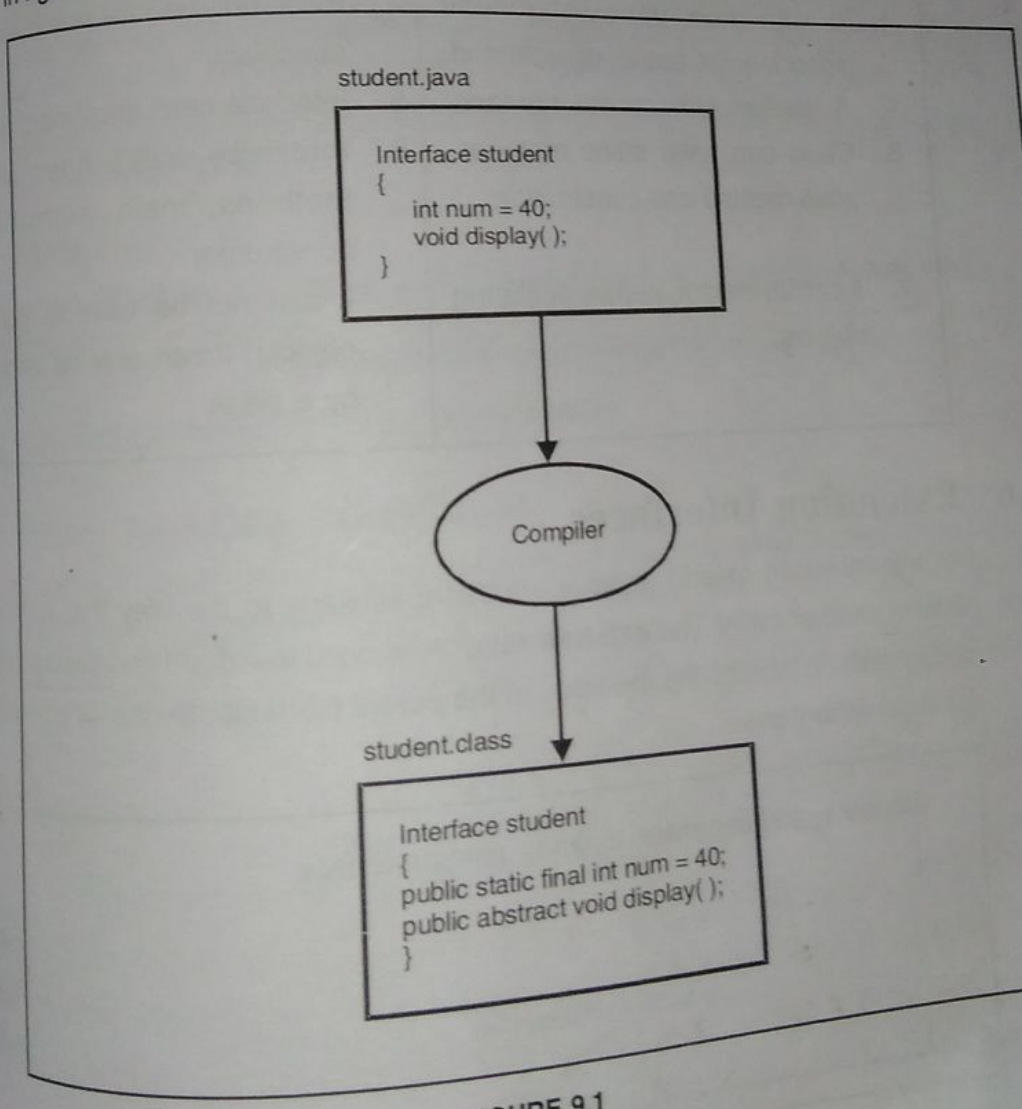


FIGURE 9.1

Differences between Class and Interface

Differences between class and interface are shown in table 9.1.

TABLE 9.1

Class	Interface
1. Class can have abstract and non-abstract methods.	1. Interface can have only abstract methods.
2. Class doesn't support multiple inheritace.	2. Interface supports multiple inheritance.
3. Class can have final, non-final, static and non-static variables.	3. Interface has only static and final variables.
4. It can use various access specifiers like public, or protected.	4. It can only use the public access specifier.
5. A class implements the interface.	5. Interface can't implement class.
6. Class can have static methods, main method and constructor.	6. Interface can't have static methods, main method and constructor.
7. It can be instantiated by declaring objects.	7. It can not be used to declare objects. It can only be inherited by a class.

9.7

5 Extending Interfaces

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface. Syntax is also-same as for class inheritance.

```
interface childinterface extends parentinterface
{
    ...
    ...          // body of child interface
}
```

Example 9.2

```
interface parent
{
    int roll = 1001;
    int marks = 85;
}
interface child extends parent
{
    void display( );
}
```

In this example, we have used two constants in **parent** interface and one method in **child** interface. The interface **child** would inherit all the constants into it. A java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, an interface can extend more than one parent interface.

The extends keyword is used once and the parent interfaces are declared in a comma-separated list.

Example 9.3

If the student interface extended both sports and test, it would be declared as :

interface student extends sports, test

Note that an interface cannot extend classes.

9.7 Implementing Interfaces

Interfaces can be used as parent classes or superclasses whose properties are inherited by classes. After an interface has been defined, one or more classes can implement it. When a class implements an interface, it must define (or implement) all the methods of the interface. A class uses the **implements** keyword to implement an interface.

Syntax is :

```
class classname implements interfacename
{
...
        // body of class
...
}
```

In this example, first and second interface have **same method** but its implementation is provided by cl... **test**, so there is no ambiguity.

At... ...an have same constant name.

```
...rface A
{
    int x=20;
}

interface B
{
    int x=500;
}

class test implements A, B
{
    public static void main(String args[])

    {
        System.out.println(x);    // reference to x is ambiguous both variables are x
        System.out.println(A.x); // valid
        System.out.println(B.x); // valid
    }
}
```

3 Marker or tagged Interface

An interface that have no member is known as marker or tagged interface. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

Example 9.9

```
public interface Serializable
{
}
```


.9 Nested Interface

An interface declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by outer interface or class. It can be accessed directly.

The nested interface must be **public** if it is declared inside the interface and it can have any access modifier if declared within the class.

Example 9.10

```
interface first
{
    void display( );
    interface second
    {
        void msg( );

    }
}

class nested1 implements first.second
{
    public void msg( )
    {
        System.out.println("Example of Nested Interface");
    }

    public static void main(String args[])
    {
        first.second obj = new nested1( );        // upcasting
        obj.msg( );
    }
}
```

Output :

```
Example of Nested Interface
```

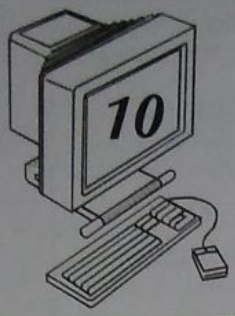
Note that, we are accessing the **second** interface by its outer interface **first** because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. Similarly, we can define and access an interface inside the class.

9.10 Interfaces verses Abstract Classes

An interface is similar to a class without instance and without method bodies. But abstract classes do allow static method definitions and interfaces don't. So an interface is like an abstract class that must be extended in exactly the manner that its abstract methods specify. The table 9.2 shows the differences between interfaces and abstract classes.

TABLE 9.2

Interface	Abstract Class
1. The **interface** keyword is used to declare a interface.	1. The **abstract** keyword is used to declare a abstract class.
2 Interface can have only abstract methods.	2. Abstract class can have abstract and non-abstract methods.
3. Interface supports multiple inheritance.	3. Abstract class doesn't support multiple inheritace.
4. Interface has only static and final variables.	4. Abstract class can have final, non-final, static and non-static variables.
5. Slow, it requires extra indirection to find corresponding method in the actual class.	5. Fast.
6. Interface can't implement class.	6. Abstract class implements the interface.
7. Interface can't have static methods, main method and constructors.	7. Abstract class can have static methods, main method and constructors.



Packages

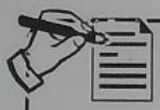
10.1 Introduction

A class is a single entity that contains the related members (data and methods) of same type of objects. If we write all the related classes, abstract classes individually, these classes will be in scattered format. Later if we want to use only these classes as a bundle of related classes, we can't use them easily.

Java environment provides a powerful means of grouping related classes and interfaces together in a single unit called **packages**. Java packages provide a convenient mechanism for managing a large group of classes and interfaces.

Classes defined within a package must be accessed through their package name. Therefore, a package provides a means to name a collection or set of classes. Each class name must be unique within a given package.

But, two classes residing in two different packages can have the same name.



Def. :

A java package is a group of similar types of classes, interfaces and sub-packages.

10.2 Advantages of Java Package

Advantage of packages are :

1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2. The classes contained in the packages can be reused.
3. Packages reduce problems with conflicts in names. It allows us to **hide** classes so that conflicts can be avoided.
4. Packages allow us to protect classes, variables and methods in larger ways than on a class-by-class basis.

10.3 Types of Packages

There are two types of packages in Java.

1. Java API packages
2. User defined packages

10.3.1 Java API Packages (or Built-in packages)

Java API packages mean the built in packages of Java. Java API (Application Programming Interface) provides a large number of classes grouped into different packages according to functionality. We can use these package classes in any of the Java program by importing them at the top of the program. The Java API is stored in packages. At the top of the package hierarchy is **Java**. Descending from **Java** are several subpackages as shown in figure 10.1.

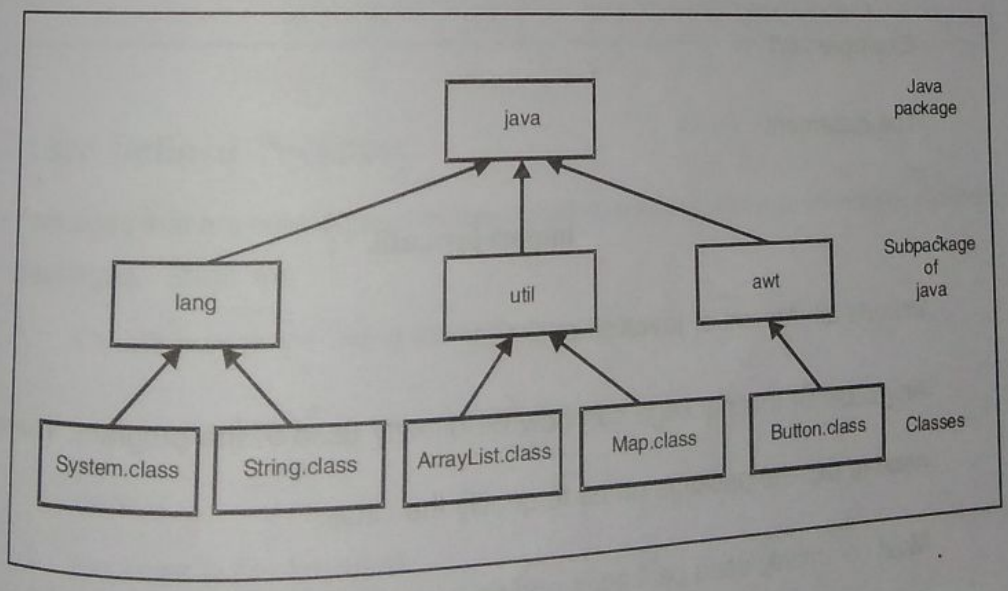


FIGURE 10.1

We have been using **java.lang** ever since the beginning of this book. It contains the **System** class, which we have been using while performing output using **println()**. The **java.lang** package is distinct and unique since it gets **automatically imported** into every Java program. However, other packages need to be explicitly imported into Java programs by using import statement. Syntax is :

```
Import packagename.classname; // import specify class only
or
import packagename.*;          //import all classes of the package
```

The first statement allows the specified class in the specified package to be imported.

The second statement imports every class contained in the specified package.

The **import statements** must appear at the top of the file, before any class declarations.

Example 10.1

The statement

import java.util.*;

imports all classes of **java.util** package.

Any class of this package can now be directly used in the program. There is no need to use the package name to qualify the class.

Most commonly used packages and their classes are given in table 10.1.

TABLE 10.1

Standard Package	Description
java.lang	Stores a large number of general purpose classes such as **System** class, **String** class and **Math** class.
java.io	Stores the Input/Output classes. For example, **DataInputStream** class.
java.util	Stores language utility classes such as vectors, date, has tables etc. For example, **Scanner** class.
java.net	Stores the classes which support networking.
java.awt	Stores classes which support the Abstract Window Toolkit. It contains classes for implementing graphical user interface.
java.applet	Stores classes for creating and implementing java applets.

10.3.2 User Defined Packages

Packages that are created and implemented by the user are known as **user defined packages**. Steps are :

1. Create a package using the following syntax :

package packagename;

Here

package is the keyword.

packagename is the name of package.

This statement must be first statement of the java program (except comment and white spaces.)

2. Create a directory of package name i.e. name of directory must be same as name of package.
3. Now, define the class (or classes) of the package and declare all classes as **public** (Note, a package can contain non-public classes but these classes become **hidden classes** of the package).
4. Store each class as **classname.java** in the package directory.
5. Compile the file. This creates **.class** file in the package directory.
6. Finally, use the public package members or classes outside the package. There are three ways to access public package members from outside the package.

   (i) **import package. * ;** // import an entire package.

   (ii) **import package.classname;** // import the package specify class

   (iii) **refer to the member by fully qualified name.**

Example 10.2

One example of fully qualified name is

```
double result = java . lang . Math . sqrt (x);
                package   class   method
                name      name    name
```

Note : **(Naming Conventions)**

(a) Generally, package names are written in lowercase letters.

(b) Class names and interface names are written in uppercase letters.

(c) Method names of a class are written in lowercase letters.

10.4 Defining A Package

To create a package we have to include a package command as the first statement in a Java source file. The classes declared after the statement in that file will belong to the specified package. **Syntax is**

package packagename ;

Here, **packagename** is the name of the package and **package** is the keyword.

Example 10.3

The statement

package mypkg ;

create a package called **mypkg**.

We can also create a hierarchy of packages, i.e. package within a package called **subpackage** for further classification and categorization of classes and interfaces. **Syntax is :**

package mypkg.mypkg2.mypkg3;

Here, **mypkg2** is a subpackage in package **mypkg**, **mypkg3** is a subpackage in subpackage **mypkg2**.

Example 10.4

The statement

package pck1.pck2.pck3;

creates a hierarchy of packages. The **pck2** is a subpackage in package **pck1** and **pck3** is a subpackage in subpackage **pck2**. To refer to a class of a nested package we can use a fully qualified name with all the containing package names prefixing the class name as shown in figure 10.2.

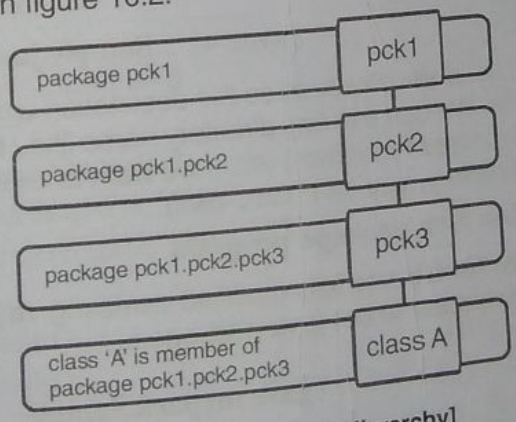


FIGURE 10.2 [Package Hierarchy]

e above package hierarchy **class A** is identified by the name **pck1.pck2.pck3.A** is called the fully qualified type name of the class.

10.5 Adding a Class to package

We can add a class to a package using the following syntax.

```
package    packagename;
public class classname
{
    ...
            // body of class
    ...
}
```

Example 10.5

Step1 : Write the following code

```
package    mypack;
public class Sample
{
        public static void main (String args[ ])
        {
                System.out.println("Welcome to Package");
        }
}
```

Step 2 : Save this file as **Sample.Java**

Step 3 : Compile java package using following syntax :

Javac -d directory javafilename

for example

Javac -d . Sample.java

The -d switch specifies the destination where to put the generated class file. We can use any directory name like / home (in case of Linux), d:/abc (in case of windows) etc. If we want to keep the package within the same directory, you can use . (dot).

Step 4 : Run java package program

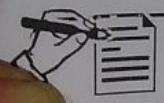
We need to use fully qualified name e.g. **mypack.Simple** to run the class. Command to run java package is :

java mypack.Sample

Output :

Welcome to package

```
// Second source File
package areapkg;
public class Circlearea
{
        public double area (double r)
        {
                return 3.14 * r * r ;
        }
}
```



Note :

Save this file as **Circlearea.java** and store it in the directory **areapkg** and compile it.

Similarly, we can add any number of classes in a package.

10.6 Using Package Members

These are three ways to access the package members from outside the package:

(a) import package.* ;

(b) import package.classname;

(c) fully qualified name

(a) Using package.*

If we use package.* then all the classes and interfaces of this package will be accessible but not subpackages. The import keyword is used to make the classes and interface of another package accessible to the current package.

Example 10.8

Let us use the classes of **areapkg** package discussed earlier.

```
import areapkg.*;
class computarea
{
    public static void main(String args [ ])
    {
            // Creating object of Circlearea class
            Circlearea obj = new Circlearea( );
            double carea = obj.area(3.5);
            System.out.println("Area of circle is " + carea);

            // Creating object of Rectarea class
            Rectarea obj1 = new Rectarea();
            double rarea = obj1.area(4.5, 8.5);
            System.out.println("Area of Rectangle is " + rarea);
    }
}
```

Output :

Area of circle is 38.465

Area of Rectangle is 38.25

Note :

Save the file Named as **computarea.java** outside the directory named **areapkg.** Finally, Compile and Run it.

(b) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

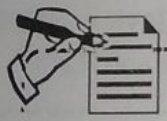
Example 10.9

```
import areapkg.Rectarea;
import areapkg.Circlearea;

class computarea1
{
    public static void main(String args [ ])
    {
            // Creating object of Circlearea class
            Circlearea obj = new Circlearea();
            double carea = obj.area(3.5);
            System.out.println("Area of circle is " + carea);

            // Creating object of Rectarea class
            Rectarea obj1 = new Rectarea();
            double rarea = obj1.area(4.5, 8.5);
            System.out.println("Area of Rectangle is " + rarea);
    }
}
```

Output :

Area of circle is 38.465

Area of Rectangle is 38.25



Note : Save the file Named as **computarea1** outside.java the directory named **areapkg.** Finally, Compile and Run it.

(c) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

Example 10.10

```
class computarea2
{
    public static void main(String args [ ])
    {
            // Creating object of Circlearea class
            areapkg.Circlearea obj = new areapkg.Circlearea();
            double carea = obj.area(3.5);
            System.out.println("Area of circle is " + carea);

            // Creating object of Rectarea class
            areapkg.Rectarea obj1 = new areapkg.Rectarea();
            double rarea = obj1.area(4.5, 8.5);
            System.out.println("Area of Rectangle is " + rarea);
    }
}
```

Output :

Area of circle is 38.465

Area of Rectangle is 38.25

Note :

Save the file Named as **computarea2.java** outside the directory named **areapkg.** Finally, Compile and Run it.

10.7 Accessibility of Packages

As we know that packages act as containers for classes and other packages. Also, classes act as containers for data and methods. Data members and methods can be declared with access modifiers such as **private, protected, public** and **default**. Meaning of these modifiers remain same as discussed in the chapter **inheritances.**

```
package mainpack.subpack;
class Sample
{
    public static void main(String args[])
    {
        System.out.println("Hello Subpackage");
    }
}
```

Output :

Hello Subpackage

Note :

Create a directory **mainpack.** Create a directory **subpack** in **mainpack** directory and save the file **Sample.java** in it and Compile it.

Now, Come out from directories **subpack** and **mainpack** and Use the following command to Run the program.

java mainpack.subpack.Sample

10.9 CLASSPATH Environment Variable

By default the Java run-time system search the class file at the current location. But if we talk about packages, it must not be stored at the same location. Therefore, we have to provide the path, where packages and class path are stored. The **CLASSPATH** is a user defined environment variable that is used by Java to determine where predefined classes are located. The CLASSPATH is also used by the Java Virtual Machine to load classes.

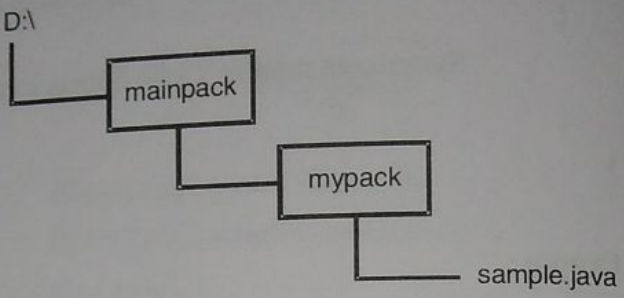
CLASSPATH can be set in two ways :

(a) By giving the command at command prompt.

set CLASSPATH=<package path>

Example 10.13

The **mypack** package is created in d:\mainpack. We want to store the class file of **sample.java** in it. The directory structure for this package is



Then the CLASSPATH needs to be set location of the package accordingly shown as :

C:\> set CLASSPATH = .; D:\ mainpack ;

Now, Java will look for java classes from the current directory instead of the **"D:\mainpack"** directory.

(b) Set the **CLASSPATH** using environment variable in windows.

Steps are :

(i) Right click on **My computer** icon as shown in figure 10.3.

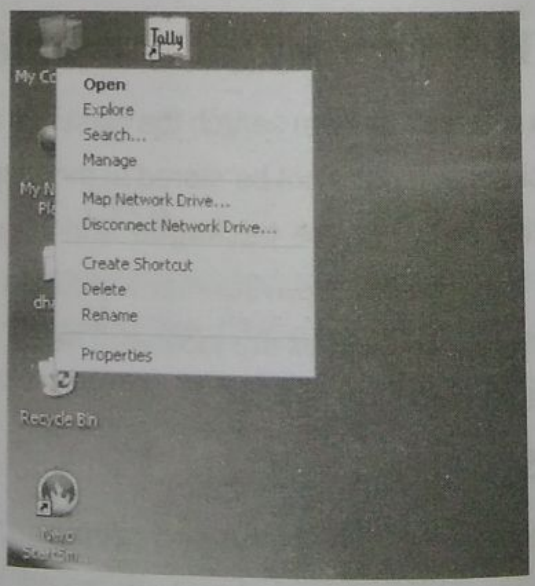


FIGURE 10.3

(ii) Select **properties**. System properties dialog box appears as shown in figure 10.4.

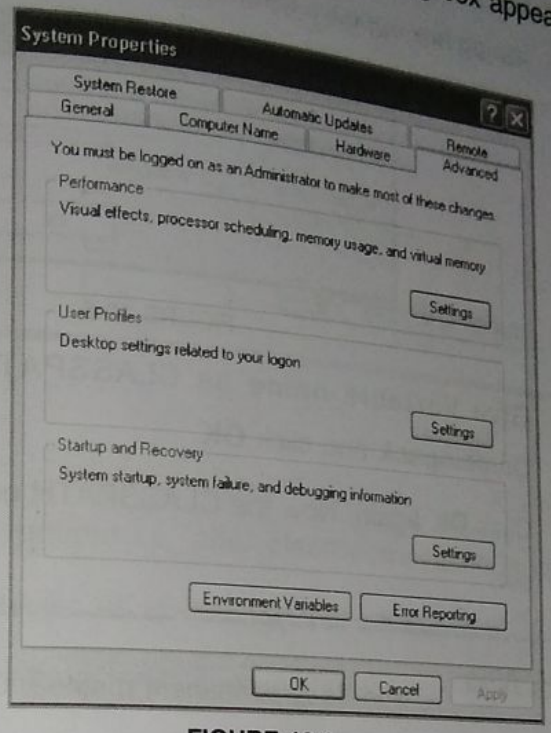


FIGURE 10.4

(iii) Click on **Advanced** tab. Click on **Environment Variables** button. Environment variables dialog box appears as shown in figure 10.5.

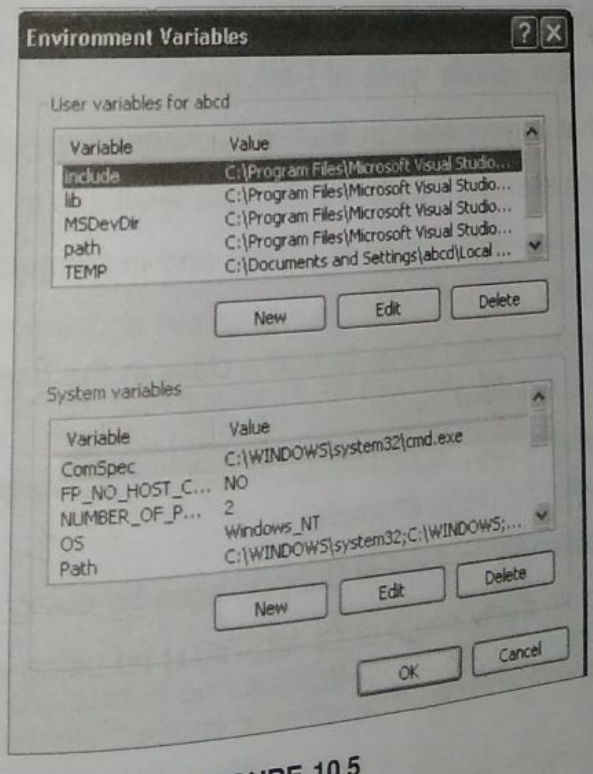


FIGURE 10.5

(iv) Click on **New** button in user variables for owner area. **New User Variable** dialog box appears as shown in figure 10.6.

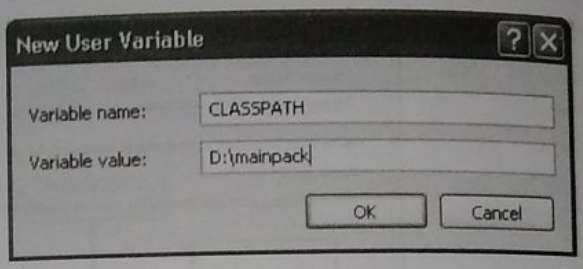
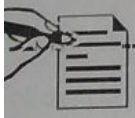


FIGURE 10.6

(v) Give **Variable name** as CLASSPATH and **Variable value** as **D:\mainpack** and click **OK**.

(vi) Click **OK** again. Now the CLASSPATH environment variable has been set.



Note : This is permanent method. Command prompt method is temporary.

Exercise

What is a Package? What are its advantages?

Explain various types of Packages.

What do you mean by Built-in packages?

Explain some Standard Packages available in Java.

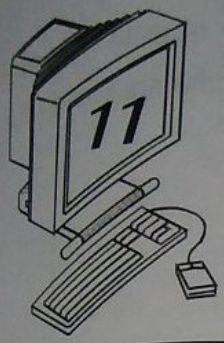
Explain various steps to Create and Implement an user defined package in Java by using example.

Explain various steps to Add a class in the user defined package in Java by using example.

How can you Access the User-defined Package Members in Java Program? Explain by using example.

Explain the concept of Subpackages by using example.

Explain the concept of CLASSPATH by using example.



Exceptions Handling

11.1 Introduction

Errors occurring at runtime i.e. after clean compilation is called **exceptions**. Exception handling allows us to manage runtime errors in a systematic manner. Some examples of runtime errors or **exception** are :

- Division by zero
- array index out-of-range
- arithmetic overflow
- unexpected arguments
- file not found etc.

The purpose of the exception handling mechanism is to provide means to detect and report an **exceptional circumstance** so that appropriate action can be taken. In this chapter, we will discuss some of the exception handling techniques of Java.

Note :

The **exception handling** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

11.2 Need for Exception Handling

The advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Example 11.1

Consider the following scenario :

```
statement 1;
statement 2;
statement 3;        // exception occurs
statement 4;
statement 5;
statement 6;
statement 7;
statement 8;
```

Suppose there is 8 statements in a program and an exception occured at statement 3, rest of the code will not be executed i.e. statement 4 to 8 will not run. If we perform **exception handling**, rest of the code (i.e. statement 4 to 8) will be executed. That is why we use exception handling in java.

11.3 Exceptions and their Types

In java, all exceptions are represented by **classes**. The hierarchy of exception classes commence from **Throwable** class which is the base class for an entire family of exception classes, declared in **java.lang** package as **java.lang.Throwable**. Exceptions are thrown if any kind of unusual condition occurs that can be caught. Sometimes it also happens that the exception could not be caught and the program may get terminated.

The hierarchy of java exception classes is shown in figure 11.1.

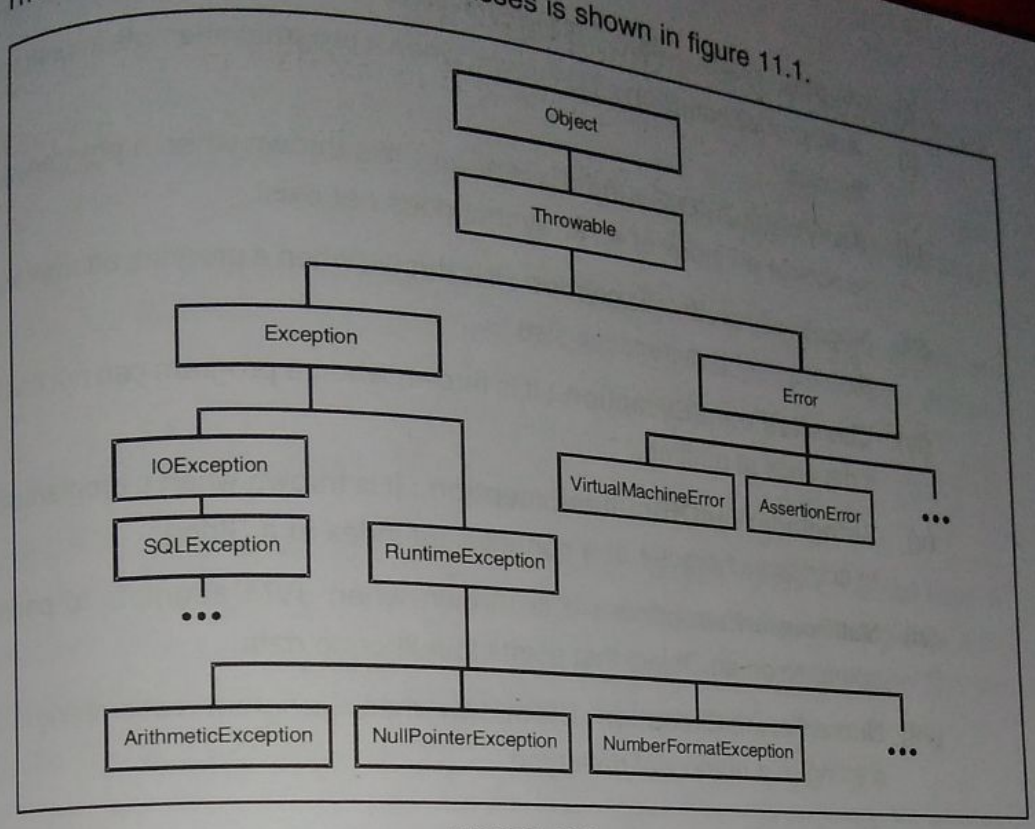


FIGURE 11.1

11.3.1 Types of Exception

There are three types of exceptions :

1. Checked Exception
2. Unchecked Exception
3. Error

11.3.1.1 Checked Exception

The classes that extend **Throwable** class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

11.3.1.2 Unchecked Exception

The classes that extend **RuntimeException** are known as unchecked exceptions e.g. ArithmeticException, NullPointerException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

The following are the common unchecked exceptions classes defined in java.lang

(i) **ArithmeticException :** It is thrown, when a program attempts to perform divi by zero.

(ii) **ArrayIndexOutOfBoundsException :** It is thrown when a program attempt to access an index of an array that does not exist.

(iii) **NegativeArraySizeException :** It is thrown when a program attempt to acce an array that has negative size.

(iv) **ClassNotFoundException :** It is thrown when a program can not find a clas it depends at runtime.

(v) **StringIndexOutOfBoundsException :** It is thrown when a program attempt to access a character at a non-existent index in a String.

(vi) **NullPointerException :** It is thrown when JVM attempts to perform a operation on an Object that points to null or no data.

(vii) **NumerFormatException :** It is thrown when a program is attempting to conve a string to a numerical data type.

11.3.1.3 Error

Errors are typically ignored in your code because you can rarely do anything abou an error. Errors are not exceptions at all, but problems that arise beyond the contro of the user or the programmer.

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionErro etc.

11.4 Exception Handling

Java exception handling is managed via five keywords : **try, catch, throw, throws** and **finally**.

A block of statements in which an exception can occur must be prefixed by the keyword **try**. This block of statements is known as **try block.** When an exception is detected, then the runtime informs the application that an error has occurred. This process of informing is called **throwing** an exception.

A **catch block** defined by the keyword **catch,** catches the exception thrown in the **try block** and handles it appropriately.

The **catch** block that catches an exception must immediately follows the **try** block that throws the exception.

The **finally** block is used to handle an exception that is not caught by any of the previous catch statement. The **finally block** is added immediately after the try block or after the last catch block.

To manually throw an exception, use the keyword **throw**. In some cases, an exception that is thrown out of a method must be specified as such by a **throws** clause.

11.4.1 Using try-catch block

Program statements that you want to monitor for exceptions are contained within a **try** block. The try block is immediately followed by one or more catch blocks. If an exception occurs within the **try** block, it is thrown. Each catch block specifies the type of exception it can catch and handle.

Syntax of try-catch block is

```
try
{
statements;          // code that might generate exceptions
                     // If exception occur, throw exception object
}
catch (ExceptionType exobj)
{
statements;          // catch exception object thrown by try,
                     // if its type matches with Exception type
                     // Handles the exception i.e. takes corrective Action
}
```

When the **try** block generates an exception, the program control jumps to the **catch** statement of the catch block and catch block is executed for handling the exception. When no exception is detected, the control goes to the statement immediately after the catch block.

Example 11.2

Program to illustrate the try-catch block.

```
class exception1
{
    public static void main(String args[])
    {
        int x=50, y=0;
        int result=0;
        try
        {
            result=x/y;        // Found Arithmetic Exception
                               // Exception Object is created & thrown
            System.out.println ("We are in try block");
        }

        catch (ArithmeticException  e)// catching Arithmetic Exception obj
        {
            System.out.println("Arithmetic exception occurred");
            System.out.println(e);
        }
        System.out.println("This is the last statement");
    }
}
```

Output :

```
Arithmetic exception occurred
java.lang.ArithmeticException: /by zero
This is the last statement
```

It is clear from the above program that the statement having the message. Thi **the last statement** is executed, but the statement having the message. We ar **try block** is not executed. The statement **System.out.println(e);** displa description of an exception.

```
    int total=0;
    try
    {
        total=a[0]+a[1]+a[2]+a[3];  //Found ArrayIndexOutOfBoundsException so
                                    //An Object is created & thrown
    }

    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("You are trying to use an invalid index");
        System.out.println(e);
    }
    System.out.println("Total Marks : " + total);
    System.out.println ("This is the last statement");
  }
}
```

Output :

```
You are trying to use an invalid index
java.lang.ArrayIndexOutOfBoundsException: 3
Total Marks : 0
This is the last statement
```

.4.2 Using Multiple Catch Blocks

We can use more than one catch clause in a single try block however every catch block can handle only one type of exception.

In many cases, more than one exception could be raised by a single piece of code. In order to handle such a situation multiple catch clauses, each catching a different exception type can be specified.

Syntax is :

```
try
{
    ...
    ...          // code that might generate exception
}
catch (type1 obj1)
{
    ...
    ...          // Handle exceptions of Type1
}
catch (type2 obj2)
{
    ...
    ...          // Handle exceptions of Type2
}
    ...
    ...
catch (typei obji)
{
    ...
    ...          //Handle exceptions of Typei
}
```

If an exception is thrown, catch statements are inspected one by one and the first one whose argument type matches the exception type is executed.

The search for handlers stops once the catch clause is finished. Only the matching catch clause executes; It's not like a **switch** statement in which you need a **break** after each **case** to prevent the remaining ones from executing.

Note :

1. At a time only one Exception is occured and at a time only one catch block is executed.
2. All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for other exception.

Using Finally Statement

The **finally** statement is used to handle an exception that is not caught by any of the previous catch statements. A **finally** block can be used to handle any exception generated within a try block. The finally block will be executed whenever execution leaves a **try/catch** block, no matter what conditions cause it. That is, whether the **try** block ends normally or because of an exception, the last code executed is that defined by **finally**. Java finally block must be followed by try or catch block. The general from of **try/catch** that includes **finally** is :

```
try
{
        . . .
        . . .          // try code
}
catch (. . .)
{
        . . .
        . . .          // catch code
}
catch (. . .)
{
        . . .
        . . .          // catch code

. . .
. . .
finally
{
        . . .
        . . .          // finally code
}
```

Since **finally** block is guaranteed to execute. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

5.1 Using Nested Try Block

The try block within a try block is known as **nested try block** in java. Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such class, exception handlers have to be nested. If the innermost try statement does not have a corresponding catch statement then the catch handler of the upper try statement are inspected for a match and this continues till all the try's are exhausted. If no catch statement matches, then the java run-time system will handle the exception.

Syntax is

```
....
try
{
    ....
    ....          // try statements
    try
    {
        ....
        ....      // inner try statements
    }
    catch(Exception e)
    {
        ....
        ....
    }
}
catch(Exception e)
{
    ....
    ....
}
....
```

Example 11.7

Program to illustrate the concept of nested try block.

```
class Exception4
{
  public static void main(String args[])
  {
    try
    {
      try
      {
        int x = 50/0;
      }
      catch(ArithmeticException e)
      {
        System.out.println(e);
      }
      try
      {
        int a[ ] = new int[4];
        a[5] = 10;
      }
      catch(ArrayIndexOutOfBoundsException e)
      {
        System.out.println(e);
      }
      System.out.println("Remaining Statements");
    }
    catch(Exception e)
    {
      System.out.println("Exception handeled ");
    }

    System.out.println("The Last Statement");
  }
}
```

Output :

```
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
Remaining Statements
The Last Statement
```

11.5.2 Using Throw Keyword

Till now we have been catching exceptions thrown by the Java run-time system. The **throw** keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception by using throw keyword.

Therefore, **throw** keyword is used to manually throw an exception. To throw an exception find the appropriate exception class. Now create an object of that class and use the throw statement. The flow of execution immediately stops after the **throw** statement; any subsequent statements are not executed.

Syntax is :

throw instance ;

where **instance** is an object of type **Throwable**. Commonly, a **new** statement is used to create an instance. We can also create a **Throwable** object using a parameter into a **catch** clause.

Example 11.8

(a) throw new ArithmeticException( );

(b) throw new NullPointerException( );

In example (a), the **throw** statement is manually throwing **ArithmeticException**. In example (b), the **throw** statement is manually throwing **NullPointerException**.

Example 11.9

In this example, we have created the validate method having integer parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message.

```
class Exception5
{
    static void validate(int age)
    {
        if(age<18)
            throw new ArithmeticException("Not Valid To Vote");
        else
            System.out.println("Welcome to Vote");
    }
    public static void main(String agrs[])
    {
        validate(15);
        System.out.println("Last statement");
    }
}
```

Output :

```
Exception in thread "main" java.lang.ArithmeticException: NotValid To Vote
at Exception5.validate(chap117.java:6)
at Exception5.main(chap117.java:12)
```

5.2.1 Rethrowing an Exception

An exception caught by one **catch** can be **rethrown** allowing it to be caught by an outer **catch**. Rethrowing an exception allows multiple handlers access to the exception. One handler can take care of one aspect of an exception while a second handler takes care of another.

Syntax is :

throw e ;

where **throw** is a keyword.

e is an exception which we want to rethrow.

Example 11.10

```
class Rethrowdemo
{
    public static void Exception_Generate( )
    {
        int a[] = {3, 6, 9, 12, 15, 18};
        int b[] = {3, 0, 6, 0, 9};              // a is longer than b
        for (int x = 0; x < a.length; x++)
        {
            try
            {
                System.out.println(a[x] + " / " +b[x] + " = " + a[x] / b[x]);
            }
            catch(ArithmeticException e)
            {
                System.out.println("Division by zero not allowed");
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("No Matching Element Found");
                throw e; // rethrowing the exception
            }
        }
    }
}
class Exception8
{
    public static void main(String args[ ])
    {
        try
        {
            Rethrowdemo.Exception_Generate( );
        }
```

```
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Abnormal Program Termination");
        }
    }
}
```

Output :

```
3 / 3 = 1
Division by zero not allowed
9 / 6 = 2
Division by zero not allowed
15 / 9 = 16
No Matching Element Found
Abnormal Program Termination
```

.3 Using Throws Keyword

The **throws** keyword is used to declare an exception. Throws is an alternate way to indicate that a method may possibly throw an exception. Any exception that is thrown out of a method must be specified as such by a **throws** clause. This is possible by adding the throws keyword after the signature of the method and followed by the name of one or more exceptions.

Syntax is :

```
return type method_name (parameter_list) throws exception_list
{
    ...
    ...     // body of the method
}
```

Here, **exception_list** is a comma separate list of the exceptions that a method can throw.

If we are calling a method that declares an exception, we must either caught (i.e. handle the exception using try/catch) or declare the exception.

(a) We handle the exception

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

Example 11.11

```
import java.io.*;
class Sample
{
    void method() throws IOException
    {
            throw new IOException("device error");
    }
}
class Testthrows2
{
    public static void main(String args[])
    {
            try
            {
                Sample obj=new Sample();
                obj.method();
            }
            catch(Exception e)
            {
                System.out.println("Exception Handled");
            }
            System.out.println("Last Statement");
    }
}
```

Output :

```
Exception Handled
Last Statement
```

(b) We declare the exception

(i) In case we declare the exception, if exception does not occur, the code will be executed fine.

(ii) In case we declare the exception if exception occures, an exception will be thrown at runtime because **throws does not handle the exception.**

Example 11.12

```
import java.io.*;
class Sample1
{
    void method() throws IOException
    {
        throw new IOException("device error");
    }
}
class Testthrows3
{
    public static void main(String args[]) throws IOException //declare exception
    {
        try
        {
            Sample1 obj=new Sample1();
            obj.method();
        }
        catch(Exception e)
        {
            System.out.println("Exception Handled");
        }
        System.out.println("Last Statement");
    }
}
```

Output :

```
Exception Handled
Last Statement
```

11.5.4 Difference between Throw and Throws

Differences between throw and throws keywords are given in table 11.1.

TABLE 11.1

Throw	Throws
1. Java throw keyword is used to explicitly throw an exception.	1. Java throws keyword is used to declare an exception.
2. Throw is followed by an instance.	2. Throws is followed by class.
3. Throw is used within the method.	3. Throws is used with the method signature.
4. You cannot throw multiple exceptions.	4. You can declare multiple exceptions e.g. public void method( ) throws IOException, SQLException.

11.6 Exception Handling with Method Overriding

There are many rules if we talk about methodoverriding with exception handling. The rules are as follows :

(a) If the superclass method does not declare an exception

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

(b) If the superclass method declares an exception

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

```
class Derived extends Base
{
    void display( ) throws ArithmeticException
    //Arithmetic Exception is child exception in hierarchy of java exception classes.
    {
        System.out.println("Exception in Derived Class Method");
    }

    public static void main(String args[])
    {
        Base obj = new Derived();
        try
        {
            obj.display();
        }
        catch(Exception e){ }
    }
```

Output :

Exception in Derived Class Method

7 User-Defined Exception (Writing Exception Subclasses)

Apart from the **built-in exception** classes, we can define our own exception classes. This is very useful when we want to define exception types which are having a behavior different from the standard exception type, particularly when we want to do validations in our application.

If we are creating our own Exception that is known as **custom exception or user-defined exception**. By the help of user-defined exception, we can have our own exception and message.

Example 11.18

```
class MyException extends Exception          // Defining own exception subclass
{
    MyException(String s)                    // Constructor
    {
        super(s);
    }
}
class TestException
{
    static void validate(int age) throws MyException
    {
        if(age<18)
            throw new MyException("Not Valid to Vote");
        else
            System.out.println("Welcome to vote");
    }
    public static void main(String args[])
    {
        try
        {
            validate(15);
        }
        catch(Exception e)
        {
            System.out.println("Exception occured: " + e);
        }
        System.out.println("Last Statement ");
    }
}
```

Output :

Exception occured : My Exception: Not Valid to Vote
Last Statement