BLOCK 2
INHERITANCE, POLYMORPHISM AND PACKAGES

BLOCK 2 INTRODUCTION

In object oriented programming reusability of code is one of the important feature which is achieved by implementing inheritance. Polymorphism proved flexibility in system design. Exceptions handling provide option of graceful exit from program in abnormal/error conditions. Use defined packages provide facility to have your own features/services as a bundle and can be reused as per requirements. This block explains Inheritance, Polymorphism, Exception Handling, and Packages and Interfaces. This block consists of 4 units and is organised as follows:

Unit 5: This unit explains how to implement inheritance in Java, advantages inheritance, and type of inheritance supported by Java. Also, access control in inheritance implementation, use of super keyword, use of final keyword, and implementation of inheritance hierarchy are explained in this unit.

Unit 6: This unit discuss about polymorphism one of the very important feature of object oriented programming. You will learn types of polymorphism and advantage of polymorphism in this unit. This unit explains method overloading and method overriding. Also, the concept of abstract class and in application is explained in this unit.

Unit 7: In this unit you will learn assertion and its use, how to apply annotations. Exception handling is one of the important features of Java programming. This unit explains concept and implementations of exceptions handling with the help of suitable examples. Also, this unit explains how to create user defined exceptions.

Unit 8: This unit explains how to create user defined package. Access rules in packages are explained in this unit. Also this unit explains how interfaces are defined and implemented.

UNIT 5 INHERITANCE

Structure **Page No.**

5.0 Introduction 111
5.1 Objectives 111
5.2 Basics of Inheritance 111
5.3 Advantages of Inheritance 115
5.4 Member Access and Inheritance 115
5.5 Types of Access Specifiers 116
5.6 Types of Inheritance 121
5.7 Use of Super Keyword 123
5.7.1 To call the superclass constructor 123
5.7.2 To access the member of the superclass 125
5.8 Creating Multilevel Class Hierarchy 126
5.9 Demonstrating Order of Constructors Execution 130
5.10 Use of Final Keyword 131
5.11 Summary 134
5.12 Solutions/Answer to Check Your Progress 134
5.13 References/Further Reading 137

5.0 INTRODUCTION

In the last unit of the previous block, you learned about class and objects. This unit takes you further in learning java programming and will explain one important concept of object-oriented programming, known as inheritance. Inheritance can be known as one of the essential tools in object-oriented programming language because it enables the reusability of properties of other classes. It takes advantage of the similarities that exist between various classes. It helps in data generalization and code reduction. If we look at the history of Inheritance, it was invented in 1969 for Simula (a programming language) and is now being used throughout many object-oriented programming languages such as Java, C++ , C# , and Python, etc.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain basic concepts of inheritance,
- Write programs using the constructors along with inheritance,
- Apply access specifier concepts in hierarchical inheritance,and
- Use inheritance in solving complex programming problems.

5.2 BASICS OF INHERITANCE

The basic idea behind Inheritance in Java programming is to enable the programmer to create multiple new classes that are built upon existing classes. When these new classes inherit the existing classes, then you can reuse the methods and fields of the

parent class. Moreover, as per requirement, you can add the new methods/functionalities and attributes to your new class (inherited class). Inheritance represents the IS-A relationship which is also known as the parent-child relationship.

The syntax of Java Inheritance

```
class child_class extends parent_class
        {
           //fields
           //methods
        }
```

In the syntax, you can see the child_class is created using the keyword class, and it inherits the properties of the parent class with the help of the *extends keyword*. And other additional methods and fields also can be created as shown in the syntax.

In general, the child class is known as sub-class, and the parent class is known as super-class.

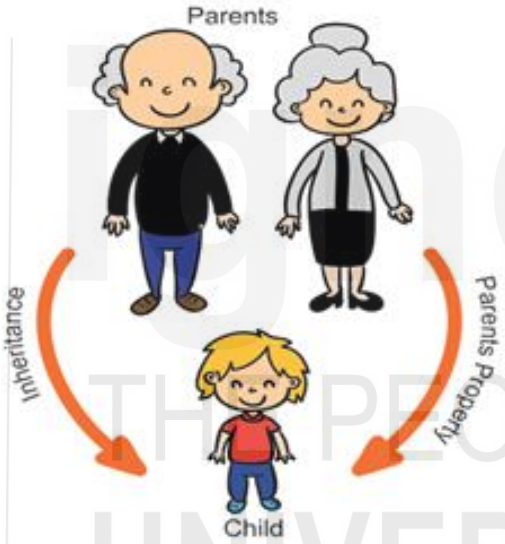


Figure 5.1: Inheritance- parents to child

Figure 5.1, shows a real-life example in which the child inherits the properties from their parents. A child will have the feature of its parents as well as its own features.In programming context, you may understand it in a way where code written in the parent class is available to the child class, and there is no need to write that code again. This reusability of code not only minimises the effort in applications development but also save a lot of time and resources.

The objective behind Inheritance in Java is to create new classes built upon existing classes, i.e. new classes will be able to use the attributes and methods of already exiting classes. Inheritance facilitates reuse of the methods and fields of the parent class. Moreover, you can also create new methods and fields in the new class as per requirement.

It also facilitates the programmer to achieve *run time polymorphism* and code reusability. This indicates that there is no need to write the same code again and again while using it in other classes. About polymorphism, you will learn in the next unit of this block.

For showing the Inheritance, there must be at least two classes : The class which inherits the methods and data members/fields from parent class is known as a

subclass(child class) of the class from which it inherits, and the class from which subclass is inherited is known as a base class or parent class.

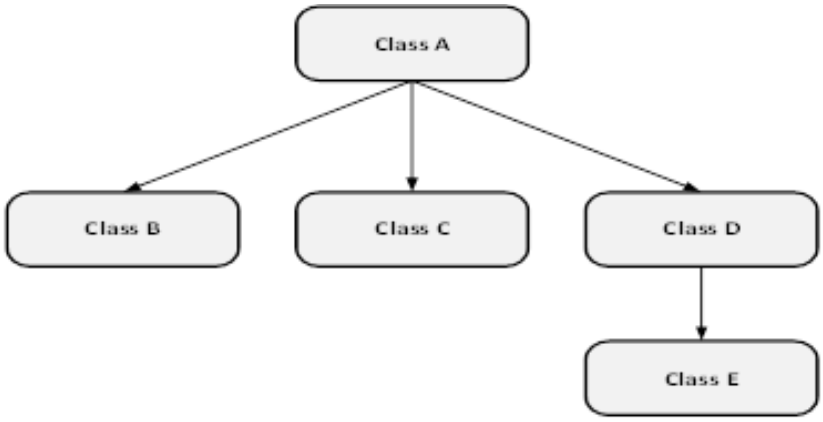


Figure 5.2: Class hierarchy

In figure 5.2, class B, class C, and class D (which are known as "sibling classes") inherit the properties of class A. If a class, for example, class B is a subclass of class A then we say that class A is the superclass of class B. Sometimes, we also use the term derived class and base class instead of subclass and superclass respectively.

Inheritance can be carried to several generations of the classes, as shown in figure 5.2. Class E is a subclass of class D, which is also a subclass of class A. In this case, we can also consider like class E is a subclass of class A, even though it is not a direct subclass. This whole phenomenon of classes representation is known as a class hierarchy.

The fundamental idea behind Inheritance in Java is to enhance code reusability by inheriting the methods and fields of parent classes. In figure 5.3, it is shown that the Electronics class is the parent class for the child classes "Phone" and "Sound System" which indicates that these two classes will have the properties of Electronics class.

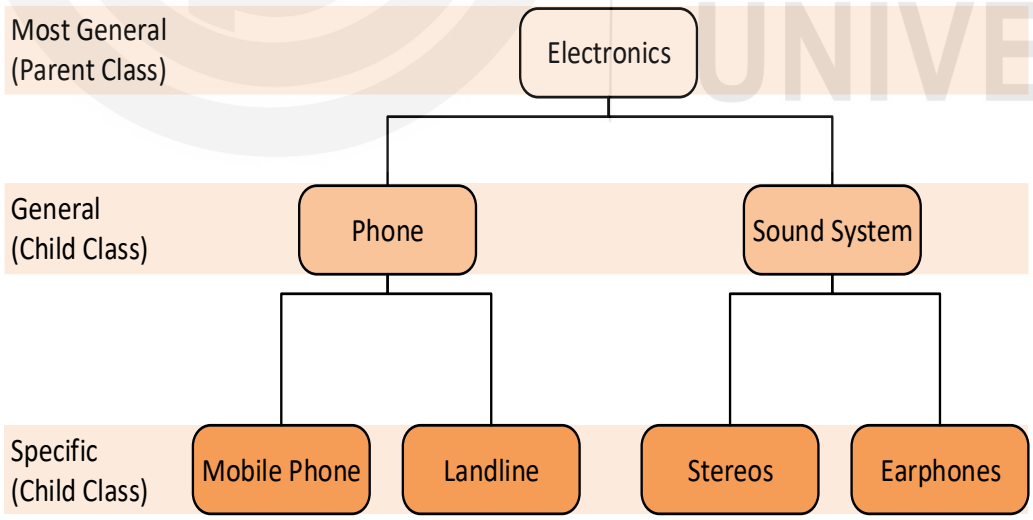


Figure 5.3: Hierarchy and generality of classes

Similarly, Mobile Phone and Landline classes are deriving the properties of the "Phone" class, which is treated as a parent class. And Stereos and Earphones are treated as a child class of the class "Sound System". This class hierarchy is moving from general to specific. In figure 5.3, Electronics is the most general class while the

classes at the bottom are the most specific.

Let us discuss the programming aspects of Inheritance of Java with the help of a simple example in which "Vehicle" is a parent class, and "four-wheeler" is a child class. The child class inherits the properties of the parent class with the help of the **extends** keyword.

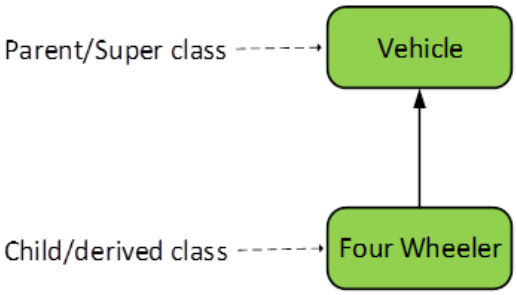
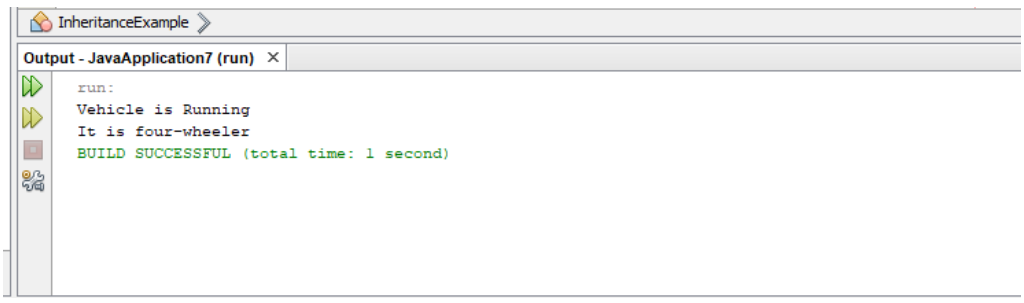


Figure 5.4: Super and Sub-class relation

Programming Example 1:

```
class Vehicle
{
  /* Vehicle is a parent class */
  void run()
  {
    System.out.println("Vehicle is Running");
  }
}

class FourWheeler extends Vehicle
{
  /* FourWheeler is a child class */
  void wheel()
  {
    System.out.println("It is four-wheeler");
  }
}

class InheritancesExamples
{
  public static void main(String args[])
  {
    FourWheelerfour_wheeler = new FourWheeler ();  /*creating object*/
    four_wheeler.run();
    four_wheeler.wheel();
  }
}
```

Output:



In example 1, class FourWheeler (child class) inherits the properties of the class Vehicle (parent class) with the help of the **extends** keyword. in the example, run() is a method of the superclass, and wheel() is the method of FourWheeler class. Now you can see the object "four_wheeler" which is an object of the child class, can inherit its parent class vehicle's method run().

5.3 ADVANTAGES OF INHERITANCE

One of the biggest advantages of Inheritance is that the code already mentioned in the parent class does not need to be rewritten in the child class,because child class may use the code without rewriting it.

Another advantage of Inheritance is it can save time and effort because of code reusability. It also minimizes the development and maintenance costs because of the clear model structure, which is quite easy to understand. It is also possible to keep some data private in base class, which derived classes cannot use.

In addition to the points mentioned above, you may consider the whole application development in context of - Inheritance represents the *IS-A relationship*, also known as the parent-child relationship.When you are designing a system, you may consider some features that will be inherited/accessed from the parent's classes, and some features which need some special implementation may directly be implemented in a derived class. More details about the implementation of this philosophy will be discussed in the later section of this unit andalso in the next unit of this block.

5.4 MEMBER ACCESS AND INHERITANCE

Access modifiers are simply keywords in Java programming that provides accessibility of a class and its members. In Java there are four types of access specifiers that is public, default, protected and private. These access specifiers are used to control what parts of a program can be accessed by whom. Well defined set of permissions can prevent unauthorized access and misuse of data.

Out of these access modifiers, protected is accessible within package and outside the package but with the help of implementing Inheritance feature only. While public and private are alike, when a member of a class is declared private, it can be accessed by other members of the class only. The public access specifier makes data members and member functions accessible outside of the class. That is why the public modifier has always preceded the main( ) method because it is called by code that is outside the program.

In case when no access modifier is used, then a member of the class will be public within its own package but cannot be accessed outside of its own package. Examples for declaration using access specifiers are given below.

public int numb;

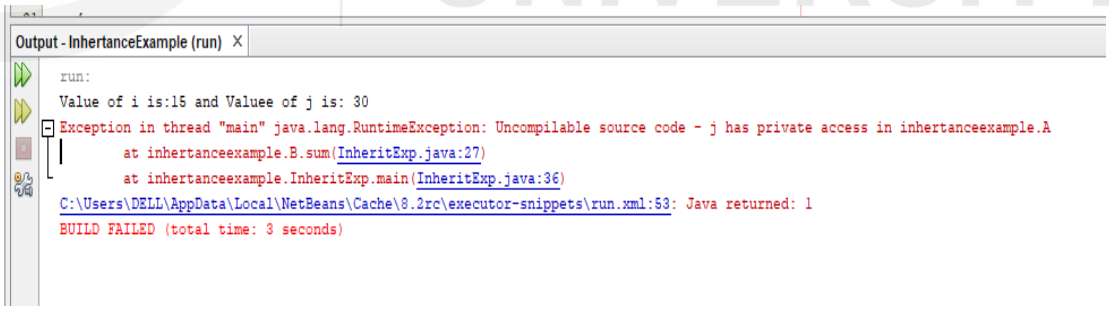
private int age;

private int mymethod (int a , char b) {…….}

Further, if we look at in terms of Inheritance, a subclass includes all of the members of its superclass, but it cannot access the superclass members that are declared as private. Let have a look at the example given below.

```
/*--In a class hierarchy, private members remain private to their class*/

// It is a superclass
class A
{
   int i;
   private int j;
   void setij(int x, int y)
{
       i = x;
       j = y;
System.out.println("Value of i is:"+i+" and Valuee of j is: "+j);
  }
}
/*---A's j is not accessible here in this class------*/
class B extends A
{
   int total;
   void sum()
{
       total = i + j;   // Error, j is not accessible here as  it is declared private
}
}
class Main
{
public static void main (String[] args)
  {
       B Bobj = new B();
       Bobj.setij(15, 30);
       Bobj.sum();
       System.out.println("Total is:" +Bobj.total);
   }
}
```

The Output:



This program will not compile because access of j inside the sum( ) method of class B causes an access violation. Since j is declared as private, it is only accessible to other members of its own class only.

5.5 TYPES OF ACCESS SPECIFIERS

If you have declared anything public, then it can be accessed in any other class and package, but it is declared as private, then it can be accessed in the defined class only.Whenever a member does not have an explicit access specification, it can be

accessed in a subclass as well as in other classes of the same package. Protected members are the members which can be seen in the direct subclass of other packages. A summary of various access specifiers is shown in table 5.1.

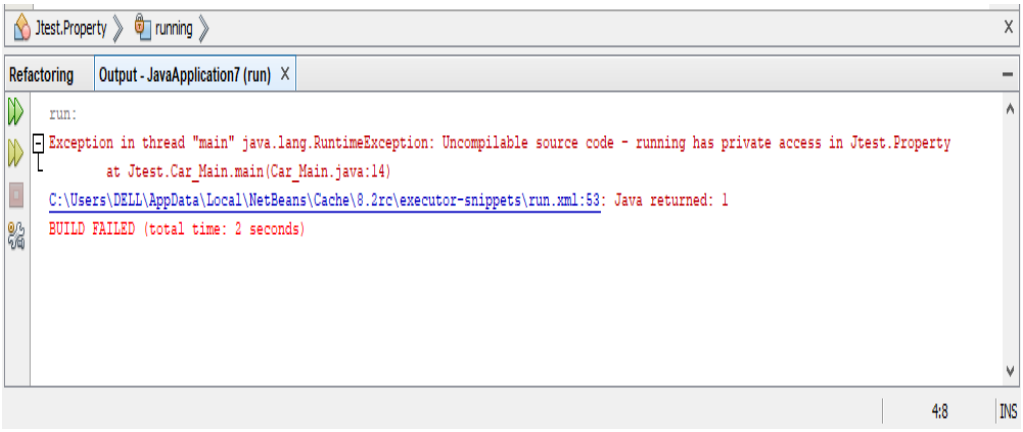
Table 5.1 Summary of Access Specifier

High restriction ----→Low restriction				
	Private	Default	Protected	Public
Inside Class	YES	YES	YES	YES
Inside Package	NO	YES	YES	YES
Outside Package subclass	NO	NO	YES	YES
Outside Package	NO	NO	NO	YES

Let's have a look at the example for private access specifier; in this example, you can see that there are two classes "Property" and "Car_Main" inside the package Jtest. In the Property class, private member running is defined, and we are trying to use this method in another class, Car_Main.

```
package Jtest;
public class Property
{
   private String running;
}

package Jtest;

public class Car_Main
{
   public static void main(String[] args)
  {
      Property obj = new Property();
      obj.running = "I is a fast running car";
      System.out.println("Property of the car is: "+obj.running);
   }
}
```

Output:



The program's output shows that private member "running" can not be accessed in another class Car_Main in the same package. It concludes as the private member is defined.

Now, if you want to access the private member of the class into another class of the same package, then it can be accessed with the help of the public method of the same class in which the private member is defined.

For example, Setproperty and Getproperty are these two public methods; with the help of these two methods, private members can be accessed into Car_Main class which is a different class.

```
package Jtest;
public class Property
{
  private String running;

  public void Setproperty(String running)
  {
    this.running = running;
  }
  public String Getproperty()
  {
    return this.running;
  }
}

package Jtest;

public class Car_Main
{
  public static void main(String[] args)
  {
    Property obj = new Property();
    obj.Setproperty("It is a fast running Car");
    System.out.println("Property of the car is: "+obj.Getproperty());
  }
}
```

The output of the program:

Property of the car is: It is a fast running Car

In this example, you can see that Setproperty method is used to set the value of the private member in the same class and then, with the help of Getproperty method; it is accessed in the Car_Main class. And you can see that the program executes successfully and shows the outcome.

The example given below explains the concept of protected and public access specifiers.

```
package Jtest;

public class Animal
{
  public int legcount;
  protected void Display()
  {
    System.out.println("I am a protected Animal");
```

```
  }
  public void Display2()
  {
    System.out.println("I am a Public Animal");
    System.out.println("I have "+legcount+" legs");
  }
}

package Jtest;
public class PetDog extends Animal
{
  public static void main(String[] args)
  {
    PetDog pd = new PetDog();
    pd.Display();
  }
}
```

Output of the program:

I am a protected Animal

Another example program for further clarification is given below:

```
package Jtest;

public class Main
{
  public static void main(String[] args)
  {
    Animal ani = new Animal();
    ani.legcount = 4;
    ani.Display2();
  }
}
```

Output of the program:

I am a Public Animal
I have 4 legs

☞ **Check Your Progress-1**

1) Ram has written the code given below, but it is showing compile-time error. Can you find out the mistake he has made?

```
class Ram
{
   //Class Ram Members
}

class Shyam
{
   //Class Shyam Members
}

class Reet extends Ram, Shyam
{
   //Class Reeta Members
}
```

2) Find out the output of the program given below.

```
class A
{
   int methodA(inti)
   {
      i /= 10;

      returni;
   }
}

class B extends A
{
   int methodB(inti)
   {
      i *= 20;

      return methodA(i);
   }
}

public class MainClass
{
   public static void main(String[] args)
   {
      B b = newB();

      System.out.println(b.methodB(100));
   }
}
```

3) Why can you not instantiate the Class Ram in the below code outside the package even though it has public constructor?

```
package package1;

class Ram
{
   public Ram()
   {
      //public constructor
   }
}

package package2;
```

```
import package1.*;

class Ravi
{
   Ram a = newRam();
}
```

4) Is the code given below written correctly? If yes, then what will it's output be?

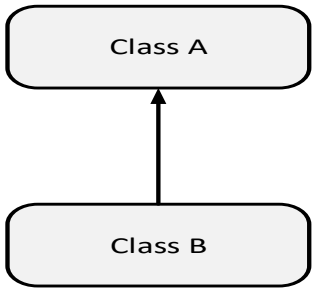
```
package pack1;

class A
{
   protected static String s = "A";
}

class B extends A
{

}

class C extends B
{
   static void methodOfC()
   {
      System.out.println(s);
   }
}

public class MainClass
{
   public static void main(String[] args)
   {
      C.methodOfC();
   }
}
```

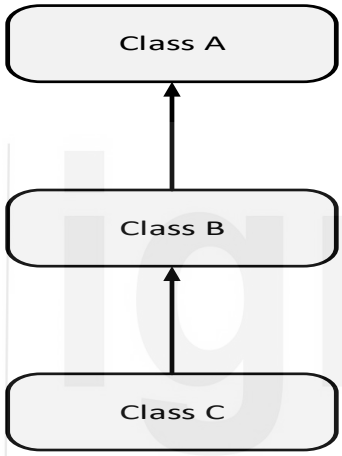
5.6 TYPES OF INHERITANCE

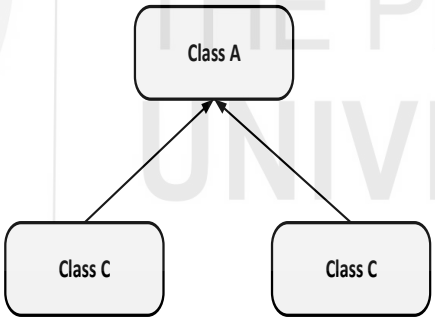
Java supports many types of Inheritance. The table given below gives a summarised presentation of the inheritance supported by Java.

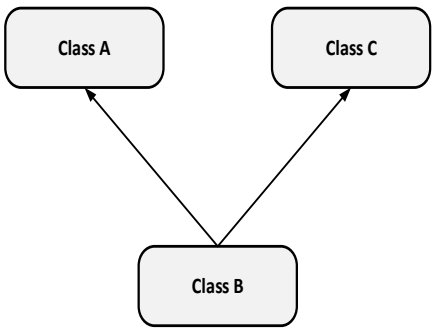
Table 5.2 Types of Inheritance

Name of Inheritance	Block Diagram	Syntax

Single Inheritance		public class A
		{
		-------- 	
		}
		public class B **extends** A
		{
		--------- 	
		}

Multi-Level Inheritance		public class A
		{
		-------- 	
		}
		public class B **extends** A
		{
		--------- 	
		}
		public class C **extends** B
		{
		--------- 	
		}

Hierarchical Inheritance		public class A{
		-------- 	
		}
		public class B **extends** A{
		--------- 	
		}
		public class C **extends** A{
		--------- 	
		}

Multiple Inheritance		public class A
		{
		-------- 	
		}
		public class B
		{
		--------- 	
		}
		public class C **extends** A,B
		{
		--------- 	
		}//**Java does not support multiple inheritances**

Java supports single inheritance, hierarchical inheritance and multi level inheritance. But multiple inheritance is not supported by java. Java provides the feature known as

Interface, which fills the gap of non supporting of multiple inheritances. You will learn about Interface in unit 4 of this block.

5.7 USE OF SUPER KEYWORDS

Java defines a special variable named "super". This keyword is used to refer to the objects of the immediate parent class. In general, super refers to the object that contains the method. It forgets about the objects of the class in which you are writing, and it remembers about the objects that belong to the superclass in that class in which you are writing.

The **super** keyword does not know about the functionalities of methods and variables of the superclass, it can only be used to call to methods and variables in the superclass. The **super** has two general forms/uses, the first one is to call the superclass constructor, and the second is to access a member of the superclass that a member of a subclass has hidden.

5.7.1 Callingthe superclass constructor

A subclass can call a constructor defined by its superclass by using the following syntax of super:

super(arg-list); here arg-list indicates the arguments required by the superclass constructor. To see the use of super(), consider the following example given below. In the given example BoxWeight class inherits the properties of Box class.

```
/* ----------Definition of the Box class----------*/
class Box
{
   private double width;
   private double height;
   private double depth;

/*----------constructor with all the specified dimensions----------*/
   Box(double w, double h, double d)
    {
      width = w;
      height = h;
      depth = d;
   }
/*----------constructor when dimensions are not specified---------- */
Box()
    {
      width = -1;
      height = -1;
      depth = -1;
   }
/*----------constructor when box is in the form of cube. ----------*/
Box (double len)
    {
      width=height=depth=len;
   }
/*----------method to compute the volume----------*/
   double Volume()
    {
```

```
        return width*height*depth;
    }
}
class BoxWeight extends Box
{
   double weight;
/*-------------constructor with all the specified parameters----------------*/
BoxWeight (double w, double h, double d, double m)
    {
       super(w, h, d);/*--------call superclass constructor---------------- */
       weight=m;
    }
BoxWeight ()
    {
       super();
       weight=-1;
    }
BoxWeight (double len, double m)
    {
       super(len);
       weight = m;
    }
}
/*----------------the main class implementation--------------------------*/
public class SuperDemo
{
   public static void main(String args[])
       {
       BoxWeight box1 = new BoxWeight (20, 30, 25, 54.5);
       BoxWeight box2 = new BoxWeight ();
       BoxWeight cube = new BoxWeight (5, 7);
       double vol;
       vol = box1.Volume();
System.out.println ("Volume of the box1 is: "+ vol);
System.out.println ("Weight of the box1 is:"+box1.weight);
System.out.println ();

vol = box2.Volume();
System.out.println ("Volume of the box2 is:"+ vol);
System.out.println ("Weight of the box2 is:"+ box2.weight);
System.out.println ();

vol = cube.Volume();
System.out.println ("Volume of the cube is: "+vol);
System.out.println ("Weight of the cube is: "+cube.weight);
System.out.println ();
   }
}
```

Output of theprogram:

```
Volume of the box1 is: 15000.0
Weight of the box1 is: 54.5
```

Volume of the box2 is: -1.0
Weight of the box2 is: -1.0

Volume of the cube is: 125.0
Weight of the cube is: 7.0

Explanation:

In the child class BoxWeight, super call the constructor of the parent class as shown below.

super(w, h, d)----------> Box(double w, double h, double d)

super()---------->Box()

super(len)---------->Box (double len)

When we review the key concept behind super( ) is like when a subclass calls super

( ), it calls the constructor of the immediate superclass. The super( ) always refers to the superclass just above the calling class. This concept is applicable in a multileveled hierarchy also.

5.7.2 To access the member of the superclass

Another use of **super** keyword is similar to **this,** except that it always refers to the superclass of the subclass in which it is used. The general syntax to access the members of the superclass is as follows.

super.*member*

here, *member* can be either a method or an instance variable. Let's have a look on superclass Animal and child class Dog, in Dog class, the sound method calls the sound( ) method of the superclass with the help of super keyword.

```
class Animal
{
   public void sound()
    {
      System.out.println("The Animal makes sound");
   }
}
class Dog extends Animal
{
   public void sound()
    {
      super.sound();
      System.out.println("The dog barks: bow wow");
   }
}
public class Mainclass
{
   public static void main(String args[])
    {
      Animal mydog = new Dog();
      mydog.sound();
   }
}
```

Output of the program is as follows.

The Animal makes sound

The dog barks: bow wow

The above example was based on to access the superclass method, now let's have an example based on to access the instance variable of the superclass.

```
/*----------Using super to overcome the name hiding--------*/
class A
{
    int i;
}
/*--------Create a subclass by extending class A----------*/
class B extends A
{
   int i;  // this i hides the i in defined class A
   B(int a, int b)
   {
      super.i = a;  // i defined in class A
      i=b;          // i defined in class B
   }
   void show()
   {
      System.out.println("i is superclass: "+super.i);
      System.out.println("i in subclass: "+i);
   }
}
class Main
{
   public static void main(String args[ ])
   {
      B subobj = new B (5, 10);
      subobj.show( );
   }
}
```

Output of the program:

```
i in superclass: 5
i in subclass: 10
```

So, you can see the instance variable i in B hides the variable i defined in class A, super keyword allows access to the variable i defined in the superclass A. Also, you have already seen that super can be used to invoke the methods hidden by subclass.

5.8 CREATING MULTI-LEVEL CLASS HIERARCHY

Till now, we have seen a few simple examples of class hierarchies that consist of using a superclass and a subclass only. However, you can build hierarchies that can contain as many layers as you want. For example, given three classes Student, Marks, and Percentage, as shown in figure 5.5. Percentage can be a subclass of Marks, which is further a subclass of Student.

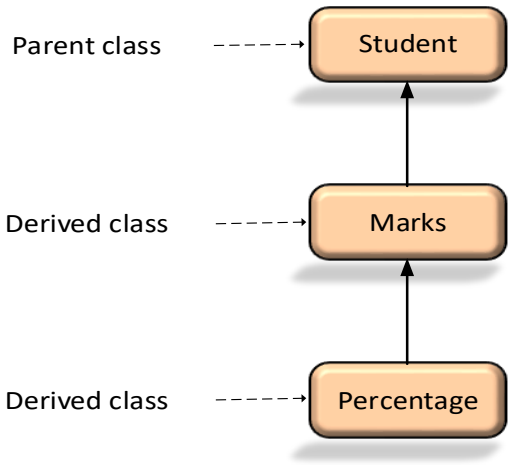


Figure 5.5 Multi-level class hierarchy

In multi-level hierarchy, each subclass inherits all of the properties found in its superclass.

```
class Student
{
  int rollnumb;
  String name;
  /*r and n are used to set the roll number and name*/
  Student (int r, String n)
  {
    rollnumb = r;
    name = n;
  }
  void show()
  {
    System.out.println ("Student Roll Number is: "+rollnumb);
    System.out.println ("Student Name is: "+name);
  }
}
class Marks extends Student
{
  int s1, s2, s3, s4, s5, sum;
  Marks(int r, String n, int m1, int m2, int m3, int m4, int m5)
  {
    super(r,n);
    s1 = m1;
    s2 = m2;
    s3 = m3;
    s4 = m4;
    s5 = m5;
  }
  void showmarks()
  {
    show();
    sum = s1 + s2 + s3 + s4 + s5;
    System.out.println ("Total marks: "+sum);
  }
}
class Percentage extends Marks
```

```
{
  float percent;
  Percentage (int r, String n, int m1, int m2, int m3, int m4, int m5, float p)
  {
    super(r, n, m1, m2, m3, m4, m5);
    percent = p;
  }
  void showpercent()
  {
    showmarks();
    percent = sum/5;
    System.out.println ("Percentage: "+percent+"%");
  }
}
class Main
{
  public static void main(String args[])
  {
    Percentage p = new Percentage (1025, "Sunil", 90, 85, 80, 88, 75, 0);
    p.showpercent ( );
  }
}
```

Output of the program:

```
Student Roll number is: 1025
Student Name: Sunil
Total marks: 418
Percentage: 83.0%
```

In the above program, the three classes Student, Marks, and Percentage form a multi-level hierarchy. The Student class serves as the parent class for the derived class Marks, which in turn serves as a parent class for the derived class Percentage.

☞ **Check Your Progress-2**

1) What will be the output of the program given below?

```
class A
{
  {
    System.out.println("You are in class A.");
  }
}

class B extends A
{
  {
    System.out.println("You are in class B");
  }
}

class C extends B
{
  {
    System.out.println("Here, Now in class C");
```

```
    }
}

public class MainClass
{
    public static void main(String[] args)
    {
        C c = new C();
    }
}
```

2) What will be the output of the program given below?

```
public class mainclass
{
    public static void main(String s[])
    {
        A a = new A();
        a.i = 21;
        B b = new B();
        b.i = 32; // LINE X
        b.j = 25;
        printI(a);
        printI(b); // LINE Y
        printJ(b);
    }

    public static void printI(A a1)
    {
        System.out.println(a1.i);
    }

    public static void printJ(B b1)
    {
        System.out.println(b1.j);
    }
}

class A
{
    int i;
}

class B extends A
{
    int j;
}
```

3) What do you mean by Generalized and Specialized classes in Java? Explain it with the help of a program.

.....

.....

.....

5.9 DEMONSTRATING ORDER OF CONSTRUCTORS EXECUTION

Constructors in Java are similar to methods that are invoked while creating an object in the class. In general,the order of constructor execution depends on the order in which the class hierarchy of the classes is created. In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.For example, given a subclass Parents and a superclass Grandparents, Grandparents constructor is executed before the Parents.

```
class Grandparents
{
  Grandparents()
  {
    System.out.println("Inside Grandparents constructor");
  }
}
class Parents extends Grandparents
{
  Parents()
  {
  System.out.println("Inside Parents constructor");
  }
}
class Child extends Parents
{
  Child()
  {
    System.out.println("Inside Child constructor");
  }
}
class CallingConstructor
{
  public static void main (String[] args)
  {
    Child c = new Child();
  }
}
```

Output of the program:

Inside **Grandparents** constructor
Inside **Parents** constructor
Inside **Child** constructor

It can be seen in the above example that the constructor execution is in the order of

derivation. If you think that this order makes sense, it is because a superclass has no knowledge of any subclass. Therefore any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore super class constructor must complete its execution first.

5.10 USE OF FINAL KEYWORD

The final keyword in Java is to **restrict the access of its contents from being modified**. If any field is declared as final, it means that the field cannot be modified and will become **essentially a constant**. This declaration can be made in one of two ways: First, you can give it a value during declaration, and second, you can assign a value within a constructor.

Assigning value while the declaration is one of the most common practices. For example

final int NUMB = 1;

The final keyword is quite useful when you want a variable to **always store the same value throughout the program**, for example

final double PI = 3.14;

The final keyword can be used along with the variables, methods, and classes like- final variable, final method, and final class respectively.

```
class Bike
{
  final int speedlimit=90; /*-------final variable--------*/
  void run()
  {
    speedlimit = 200;
    System.out.println("Speed limit is:" +speedlimit);
  }
  public static void main(String args[ ])
  {
    Bike bike = new Bike( );
    bike.run( );
  }
}
```

Output of the program:

Error: cannot assign a value to final variable speedlimit.

There can be a case of the final blank variable when it is not initialized during declaration. Then it is a must to initialize in the constructor of the class; otherwise, it will throw a compile-time error, for example.

```
class Blank_Test
{
  final int i;
  Blank_Test(int x)
  {
    i = x;       /*---initialization of final variable inside the constructor---*/
  }
```

```
}
class Mainclass
{
  public static void main (String[] args)
  {
    Blank_Test bt1 = new Blank_Test(15);
    System.out.println("Value of i is:"+bt1.i);
  }
}
```

Output of the program:

Value of i is: 15

Use of final keyword with the variables are stated above, now other two uses, i.e., with method and class apply to Inheritance.

Using final to Prevent Overriding

Overriding in Java is one of the most powerful features; there will be times when you will want to prevent it from occurring. If you want to disallow a method to be overridden, mention the final as a modifier at the beginning of its declaration. The method declared as final cannot be overridden. Look at the example given below.

```
class Ram
{
  final void method1( )
  {
    System.out.println("This is a final method.");
  }
}
class Ravi extends Ram
{
  void method1( )
  {
    /*---Error, Can't override----*/
    System.out.println("Illegal, method cand not be overridden");
  }
}
```

Output of the program:

Error: method1( ) in Ravi cannot override method1( ) in Ram

Because method1( ) is declared as final that's why it cannot be overridden in Ravi. If you will try to do so, it will give a compile-time error.

Using final to Prevent Inheritance

final keyword can also be used to prevent a class from being overridden. To do this, start the class declaration with the final keyword; doing this, it implicitly declares all of its methods as final.

```
final class Ram
{
  /*-------class Ram is declared as final---------- */
  void method()
  {
    System.out.println("This is a final class.");
```

```
  }
}
class Ravi extends Ram
{
  void method()
  {
    System.out.println("Ravi can not inherit the class Ram");
  }
}
```

Output of the program:
Error: cannot inherit from final Ram

☞ Check Your Progress-3

1) What do you mean by method hiding in Java?

---

2) The code given below showsa compile-time error. Can you suggest the appropriate corrections?

```
classX
{
   publicX(inti)
   {
      System.out.println(“Parent Class.”);
   }
}

classY extendsX
{
   publicY()
   {
      System.out.println(“Child Class.”);
   }
}
```

---

3) What will be the output of the program given below?

```
class Base
 {
   public void Print()
 {
      System.out.println("Welcome to Base class");
   }
}

class Derived extends Base {
```

```
    public void Print() {
        System.out.println("Now, it is Derived class");
    }
}

class Main
{
    public static void DoPrint( Base o ) {
        o.Print();
    }
    public static void main(String[] args) {
        Base x = new Base();
        Base y = new Derived();
        Derived z = new Derived();
        DoPrint(x);
        DoPrint(y);
        DoPrint(z);
    }
}
```

.....

.....

.....

4) Can you guess? How many public classes can a Java program have?

.....

.....

.....

5.11 SUMMARY

In this unit, we have discussed the concept of Inheritance, which is one of the important feature of object oriented programming. This unit explained how to derive the properties of one class into another class. We also discussed the access specifiers of data members along with a comparative summary.Also in the unit multi-level class hierarchy has been discussed. A few key terms like “Super”, “Final” and “This” are also discussed. Also in this unit,demonstration of constructors calling has beendiscussed.

5.12 SOLUTIONS/ANSWER TO CHECK YOUR PROGRESS

☞ **Check your progress 1:**

1) In Java programming, a class can not extend more than one class; here in this question, Reeta extends more than Ram and Shayam two classes.

   Java does not support Multiple Inheritance. If we try to perform, then it may suffer from collision of the methods with the same name because multiple classes may have the method with the same name.

   For example, if a Class $Child$ extends the properties of the class $Parent_1$ and class $Parent_2$ which have a method with the same name, then Class $Child$ will have two methods with the same name. This will create ambiguity and confusion

about which one to use. Therefore, to avoid this situation, java does not support multiple Inheritance.

2) The output of the program is: 200

3) In the given code, class Ram itself has been defined with default access modifier, which indicates that class Ram can be instantiated within the package in which it is defined but it cannot be instantiated outside the package, even though if it has a public constructor.

4) Yes, it is written correctly. Its output will be A

☞ **Check your progress 2:**

1) The output of the program is given below.

   You are in class A.

   You are in class B

   Here, Now in class C

2) The output of the program is:

   21

   32

   25

3) The top-level class or superclass is known as "Generalized class". It contains common data and common behaviour. And the low-level or sub-level classes are known as "specialized classes". In general, it contains more specific data.

Example:

```
class Person
{
int name;
int age;
}
Class Student extends Person
{
int roll_Number;
int marks;
}
class Employee extends Person
{
int employee_Id;
float emp_Salary;
}
```

In this class hierarchy, Person is generalized class and Student and Employee are the specialized classes.

☞ **Check your progress 3:**

1) If a subclass contains a static method with the same signature as a static method defined in the superclass, then the method defined in the subclass hides the one defined in the superclass.

There is a difference between hiding a static method and overriding an instance method:

- Invoked overridden instance method will be from subclass.
- And hidden static method that gets invoked depends on whether it is invoked from superclass or subclass.

For example, suppose that there are two classes; a superclass Animal and subclass Cat, which contains one instance method and one static method.

```
/*---------------Animal Class-----------------------------*/
package Jtest;
public class Animal
{
   public static void Eating_habit()
   {
      System.out.println("Generally, Animals eat grass.");
   }
   public void InstanceEating_habit()
   {
     System.out.println("Instance Method: Animals eat grass");
   }
}
/*-----------------------------Cat class----------------------------*/
package Jtest;
public class Cat extends Animal
{
   public static void Eating_habit()
   {
     System.out.println("Static: Cat drinks the Milk.");
   }
   public void InstanceEating_habit()
   {
     System.out.println("Instance Method: Cat drinks the Milk.");
   }
   public static void main(String[] args)
   {
     Cat mCat = new Cat();
     Animal mAnimal = mCat;
     Animal.Eating_habit();
     mAnimal.InstanceEating_habit();
   }
}
```

Output of the program:

Generally, Animals eat grass.

Instance Method: Cat drinks the Milk.

In the program, you can see that the Cat class overrides the instance method in Animal and hides the static method in Animal. The main method creates an instance of Cat and invokes Eating_habit, and InstancEating_habit.

2) Write explicit calling statement to super class constructor in Class Y constructor.

```
class X
{
   public X(int i)
   {
      System.out.println(“Parent Class.”);
   }
```

```
    }

    class Y extends X
    {
       public Y()
       {
          super(15)                    /* correction */
          System.out.println("Child Class.");
       }
    }
```

3)

Output:

```
Welcome to Base class
Now it is Derived class
Now it is Derived class
```

4) A Java program can have maximum only one public class.

5.13 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Savitch, Walter, " Java: An Introduction to roblem solving & programming", Pearson Education Limited, 2019.
- Neil, O. , "Teach yourself JAVA", Tata McGraw-Hill Education, 1999.
- Sarcar, Vaskaran. " The Concept of InheritanceIn Interactive Object-Oriented Programming in Java", Apress, Berkeley, CA, 2020.

UNIT 6 POLYMORPHISM

Structure **Page No.**

6.0 Introduction 138
6.1 Objectives 139
6.2 Introduction to Polymorphism 139
6.3 Advantages of Polymorphism 140
6.4 Types of Polymorphism 141
6.5 Method of Overloading 141
6.6 Method of Overriding 143
6.7 Abstract Class 145
6.8 Application of Abstract Class 147
6.9 Summary 153
6.10 Solutions/Answer to Check Your Progress 154
6.11 References/Further Reading 155

6.0 INTRODUCTION

Polymorphism literally means "having many forms" and is used as a feature in object-oriented programming that provides one interface for a general class of actions. The specific action is implemented to address the exact nature of the situation.

Figure 6.1: An example of polymorphism

Polymorphism in Java is the ability of an entity to take many forms as per requirement. As shown in figure 6.1, a man standing in the middle is only one, but he takes multiple roles like a dad for his children, an employee, a salesperson, and many more. This is known as polymorphism because one person takes many roles.

Another example may be the saving of two contact numbers of the same person. What can we do for this? We save these two numbers with the same name or in other words both number can be saved in one contact name only. That is also a kind of polymorphism. In a similar fashion, in java programming, one object can take multiple forms depending on the context of the program. In this unit, you will learn the concept of polymorphism and an important concept known as abstract class.

6.1 OBJECTIVES

After reading this unit, you will be able to:

- Explain the basics of polymorphism,
- Write the program on methodoverloading and method overriding,
- Differentiate between overloading and overriding,
- Use the abstract class and its application in programming, and
- Explain the difference between abstract class and interface.

6.2 INTRODUCTION TO POLYMORPHISM

The objective behind polymorphism in Java is to facilitate you for using the methods inherited by inheritance to perform different tasks. The object can take many forms. If there are one or more classes or objects related to each other by inheritance in a program, then polymorphism occurs. So, here you can say that the goal of polymorphism is communication, but it follows a different approach.

As you know, inheritance represents the IS-A relationship, so any java object is polymorphic if it can pass more than one IS-A test.

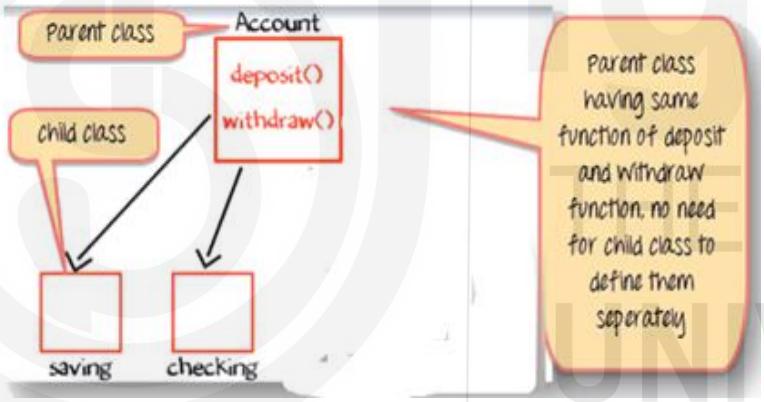


Figure 6.2: A scenario of inheritance

In figure 6.2, Account is the parent class with two methods: deposit( ) and withdraw( ). Also,Account class has two child classes: Savings and Current, and both the child classes performs the same operation of deposit and withdraw. So the child classes will not define the methods again and use the inherited one.

Figure 6.3: Single action in different ways

The concept of polymorphism in Java makes it possible to perform a single action in different ways with the idea of reusability. In figure 6.3, it is shown that all having speak( ) method in their interface, but each one of them performs it differently. All the descendants have performed with the same heads but with different method bodies.

Let us discuss the programming aspects of polymorphism of Java with thehelp of a simple example in which "Animal" is a parent class and "Cat" is a child class. Both the parent and child classes are using the same method makeSound().

Example Programming:

```
class Animal
{
  /*Animal is a parent class*/
  void makeSound()
  {
    System.out.println("Now Speak!");
  }
}
class Cat extends Animal
{
  /*Cat is a child class*/
  void makeSound()
  {
    System.out.println("Meow");
  }
}
class PolymorphismExample
{
  public static void main(String args[])
  {
    Animal a = new Animal();   /*creating object*/
    Cat d = new Cat();         /*creating object*/
    a.makeSound();
    d.makeSound();
  }
}
```

Output:

```
Now Speak!
Meow
```

In the above example program, you can see that both the classes are using the same method in different ways.

6.3 ADVANTAGES OF POLYMORPHISM

The major benefit of polymorphism is the reusability of codes which saves a lot of time and effort. Existing old classes and codes that were once tested and already implemented can be reused wherever required by using the concept of polymorphism, and the new functionality in the existing system can be added using the same interface.

Another benefit of polymorphism is that multiple data types(double, float, int, long, etc) can be stored in a single variable, making it easier to search for and implement these variables. You can use the single variable name to represent commonality in the features.

Also, Coupling between two different methods, i.e. the degree of interaction that violates the principle of hiding information, can be reduced by polymorphism.

6.4 TYPES OF POLYMORPHISM

In general, Java programming supports types of polymorphism which are as follows.

- Static Polymorphism
- Dynamic Polymorphism

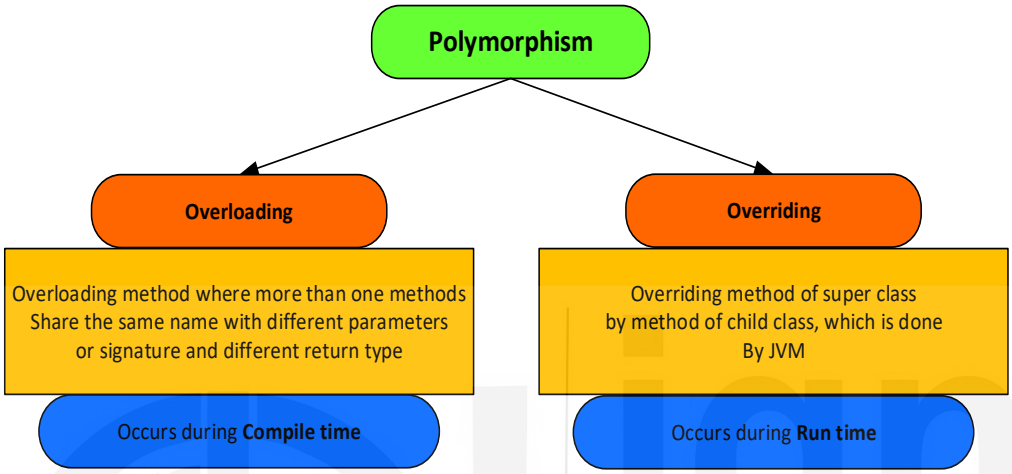


Figure 6.4: Types of Polymorphism

Static polymorphism is also known as **compile-time polymorphism** because compiler resolves the polymorphism during the compilation of the program by checking the method signatures. Static polymorphism can be achieved through **method overloading** with the same name but different parameters., which means the name of the methods is the same but the signature of the methods is different.

Dynamic polymorphism is also known as **run time polymorphism** because JVM(Java Virtual Machine) resolves the call to an overridden method at the runtime of the program. Dynamic polymorphism can be achieved by method overriding with the same name and parameters but in different classes. In the coming sections, you will learn more about method overloading and method overriding in detail.

6.5 OVERLOADING OF METHOD

Method overloading is one of the ways through which polymorphism is supported by Java. Method overloading is a concept of declaring multiple methods with the same name and different parameters of the same or different data types in the same class.Method overloading is compile-time or static polymorphism.

When an overloaded method is invoked, Java uses the number of parametres or data types of the parameters as it guide to distinguish which version of the overloaded method is to be called.Let us take an example of a method sum(int x, int y) having two parameters is different from the argument list of a method sum(int x, int y, int z) having three parameters.

There are three ways to overload a method:

1. There are different number of parameters in the argument list of the methods.

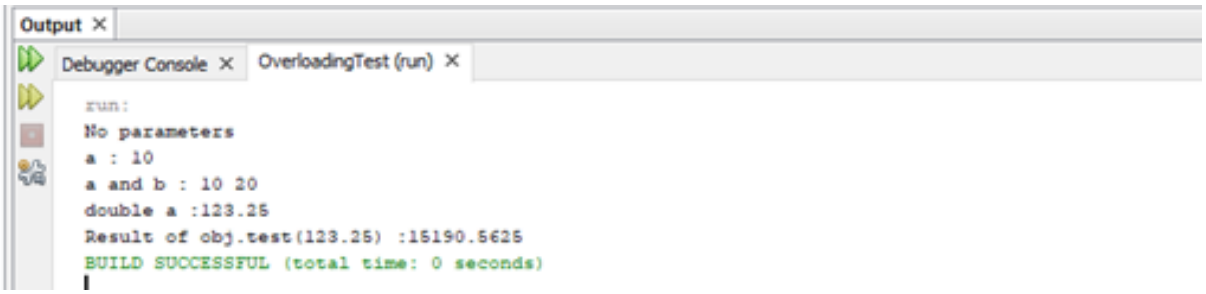
Example: sum(int, int)
sum(int, int, int)

2. There are different data types of the parameters in the argument list.
   Example: sum(float, float)
   sum(float, int)
3. There is a different sequence of data types of the parameters.
   Example: sum(float, int)
   sum(int, float)

Here is a simple example to illustrate the concept of method overloading.

```
/*-------Program to demonstrate method overloading-------*/
package overloadingtest;
class DemoOverload
{
void test( )
{
System.out.println("No parameters");
}
/*------overload test for one integer parameter-------*/
void test(int a)
{
System.out.println("a : " + a);
}
/*------overload test for two integer parameters-------*/
void test(int a, int b)
{
System.out.println("a and b : " + a + " " + b);
}
/*------overload test for a double parameter-------*/
double test(double a)
{
System.out.println("double a :" + a);
return a*a;
}
}
class MyOverloading
{
public static void main(String args[])
{
DemoOverloadobj = new DemoOverload( );
double result;
/*-------call all versions of test( )-------*/
obj.test( );
obj.test(10);
obj.test(10,20);
result = obj.test(123.25);
System.out.println("Result of obj.test(123.25) :" + result);
}
}
```

Output:



In the above program, test( ) is overloaded four times. The first version of test( ) takes no parameters, the second version takes one parameter of integer type, the third version takes two parameters of integer types, and the last version takes one parameter of a double data type with the double return type. However, return type does not play any role in method overloading.

☞ **Check Your Progress-1**

1) What is method overloading?

.....

.....

.....

2) What is compile-time polymorphism?

.....

.....

.....

3) Write a program to demonstratethe overloading of three methods with the same name.

.....

.....

.....

6.6 OVERRIDING OF METHOD

In a class hierarchy, when a method in the subclass has the same name and parameters as in its superclass, that method is said to override the method in superclass. When an overridden method is called from its subclass, then always the version of subclass method will be referred to, and the version of the method of superclass will remain hidden. This mechanism provides an opportunity for the subclass to have its own specific implementations of the overridden methods.

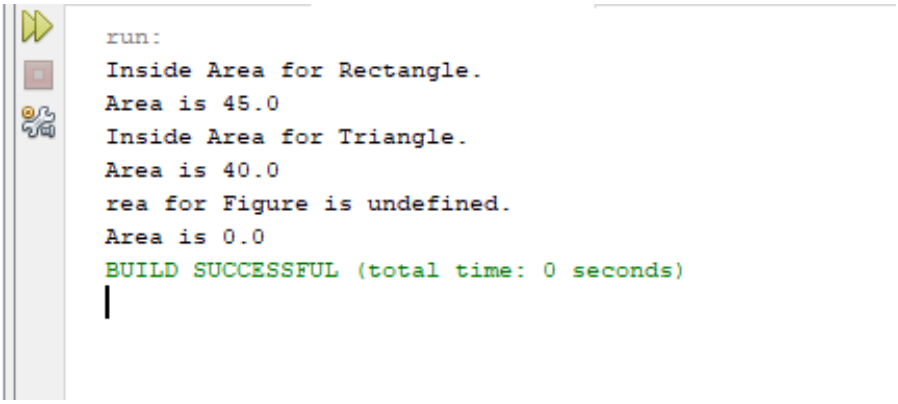
Overriding the method in Java is one of the ways to achieve runtime polymorphism.

Consider the following example:

```
class Figure
{
  /*Using runtime polymorphism*/
  double dim1;
  double dim2;
```

```
  Figure(double a, double b)
  {
    dim1 = a;
    dim2 = b;
  }
  double area( )
  {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}
class Rectangle extends Figure
{
  Rectangle(double a, double b)
  {
    super(a, b);
  }
  double area( )
  {
    /*Override area for Rectangle*/
    System.out.println("Inside Area for Rectangle.");
    return dim1*dim2;
  }
}
class Triangle extends Figure
{
  Triangle(double a, double b)
  {
    super(a, b);
  }
  double area( )
  {
    /*Override area for Right Triangle*/
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
class FindAreas
{
  public static void main(String args[])
  {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure ref;
    ref = r;
    System.out.println("Area is " + ref.area( ));
    ref = t;
    System.out.println("Area is " + ref.area( ));
    ref = f;
    System.out.println("Area is " + ref.area( ));
  }
}
```

Output:



In the above example, there is a superclass called Figure that stores the dimensions of two-dimensional objects. It also defines the method area( ) that calculates the area of an object. The two subclasses: Rectangle and Triangle are derived from the superclass Figure. Both the subclasses override the method area( ) to return their respective areas.

Certain rules must be followed while overriding the methods:

1. While overriding child class method signature and parent class method signatures must be the same; otherwise, it will generate compilation error.
2. The return types of overridden method and overriding method must be the same.
3. Methods declared final cannot be overridden.
4. Static methods are bound with class. Hence, it cannot be overridden.
5. The access level of methods cannot be more restrictive than the overridden method's access level.

6.7 ABSTRACT CLASS

Sometimes, there are certain situations where you want to create a superclass that only defines the generalized form that will be shared by all of its subclasses, leaving the details to be filled by the subclasses. In Java, an abstract method is a solution to this problem.

Abstract methods have only their declaration in the superclass. They are not defined in the superclass. These methods are overridden by the subclasses. The general syntax to declare abstract method is:

abstract ***type name***(**parameter-list**);

If any class has one or more abstract methods, it must be declared as abstract. This is done by simply using the **abstract** keyword. The abstract keyword is placed in front of the **class** keyword at the beginning of the class declaration. Instance of an abstract class cannot be directly created using **new** operator because its object cannot be created. It is also called an incomplete class as it is not fully defined. Abstract constructors or abstract static methods cannot be declared. Any subclass of an abstract class must either implement all the abstract methods of superclass or declare itself **abstract**.

You need to consider few key points for the abstract class.

- An abstract class is always declared with an abstract keyword.

- It can contain both abstract as well as non-abstract methods.
- You cannot instantiate the abstract class.
- Also, it can have final method if needed, which forces the subclasses not to change it.

Let us consider a simple example of a class with abstract method, followed by a class that implements that method.

```
abstract class Bank
{
  /*abstract class declaration*/
  abstract int getRateOfInterest( );
  /*abstract method declaration*/
}
class SBI extends Bank
{
  int getRateOfInterest()
  {
    /*overriding abstract method*/
    return 7;
  }
}
class PNB extends Bank
{
  int getRateOfInterest()
  {
    /*overriding abstract method*/
    return 8;
  }
}
class TestBank
{
  public static void main(String args[])
  {
    Bank b;       /*cannot be directly instantiated with new operator*/
    b = new SBI();
    System.out.println("Rate of Interest is: "+b.getRateOfInterest()+"%");
    b = new PNB();
    System.out.println("Rate of Interest is: "+b.getRateOfInterest()+"%");
  }
}
```

Output:

```
Rate of Interest is: 7%
Rate of Interest is: 8%
```

The above example consists of one superclass Bank which is declared abstract with an abstract method getRateOfInterest( ). SBI and PNB are two subclasses, overriding the method getRateOfInterest( ).

Let's take an example-based on employee and department to discuss the abstract class and abstract method.

```
abstract class Employee
{
  final int salary = 20000;
  public void display()
```

```
  {
    System.out.println("This is a method to display the employee    information.");
  }
  abstract public void Department();
}
public class Teacher extends Employee
{
  public static void main(String args[])
  {
    Teacher obj = new Teacher();
    obj.display();
    obj.Department();
    //obj.salary = 30000;
  }
  /*public void Department()
  {
    System.out.println("Computer Science Department");
  }
  */
}
```

Output of the program:

run:
java.lang.ExceptionInInitializerError Caused by: java.lang.RuntimeException: Uncompilable source code - Overridingtest.Teacher is not abstract and does not override abstract method Department() in Overridingtest.Employee
at Overridingtest.Teacher.<clinit>(Teacher.java:12)
Exception in thread "main"
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53:
Java returned: 1
BUILD FAILED (total time: 2 seconds)

In this example, you can see that output of the program is throwing an error because the abstract method in Employee class must be defined in derived classes. However, it is also defined but commented. That is the reason it is throwing an error. If you remove the comment and method is part of the code, then it will execute properly. Try to do it as a practice exercise.

6.8 APPLICATION OF ABSTRACT CLASS

In this section, you will see a an application of Abstract class along with the uses Interface. Interface is also having similar uses as abstract class, and it is defined as collection of methods. More about interface you will learn in unit 8 of this course. If you need to know more about interface in detailplease refer unit 8 of this block then return here to proceed further. Before discussing the application, let's have a look on some comparative points of Abstract class and interface.

Abstract Class	Interface
1.Any class can be marked as abstract by using the "abstract" keyword. It can have both abstract and non-abstract methods.	1. Interface is defined with the help of interface keyword. It can contain only abstract methods.

2. It does not support multiple inheritance.	2. Multiple inheritance can be achieved (implemented) using interface.
3. In abstract class, you can extend java class and implement many java interfaces.	3. Only one java interface can be extended.
4. Members of abstract class can be public, private, and protected etc.	4. Member of java interface are by default public.

Let's have a look on the program given below, in this program, you can see that Payee is an interface and SalaryEmployee, and CommissionEmployee are the two class. These classes implement the interface Payee.

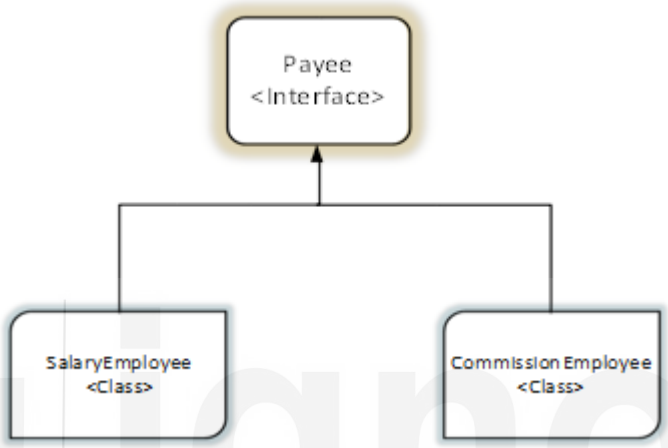


Figure 6.5: A scenario of interface implementation

```
/*-------------------------------Payee Interface-----------------------------------*/

package Jtest;
public interface Payee
{
   String Ename();
   Double EgrossPayment();
   Integer EbankAccount();
}
```

The class PaymentSystem is implemented to store the payee's information with the help of ArrayList.

```
/*-------------Payment System class----------------------*/
package Jtest;

import java.awt.List;
import java.util.ArrayList;

public class PaymentSystem
{
   private ArrayList<Object> payees;
   public PaymentSystem()
   {
      payees = new ArrayList<>();
   }
   public void addPayee(Payee payee)
   {
      if(!payees.contains(payee))
      {
         payees.add(payee);
      }
   }
```

```
  public void processPayments()
  {
    for(Object payee: payees)
    {
      Double grossPayment = ((Payee) payee).EgrossPayment();
      System.out.println("Paying to"+((Payee) payee).Ename());
      System.out.println("tGrosst"+grossPayment);
      System.out.println("tTransferred to account:"+ ((Payee)
      payee).EbankAccount());
    }
  }
}
```

The SalaryEmployee class is to set the name, bank account details and gross wage of the employee by implementing the Payee interface.

```
/*----------------SalaryEmployee class-------------------------*/
package Jtest;

public class SalaryEmployee implements Payee
{
  private String name;
  private Integer bankAccount;
  protected Double grossWage;
  public SalaryEmployee (String name, Integer bankAccount, Double grossWage)
  {
    this.name = name;
    this.bankAccount = bankAccount;
    this.grossWage = grossWage;
  }
  public Integer EbankAccount()
  {
    return bankAccount;
  }
  public String Ename()
  {
    return name;
  }
  public Double EgrossPayment()

  {
    return grossWage;
  }
}
```

The CommissionEmployee class sets the information of the employee by implementing the Payee interface. It has also implemented some addition methods which are doCurrentCommission and giveCommission.

```
/*--------------------CommissionEmployee Class----------------------*/
package Jtest;

public class CommissionEmployee implements Payee
{
  private String name;
  private Integer bankAccount;
  protected Double grossWage;
  private Double grossCommission = 0.0;
  public CommissionEmployee(String name, Integer bankAccount, Double grossWage)
```

```
  {
    this.name = name;
    this.bankAccount = bankAccount;
    this.grossWage = grossWage;
  }
   public Integer EbankAccount()
  {
    return bankAccount;
  }
   public String Ename()
  {
    return name;
  }

   public Double EgrossPayment()
  {
    return grossWage + doCurrentCommission();
  }

  public Double doCurrentCommission()
  {
    Double commission = grossCommission;
    grossCommission = 0.0;
    return commission;
  }
   public void giveCommission (Double amount)
  {
    grossCommission +=amount;
  }
}
```

In the above two programmes, we can see that you can see that SalaryEmployee and CommissionEmployee abstract methods are implemented again and again because the classes that are using the Payee interface, they have to define the Payee methods.

```
/*------------Employee class---------------------*/
package Jtest;

public abstract class Employee implements Payee
{
   private String name;
   private Integer bankAccount;
   protected Double grossWage;
   public Employee(String name, Integer bankAccount, Double grossWage)
  {
    this.name = name;
    this.bankAccount = bankAccount;
    this.grossWage = grossWage;
  }
   public String name()
  {
    return name;
  }
   public Integer bankAccount()
  {
    return bankAccount;
  }
}
```

```
/*--------------Main PaymentApplication class----------------------------*/
package Jtest;

public class PaymentApplication
{
  public static void main(final String []args)
  {
  PaymentSystempaymentSystem = new PaymentSystem();
  CommissionEmployeeraviSingh = new CommissionEmployee("Ravi Singh", 5000, 301.0);
  paymentSystem.addPayee(raviSingh);
  CommissionEmployeesksingh = new CommissionEmployee("Sunil Singh", 6000, 545.0);
  paymentSystem.addPayee(sksingh);
  SalaryEmployeemaryBrown = new SalaryEmployee("Mary Brown", 7000, 500.0);
  paymentSystem.addPayee(maryBrown);
  SalaryEmployeesusanWhite = new SalaryEmployee("Susan White", 8000, 555.0);
  paymentSystem.addPayee(susanWhite);

  raviSingh.giveCommission(40.0);
  raviSingh.giveCommission(35.0);
  raviSingh.giveCommission(45.0);

  sksingh.giveCommission(50.0);
  sksingh.giveCommission(51.0);
  sksingh.giveCommission(23.0);
  sksingh.giveCommission(14.5);
  sksingh.giveCommission(57.3);

  paymentSystem.processPayments();
   }
}
```

You can see the output of the program given below also visualize the implementation of abstract methods in the classes.

Output

```
Paying toRavi Singh
tGrosst421.0
tTransferred to account:5000
Paying toSunil Singh
tGrosst740.8
tTransferred to account:6000
Paying toMary Brown
tGrosst500.0
tTransferred to account:7000
Paying toSusan White
tGrosst555.0
tTransferred to account:8000
```

Now look at the scenario given in the figure 6.6,

```
/*---------------Modified classes to reduce the code duplicity--------------*/
```

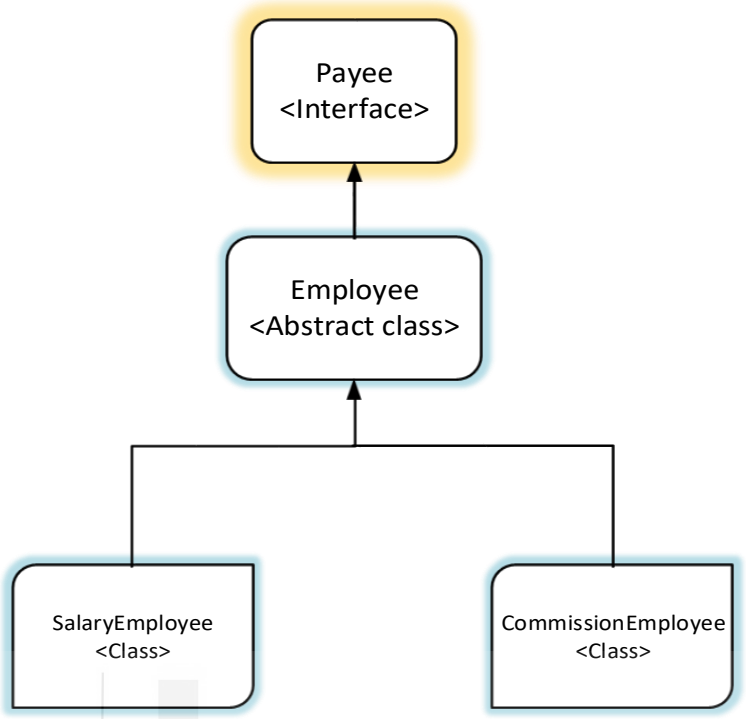


Figure 6.6: A Scenario to interface class

☞ Check Your Progress-2

1) Why do we use polymorphism in Java?

.....

.....

.....

2) What do you mean by polymorphic parameters? Explain it with the help of suitable program.

.....

.....

.....

3) Find the answers to the program given below.

It return type, method name and argument list is the same.

```
class Demo
{
  public int myMethod(int num1, int num2)
  {
System.out.println("First myMethod of class Demo");
    return num1+num2;
  }
  public int myMethod(int var1, int var2)
  {
System.out.println("Second myMethod of class Demo");
    return var1-var2;
  }
}
class Sample4
{
  public static void main(String args[])
  {
```

```
    Demo obj1= new Demo();
    obj1.myMethod(10,10);
    obj1.myMethod(20,12);
  }
}
```

---

4) Find the answers of the program given below.

Its return type is different, while mehod name and argument list is same.

```
class Demo2
{
  public double myMethod(int num1, int num2)
  {
System.out.println("First myMethod of class Demo");
    return num1+num2;
  }
  public int myMethod(int var1, int var2)
  {
System.out.println("Second myMethod of class Demo");
    return var1-var2;
  }
}
class Sample5
{
  public static void main(String args[])
  {
    Demo2 obj2= new Demo2();
    obj2.myMethod(10,10);
    obj2.myMethod(20,12);
  }
}
```

6.9 SUMMARY

In this unit, we discussed the important concept known as polymorphism. This unit also discussed the types of polymorphism. The concepts of overloading and overriding have been explained with the help of example programs. Property inheritance is discussed with the help of extends and implements keywords. Furthermore, in the end, we talked about, code reusability concept using abstract class instead of using the implement method of interface.

6.10 SOLUTION/ANSWER TO CHECK YOUR PROGRESS

Check Your Progress – 1

1) Whenever several methods have the same names with:

- Different method signatures and different numbers or types of parameters.
- Same method signature but the different number of parameters.
- Same method signature and the same number of parameters but of a different data type.

It is determined at the compile time.

2) Compile-Time Polymorphism: The best example of compile-time or static polymorphism is the method overloading. Whenever an object is bound with their functionality at compile time is known as compile-time or static polymorphism in java.

3) The class Summation given below have the three methods with the same name sum.

```
public class Summation
{
public int SUM(int m , int n)
{
     return (m + n);
}
public int SUM(int m , int n , int p)
{
     return (m + n + p) ;
}
public double SUM(double m , double n)
{
     return (m + n);
}
public static void main( Stringargs[])
{
Summationob = new Summation();
ob.SUM(15,25);
ob.SUM(15,25,35);
ob.SUM(11.5 , 22.5);
}
}
```

Check You Progress – 2

1) Polymorphism in Java makes possible to write a method that can correctly process lots of different types of functionalities that have the same name. We can also gain consistency in our code by using polymorphism.

The list of advanatages of polymorphism are as follows.

It provides reusability to the code. The classes that are written, tested and implemented can be reused multiple times. This, in turn, saves a lot of time for the coder. Also, the code can be changed without affecting the original code.

A single variable can be used to store multiple data values. The value of a variable inherited from the superclass into the subclass can be changed without

changing that variable's value in the superclass or any other subclasses.

With lesser lines of code, it becomes easier for the programmer to debug the code.

2) Parametric polymorphism allows a name of a parameter or method in a class to be associated with different types. Lets have look at the example given below, in which *content* is defined as string once then after that defined as *Integer*.

```
public class Department extends Employee
{
private String content;
public String setContentDelimiter()
{
int content = 100;
this.content = this.content + content;
}
}
```

Here, the local declaration of a parameter always overrides the global declaration of another parameter with the same name. To handle this issue, it is advisable to use global references such as this keyword to indicate the global variables within a local context.

Problem with the polymorphic parameter is, it suffers from variable hiding.

3) It will throw a compile time error because more than one method with the same name and arugument list cannot be defined in a same class.

4) Run this program you will observe that it will throw a compilation error, because you cannot have more than one method with the same name and argument list in a class even though their return type is different. The point to note is that the method return type does not matter in case of overloading.

6.11 REFERENCES/FURTHER READINGS

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Savitch, Walter, " Java: An introduction to problem solving & programming", Pearson Education Limited, 2019.
- Neil, O. , "Teach yourself JAVA", Tata McGraw-Hill Education, 1999.
- Sarcar, Vaskaran. "The Concept of Inheritance", In Interactive Object-Oriented Programming in Java, pp. 65-90. Apress, Berkeley, CA, 2020.

UNIT 7 ASSERTIONS, ANNOTATIONS AND EXCEPTION HANDLING

Structure **Page No.**

7.0 Introduction 156

7.1 Objectives 157

7.2 Assertion and its use 157

7.3 Annotations 163

7.3.1 Built-In Java Annotations 164

7.4 Exception Handling 169

7.4.1 Checked and Unchecked Exceptions 171

7.4.2 Using try, catch and finally method to handle exception 173

7.5 Throw and Throws 177

7.6 Final vs Finally vs Finalize 179

7.7 User Defined Exception 179

7.8 Summary 183

7.9 Solutions/ Answer to Check Your Progress 183

7.10 References/Further Reading 185

7.0 INTRODUCTION

Java is known for its robustness, which comes from its strong exceptions handling feature. In this unit we will discuss Assertions, Annotations and Exception Handling. Assertions are infrequently used, but they play a vital role in testing and verifying various assumptions which developers make while writing their code based on the RSD/TSD (Requirement/Technical Specification Document) which they receive from Architect team. First, we will try to understand what benefits usage of assertions brings. The next part in this unit touches upon the usage of annotations, which helps us to maintain the documentation and changes of our code during the life-cycle of our code with respect to the changes that have happened overtime in the language itself. In the third and final part of this unit we will discuss at length one of the reasons why java has been perceived as such a robust language, as it not only had vast constructs to handle normal flow of program but also a completely structured methodology to handle even the exceptions which might occur. We will study Exception Handling in Java, in detail.

At the end of this unit you will be able to define and use annotations in Java. Also you should be able to explain what assertions are, where are they used and what advantages they bring in. Then you will go through importance of metadata and significance of writing meaningful comments in your program. This will underline the fact that documentation for any program is very important. It is so important that a language like java had to introduce a new construct which will help it easily and effectively maintain documentation using annotations. Also you will learn about exception handling in Java. We will go through various types of exceptions, the exceptions hierarchy, distinguish between different types of exceptions and study in detail about Runtime and IOExceptions. You will appreciate how apt and wide coverage of exceptions is provided in java. Towards the end you will learn to define your own Exception.

7.1 OBJECTIVES

After going through this unit we will be able to :

- Define assertions,
- Differentiate Assertions and Exception Handling,
- Explain usage and Advantages of Assertions,
- Describe Annotations, Annotation types and Metadata,
- Use Annotations,
- Explain Exception handling,
- Differentiate between errors and exceptions,
- Describe Exception hierarchy,
- Describe types of Exceptions,
- Show difference between throw and throws,
- Use final finally and finalize keywords, and
- Define your own exception methods.

7.2 ASSERTIONS AND ITS USE

We all would have heard about the role of proofreaders in the publishing industry. The main objective of proofreading a document is to ensure that none of the mistakes/undesired statements actually reach the print or the printing stage. Effectively this is the task in which assertions help us perform over our programs . Now let us understand how we use assertions in Java.

The meaning of the term assert stated in dictionary is to **'declare positively'**. But then, is not it all that we do when we write our programs? Then what is the need to have a specific statement for this purpose and if it is there, how should we use it ? What is the applicability and, even more importantly, where we *should not* apply it? We will also study the difference between using an assertion statement versus a more commonly known procedure of **exception handling**.

Let's start with a question. How often do you write a code that will not go to production or final use? Well, companies hire and pay programmers to write code they can use in production, don't they? Then why will anybody write a code that will not go to production and, more importantly, gets paid for something that will not go to production? It sounds dubious, does'nt it?

Well, let's try and understand this using another simple situation. During the normal course of execution of a program if a piece of code or condition is *unreachable* it should be tested beforehand. This is where an *assert a statement* comes into picture. Effectively the Java assert statement is meant to be used in **non production environment.** In fact, it is one of those rare statements which are specifically made for usage in non-production environments only, but how? To be more precise assertions are by default disabled in any environment and have to be ***explicitly enabled***, in order for themto be used. In other words, to utilize assertions in any environment you need to enable it, and in production, as a recommended practice, we do not enable assertions. Simply put, when assertions are enabled they can be utilized to detect code issues i.e. only if there is a bug in a program then only an assert statement will be triggered.

So only an occurrence of an **extraordinary condition** will lead Java assert to trigger, resulting in an **Assertion Error** to be thrown. The execution of Assertion Error will thus prevent a Java application to reach an otherwise unreachable code and ***terminate abruptly.*** As an add-on, it also provides the provision to display a meaningful message if the extraordinary condition has been encountered in Java application.

So why is Java assert so different or unique? To answer this, first, let us understand these two points :

1. Firstly, the use of *assert keyword* is handy in a test environment, or for **temporarily** performing a difficult debugging routine on production systems (though debugging and troubleshooting your code in production is not a recommended approach to take). There is no other keyword in Java that is specifically targeted towards testing and debugging the Java code.

2. Secondly, the Java assert keyword is ignored by default (not processed when the Java virtual machine(JVM) is run using its default configuration).

To use assertion a special flag namely **enable assertions** is passed as a command line parameter at runtime to the JVM, which signifies the code associated with the Java assert to be executed. Uniquely enough, no other keyword in Java language is disabled by default at runtime except our assert.

To learn these topics, we will take a hands-on approach for:

- Enabling assertions in Java
- Disabling assertions

Also, you need to learn:

- Where to utilize assertion and
- Where not to utilize assertion

Let’s take the situation below and see how we can enable and disable assertions in java.

We make a lot of assumptions while writing our programs, a Java assertion helps us to check whether the assumptions which we made during the writing of our code, were correct or not. It sounds like we are doing some sort of testing, testing of our own code. Well, you got it exactly right, Java assert is actually a helpful tool that is used for troubleshooting applications during and the development and testing phase of SDLC. The main purpose of using assertions is to test for conditions that should never be evaluated during the normal course of execution of our code. In case if such conditions did occur, an error message should be generated and the application terminated after throwing an AssertionError.

There are two different syntax usage of Java asserts statement either with a single Boolean articulation or boolean expression and execution statement thereof.

- Assert expression;
- Assert expression1: expression2;

where:

- Expression1 is a boolean expression.
- Expression2 is an expression that can either be a simple statement or has some value (baring invocation of a method that is declared void.)

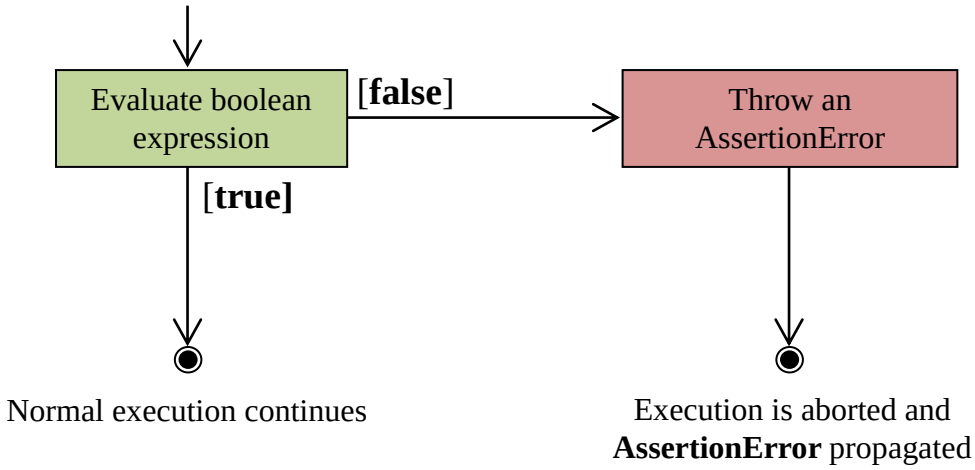


Figure 7.1: Assertion Execution

Following is the syntax used for a Java assert :

assert :< boolean condition > : < string message for logging>

Another variation and more meaningful usage syntax is Java assert, which will trigger and log/display/print a string message to the Java console as follows:

assert : (true == true) : " OOPS!!! Something bad just happened.";

Let us take a situation in which if a person going in for RTPCR test for testing of Covid19 will be declared positive or negative based on the Cycle Threshold(CT) value. If the CT value is greater than 29(any random value not medically correct), the person will be declared as Covid positive.

Below is a simple example program:

```
class CovidTest
{
        public static void main( String args[] )
        {
                int RTPCRvalue = 31;
                assert value >= 29 : " Covid-19 Positive";
                System.out.println("value is "+ RTPCRvalue);
        }
}
```

Compile and run the above Java program. What is the output?

Exception in thread "main" java.lang.AssertionError: Covid-19 Positive
at Test.main(Test.java:8)

If you received the above output that means assertions are enabled on your system and if not,then let us learn how to enable or disable assertions.

Enabling and Disabling Assertions in Java

The syntax for the Enabling assertions is-

```
java –ea Test
```

or

```
java –enableassertions Test
```

The syntax for disabling Java Assert-

```
        java –da Test
or
        java –disableassertions Test
```

There are many variations for enabling assertions using the command line.

#1) java –ea
When we issue the above command at command-line, then the assertions are enabled in all classes except for system classes.

#2) java –ea Main
The above command is used to enable assertion for all classes in the Main program.

#3) java –ea DemoClass Main
This command enables assertions for only one class – 'DemoClass' in the Main program.

#4) java –ea com.packageName… Main

The above command enables assertion for package com.packageName and its sub-packages in the Main program.

#5) java –ea … Main
Enables assertion for the unnamed package ,which is in the current working directory.

#6) java –esa: arguments OR java –enablesystemassertions: arguments
This command enables assertions for the system classes.

To add the assertions, we have to simply add an assert statement :

Imagine a situation where Bluetooth enabled app on the phone establishes connections with nearby mobile phones, and if a connection is established to a phone of infectious person whose data is updated in app or there are people who are at high risk of COVID-19 (again based on data in app which you're sharing) in your proximity, then you would like it to flash a message "proximity to infection".

```
public void setup_connetion ()
{
    Connection conn = getConnection ();
    assert conn == null;
}
```

The above assert, also canbe given differently as shown below:

```
public void setup_connection ()
{
    Connection conn = getConnection ();
    assert conn == null: "Proximity to infection";
}
```

Both of the above code constructs check if the connection returns a null value. If it returns a non-null value, then JVM will throw an error – AssertionError. But you can see, in the second case, a message is provided in the assert statement so that the given message will be used to construct AssertionError. When the second case with assertions enabled,is executed, the exception will look like this:

Exception in thread "main" java.lang.AssertionError: Proximity to infection.

Why Use Java Assertions?

When a programmer needs to check if his/her assumptions are right or not.

- Pass on arguments to private methods. Developers provide private arguments to check their code, and developers may also want to check his/her own assumptions about the arguments.
- Cases that are conditional
- To evaluate conditions at the beginning of any method.

Significance of Assertions on Performance

Developers have an interesting motive in using assertions as well. Consider a possibility when using a conditional statement to detect an issue, developers write these statements and then forgets to remove them from code once their purpose (of detection of issue) is solved, this might result in a performance bottleneck where several CPU cycles are wasted if they are not removed from code before shipping to production. This (addition and removal of temporary code for debugging) is a tedious task to do and involves multiple teams. To understand this, consider a situation where a team is following *extreme programming* principles and programmer 'A' submits his/her code for testing in which he/she utilizes conditional statements to check bugs/assumptions. In case the bugs are identified and corrected or the assumptions are dumbfounded, they will still remain in the code and further released to the next environment and also to production. Unless these are explicitly removed (which is a sort of rework again and thus inefficient), since these lines were only intended for development and testing, once they reach production, they will consume precious processor cycles needlessly as the conditions will still be getting evaluated even after the bug had been removed.

Now compare this to what happens in the case of when the developer uses assertions. As you know that the default behavior of JVM is to ignore and assert statement at runtime the piece of code with assertion statements thus will never get executed and save precious processor cycles. All the assert statements whose purpose was to troubleshoot are thus ignored (rememberassertions are disabled by default in every java environment), and the precious Clock cycles are saved.

Now we know that there is a performance benefit for using Java assert statement rather than the programmers writing conditional statements for the same purpose.

Where to Utilize Assertion in Java :

Java Assert can be utilisedin the following ways;

- To ensure that an inaccessible looking code is really inaccessible.
- Java Assertions of private techniques. Private contentions are given by designer's code just as a developer might need to check his/her assumptions about contentions.
- In Restrictive cases.
- Conditions toward the start of any technique.
- To verify that the 'default' switch case is not reached.
- Ensure that presumptions written in remarks are correct.
- Check the protest's state.
- At the start of the strategy.
- After strategy summon.

Where not to use Assertions :

- Don’t use assertions to replace error messages
- Don’t use assertions to check arguments in the public methods as they may be provided by user.
- Don’t use assertions on command line arguments.
- You should use Error handling to handle errors where user input is sought instead of Assertions.

Assertions vs Exception Handling

Assertions are primarily used to check logically impossible situations or inconceivable circumstances. For instance, assertions can be used to check the expression or state that a piece of code expects before it starts execution (pre-condition) or a state that it expects at the end or completion of its execution (post-condition). Unlike the normal exception handling constructs in Java, assertions are generally disabled at runtime.

Two examples of common usage scenarios for assertions

1. Unreachable code
2. Documenting Assumptions

Example :

```
switch (dayOfWeek)
{
  case "Sunday":
    System.out.println("It’s Sunday!");
    break;
  case "Monday":
    System.out.println("It’s Monday!");
    break;
  case "Tuesday":
    System.out.println("It’s Tuesday!");
    break;
  case "Wednesday":
    System.out.println("It’s Wednesday!");
    break;
  case "Thursday":
    System.out.println("It’s Thursday!");
    break;
  case "Friday":
    System.out.println("It’s Friday!");
    break;
  case "Saturday":
    System.out.println("It’s Saturday!");
    break;
}
```

The above switch statement indicates that the days of the week can be only one of the above 7 values. Having no default case means that the programmer believes that one of these cases will always be executed, , which logically is true in our example, but if an invalid input is received, the assumption might prove wrong. Thus, we try adding a default case to document assumption while still maintaining that the default statement is logically Unreachable for all valid reasons.

```
default:
   assert false: dayofWeek + " is invalid day";
```

If day Of Week – the parameter passed to switch-case has a value other than the valid days, an Assertion Error is thrown.

☞ **Check your progress -1**

1) What is the default state of assertions in Java (Enabled or Disabled)? How can we enable or disable assertions in Java?

.....

.....

.....

2) Does assert throw an exception Java?

.....

.....

.....

3) What happens when an assert fails in Java?

.....

.....

.....

4) What does an assert return in Java?

.....

.....

.....

5) Can we catch the assertion error?

.....

.....

.....

6) How do you assert an exception?

.....

.....

.....

7.3 ANNOTATIONS

Data that describes or holds additional information about your data is called Metadata, and that is what Annotations are. Annotations are a form of metadata. Here in our case, while studying Java, Annotations provide data about a program/code that itself is not part of the program. Since Annotations do not directly impact the functioning of the code, they still serve an essential purpose when embedded in code/program they annotate – they provide documentation to the code.

Though Annotations are a relatively new addition to Java , They make our life very easy in terms of our documentation – without putting too many comments, suggestions, notifying other programmers/users not to use a function etc. Following are the significant uses of annotations, including:

- Providing information to the compiler - The Java compiler uses Annotations to suppress warnings and detect errors.
- Compile time and delivery time handling - We can use Annotations XML files, etc., generated with some software tools.
- Runtime processing - some types of Annotation can be evaluated at runtime.

Let us try tofind answer of some frequently asked questions like applying Annotation; where do we use annotations? What types of annotations can we use in Java? Are there predefined types of Annotation? Can we write our own Annotations in Java? What kind of Annotations can be used in combination with pluggable type systems to write robust code with stronger type checking, and how we implements repeated Annotations.

7.3.1 Built-In Java Annotations

There are different types of built in associate annotations in Java. One Annotation category is applied to Java code and the second category annotates the first category.

Built-In Java Annotations applied to Java code :
- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations Applied to other Java annotations :
- @Target
- @Retention
- @Inherited
- @Documented

Let us understand the built-in annotations first, as they are very simple and straightforward.

@ Override

Inheritance is a fundamental function of any object-oriented programming language that we generally use to infer the properties of the parent unchanged and to override or change some of them. The @Override annotation helps us determine that the derived class method is overriding its main class method. In case it does not, the @Override annotation ensures that a compile-time error occurs.

It can be a silly mistake, such as a spelling mistake, incorrect capitalization (remember that Java is a case-sensitive language), or a simple typo. Therefore, it is recommended to mark the activation override annotation that provides the assurance that a method is overridden.

Example:

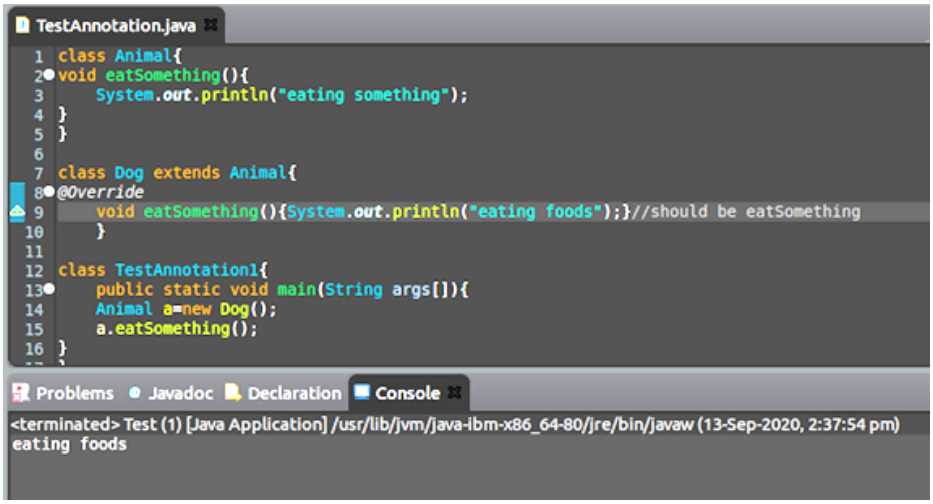
```
class Animal
{
void eatSomething()
{
System.out.println("eating something");
}
}
```

```
class Dog extends Animal
{
@Override
void eatsomething()
{
System.out.println("eating foods");
}//should be eatSomething
}
class TestAnnotation1
{
public static void main(String args[])
{
Animal a=new Dog();
a.eatSomething();
}
}
```

Output when there is a spelling mistake:
Replace
Output once the correction is made :

Output when there is a spelling mistake:

Output once the correction is made :



Following is another example of use of @override

Example:

```
class ViralTest
{
     public void conductTest()
     {
        System.out.println("This is test for presence of Viral Infection.");
     }
}

class covid19Test extends ViralTest
{
    @Override
    public void conductTest()
    {
       System.out.println("This is test to detect presence of Covid19 Infection.");
    }
}

class Main
{
    public static void main(String[] args)
    {
        covid19Test c1 = new covid19Test();
        c1.conductTest();
    }
}
```

Output

In this example, the method conductTest() is present in both the superclass ViralTest and subclass covid19Test. When the method conductTest() is called, the method of the subclass covid19Test is called instead of the method in the superclass ViralTest.

@SuppressWarnings

@SuppressWarnings annotation: This annotation is used for suppressing the warnings issued by the compiler.

```
import java.util.*;
class VerifyAnnotation2
{
  //@SuppressWarnings("This will suppress compiler warnings")
  public static void main(String args[])
  {
    ArrayList list=new ArrayList();
    list.add("sonoo");
    list.add("vimal");
    list.add("ratan");
    for(Object obj:list)
    System.out.println(obj);
  }
}
```

Output: Comple Time Error

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;
class TestAnnotation2
{
  @SuppressWarnings("This will suppress warnings")
  public static void main(String args[])
  {
    ArrayList list=new ArrayList();
    list.add("Ravi");
    list.add("Vimal");
    list.add("Rameshwar");
    for(Object obj:list)
    System.out.println(obj);
  }
}
```

Now no warning at compile time.

Focus on the statement @SuppressWarnings("This will suppress compiler warnings ") which is an annotation, If you remove it , it will result in a compile time warning as we are using non-generic collection.

@Deprecated

Languages evolve and so does the validity of the keywords and features which they offer some new keywords come to usage and few older ones are @Deprecated. Deprecated means that a feature or a keyword may see sunset and thus can be removed in the future versions/releases of the language. @Deprecated annoation helps us mark the same to a method . Once the compiler detects a deprecated method it prints a warning informing user that it may be removed in the future versions, therefor avoid use of such methods.

Example

```
class Mycode
{
	/*
	-> @deprecated
	-> This method is deprecated and has been replaced by newMethod()
	*/
	@Deprecated
	public static void deprecatedMethod()
	{
		System.out.println("This is a deprecated method");
	}

	public static void main(String args[])
	{
		deprecatedMethod();
	}
}
```

Output :

Deprecated method

Example

```
class A
{
    void m(){System.out.println("hello m");}
    @Deprecated
    void n(){System.out.println("hello n");}
}
class TestAnnotation3
{
    public static void main(String args[])
    {
        A a=new A();
        a.n();
    }
}
```

Now, at the time of compilation :

Note: Test.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

Where as when you execute/run it :

Output : hello n

Java Custom Annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

1. @interface MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

☞ Check Your Progress - 2

1) Describe some basic Annotations from the Standard Library?

……..

……..

……..

2) Which Java package is used for Annotations ?

……..

……..

……..

3) Can you create your own annotation – Describe how?

……..

……..

……..

4) Are Annotations inherited in Java? If not, what can be done so that they are inherited?

……..

……..

……..

5) Annotation class is extended from which class in Java?

……..

……..

……..

6) What is the package name for Annotation class?

……..

……..

……..

7) What are meta-annotations? Name a few.

……..

……..

……..

7.4 EXCEPTION HANDLING

In simple words, "an exception is a problem that arises during the execution of a program". It can occur for many different reasons to exemplify a few:

- say a user has entered an invalid data or
- in case of a file that needs to be opened cannot be found or
- a network connection that has been lost in the middle of communications or
- the JVM has run out of memory.

Many such exceptional cases may occur while executing Java programs. If we as programmers do not handle them, it leads to a system failure. So handling an exception is very important, and that is where Java introduces the feature of exception handling so that the normal execution flow will not be abruptly terminated in case of occurrence of exception. Mechanism of exception handling provides an opportunity for a graceful exit from the program in case of the occurrence of an exception. For example, if exception conditions such as 'Classnotfound', IOException, SQLexception etc. occur, then by appropriate exception handling, users may be informed about such situations, and program execution can be closed gracefully.

Before we learn more about exception handling, let us see the difference between error and exception.

- Errors are impossible to recover, but exception can be recovered by handling them.
- Errors are of type unchecked, but exceptions can be either checked or unchecked. (you will learn checked and unchecked types of exceptions later in this unit)

- Errors are something that happened at runtime but exception can occur or happen either at compile time or runtime.
- Exceptions are caused by the application itself, whereas errors are caused by the environment on which the application is running.

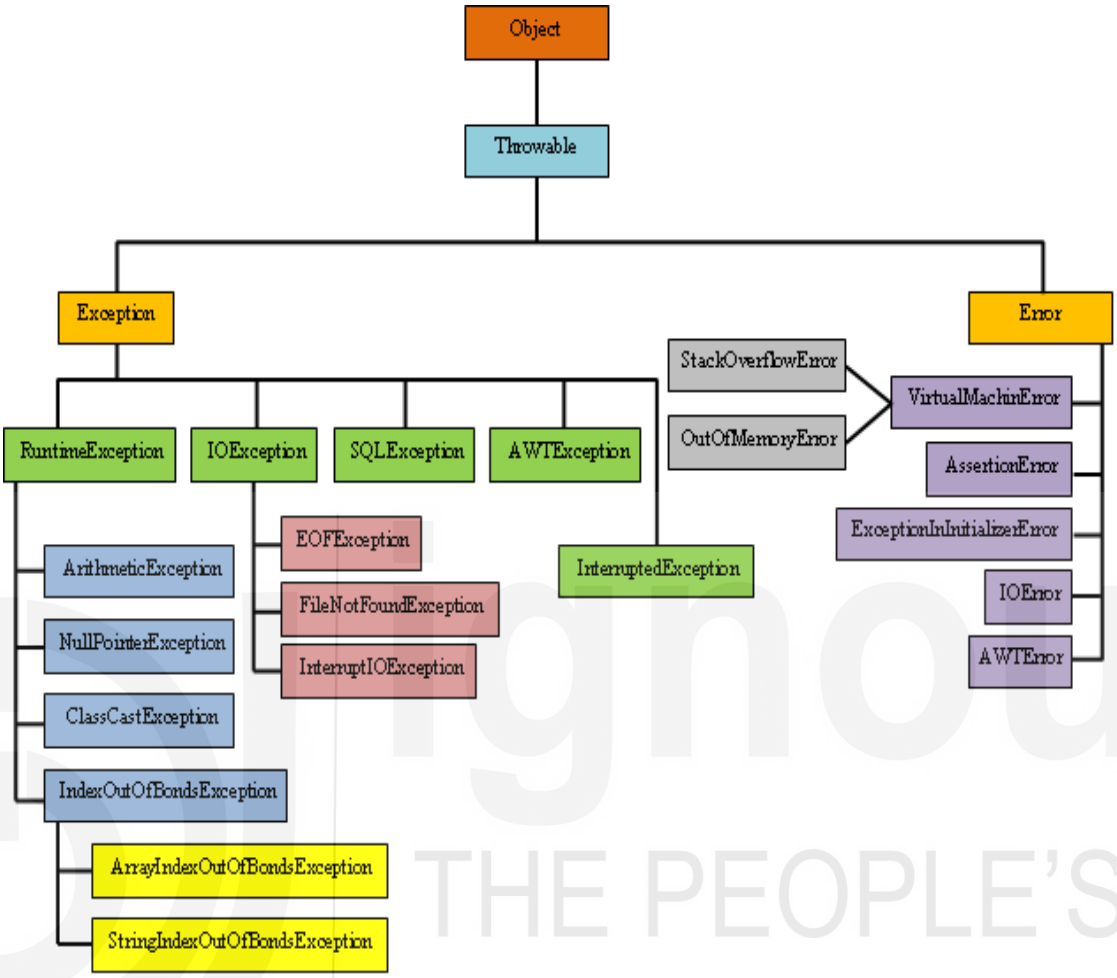


Figure 7.2: Throwable Class Hierarchy in Java

Now let us see what is exception hierarchy in Java. All exceptions and errors types are subclasses of **class Throwable,** which is the base class of the hierarchy. The **Object class** is the parent **class** of all the **classes in Java** by default, and class Throwable is a subclass of Object class. Here one branch is headed by exception, that is, this class is used for exceptional conditions that the user program should catch. For example, NullPointerException, RuntimeException etc. and other branch Error are used by the Java runtime system to indicate the errors that have to do with the runtime environment itself, that is JRE. For example, VirtualMachine error or StackOverflow error etc. Figure 7.2 show the Throwable class and its subclasses.

Exception class and its Subclasses in Throwable class Hierarchy

Here one important and interesting question arises that how does Java virtual machine handle exceptions whenever an exception has occurred inside a method?

The answer to this question is that the method creates an object known as **exception object** and hands it off to the runtime system. This exception object contains name and description of the exception and also the current state of the program in which the exception has occurred. The process of creating the exception object and handling it to the runtime system is called **'throw'ing** an exception, then byusing **try, catch and**

finally clauses these exceptions can be handled in Java. This is how Java virtual machine handle exceptions internally.

In figure 7.3 Exception class and its sub classes are shown. For more details and current updates you can see the current documentation of Java. Having a good understanding of sub classes of exception class is essential for better programming in respect of exceptions handling in Java.

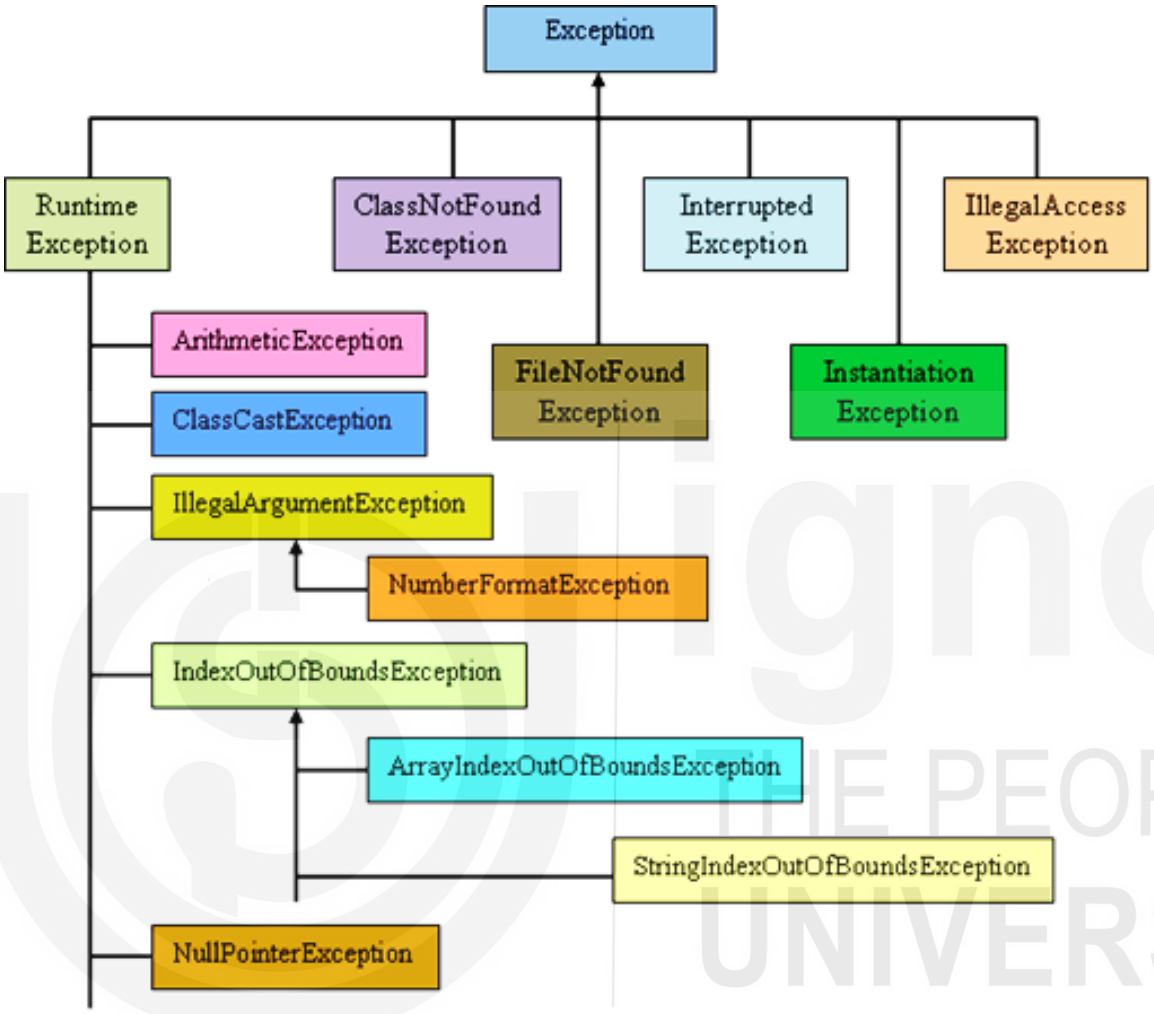


Figure 7.3: Exception Class and its subclasses

7.4.1 Checked and Unchecked Exceptions

In simple terms, exceptions that happen at compile time are **Checked Exceptions**. That is,the compiler checks such exceptions at the time of compilation. The checked exceptions cannot be ignored; the programmer should handle these exceptions.In other words,a checked exception must be either caught or declared in the method in which it is thrown. If we donot write the code to handle them, the compiler gives an error. On the other hand, the exceptions that occur at the time of execution, like IOExceptions etc, are ignored at the time of compilation are called **Unchecked Exceptions**. These Unchecked exceptions are also called Runtime exceptions. Examples of unchecked exceptions are: divide by zero, array out of bounds, null pointer exception etc. The unchecked exceptions are built-in exceptions in Java. By good programming practice, many a time, unchecked exceptions can be avoided.

For exception handling in Java a basic format of programming is used. Now let us see it through a program code.

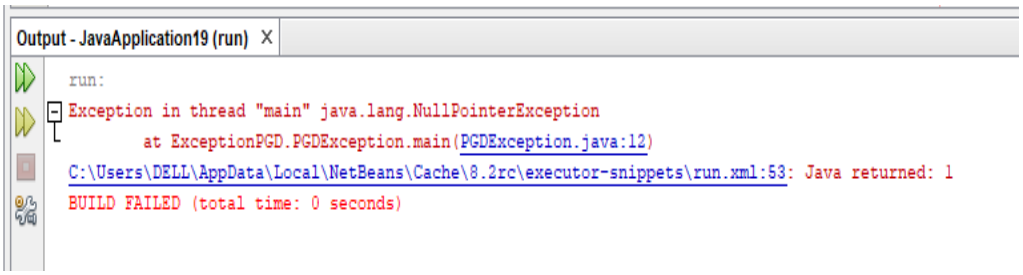
```
class Exception
{
  public static void main(String args[])
  {
    try
    {
      //code that may raise an exception
    }
    catch(Exception e)
    {

    }
    //Rest of the program
  }
}
```

Understanding the basic structural construct example of exception. Here we have defined the class and inside the main method we have a **try block**, in this try block we are going to write code that will raise exception or it contains the code that may raise exception and then the raised exception will be handled in the **catch block**. Now through a small program, let us see how the exception is handled once it occurs.

First, we will create a new package called ExceptionPGD, and then we will create a new class called PGDException. Inside the class the first thing that we are going to do is to write the main method, so here we'll create string str and make it "NULL". After this we will try to retrieve the length of the string. Notice that we are setting string to "NULL" intentionally, so when we execute this program, it will throw an exception, why, because the string is null and we are trying to retrieve the length of the string. Here you will find that it throws null pointer exception because the string is not there or it does not contain any values. Well, this is what we wanted it to do, throw an exception.

```
package ExceptionPGD;
public class PGDException
{
        public static void main(String args[])
        {
                string str = null;
                system.out.println(str.length( ));
        }
}
```

Output:



As we said it throws NullPointerException because the string is null, it does not contain any values. With this understanding of exception, let us begin our journey of understanding how to handle exceptions, with the help of another program.

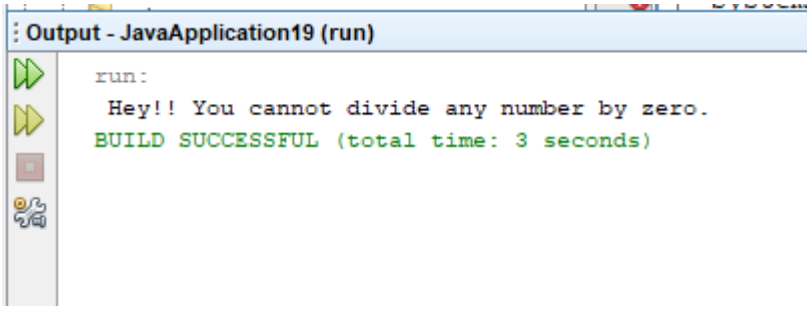
7.4.2 Using try, catch and finally method to handle exception

We have already seen the construct of an exception handling program earlier in this chapter. Now we will put it to practice, and to do that here in try block we are going to write the code that will raise exception, this time an ArithmeticException. Just to get the basic rule of maths right. Do you remember if we try to divide a number by zero, assuming both a and c are integer variables and we try, c=a/0 , what happens here ? If we try to divide a number by zero then it says we cannot divide the number by zero because it raises arithmetic exception a base arithmetic anomaly.

Now as it throws an exception and we saw in our last example that the program exited after throwing an exception. But we don't want to exit if an exception occurs. So what should we do to maintain the normal flow of execution ? Well, this is where we write the catch block to handle this exception. Let us run the program and see the output.

```
package ExceptionPGD;

public class PGDException
{
  public static void main(String args[])
  {
    try
    {
      int a=90, b=0;
      int c=a/b;
      system.out.println("Resultant =" +c)
    }
    catch(ArithmeticException e)
    {
      System.out.println(" Hey!! You cannot divide any number by zero.");
    }
  }
}
```

Output:



As explained above, the exception has been 'caught' by catch block so here first when the execution is started an exception is thrown and then the thrown exception is caught by the code of catch block which handles arithmetic exception and prints "Hey !! You cannot divide a number by zero".

This is how basic exception handling works. Now let us learn various types of exceptions in Java. In Java there are two types of exceptions built-in exceptions and user defined exceptions. First let us see what are built-in exceptions.

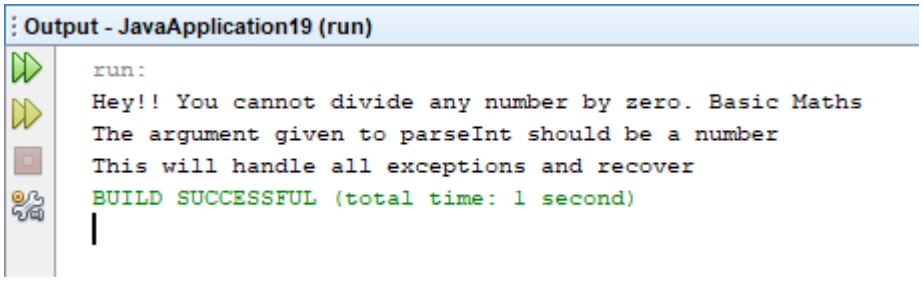
Built-in exceptions are the exceptions which are available in Java libraries and they are suitable to explain certain error situations like arithmetic exception, array index out of bound and class not found exception, runtime exception, number format exception etc.

Various methods of exceptions handling are try, catch, finally, throw and throws. Two commonly used exception handling concepts are throw and throws. Only single exception is **thrown** by using **throw method**. Multiple exceptions can be **thrown** by using **throws** method. We have already seen try block which is used to enclose the code that may throw exception and catch block which will handle the thrown exception. Syntax as you have already seen is very simple. We write a code that throws exception and we can handle using the catch block next. Now let us try nested try block, which is nothing but a try within a try block.

```
package ExceptionPGD;
public class PGDException
{
  public static void main(String args[])
  {
    try
    {
      int a=90, b=0;
      int c=a/b;
      System.out.println("Resultant =" +c)
    }
    catch(ArithmeticException e)
    {
      System.out.println(" Hey!! You cannot divide any number by zero. Basic Maths");
    }
    try
    {
      try
      {
        int num = Integer.parseInt("PGDegree");
        System.out.println(num);
      }
      catch(NumberFormatException e)
      {
        System.out.println(" The argument given to parseInt should be a number");
      }
      try
      {
        int a[] = new int[5];
        a[7]=25;
      }
      catch(ArrayIndexOutofBoundsException e)
      {
        System.out.println("Oops!!! Don't think your array extends to that place,
        array index out of bounds exception");
      }
      System.out.println("Marks culmination of Nested Try block; print some other
      stmt");
    }
```

```
    catch(Exception e)
    {
      System.out.println("This will handle all exceptions and recover");
    }
  }
}
```

Output:



Now, letus see example of use of multiple ***catch*** clause in a program.

```
try{                    }
catch (Exception e1)
{// Catch Block}
catch (Exception e2)
{// Catch Block}
catch (Exception e3)
{// Catch Block}
```

To understand the utility of having multiple catch blocks, think of a situation where code may throw multiple exceptions and there is a need to handle each one of those separately and differently. Put simply, if you want to handle each exception differently then use multiple catches.

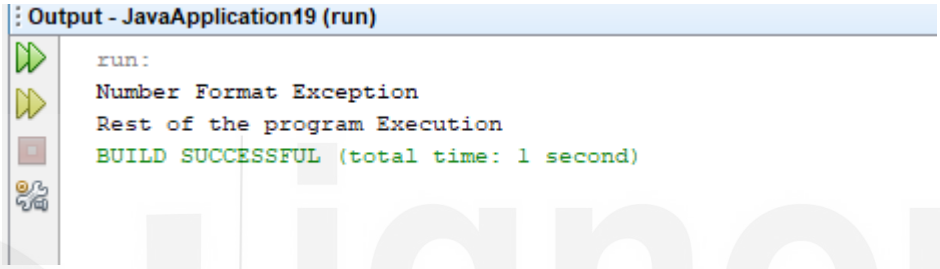
So here we have one try block and have different catch blocks. Now what we should practice while there is a need to handle such different types of exceptions while writing our code is to handle them from sub class exception to super class exception (specialized to generalized). Let us understand this with the help of an example. We have already seen the Exception hierarchy, from this we know that class FileNotFoundException extends IOException i.e. while writing multiple catch blocks we should first handle FileNotFoundException and then IOException. If we do the reverse i.e. handle IOException first, this will result in an unreachable code as FileNotFoundException will never be reached because IOException would have already handled that situation too. So you need to remember to place subclass exceptions higher in the list of catches.

Let us see an example of catching multiple exceptions:

```
Package ExceptionPGD;
public class PGDException
{
  public static void main(String args[])
  {
     try
     {
       int c  =  Integer.parseInt("Checking Virus");
       System.out.println("Resultant:", +c);
```

```
    }
    catch (NumberFormatException e)
    {
      System.out.println("Number Format Exception");
    }
    catch (Exception e)
    {
      System.out.println("This will handle all Exceptions");
    }
    System.out.println("Rest of the program Execution");
  }
}
```

Output:



Java has provided union catch feature since Java 7, where we may handle multiple exceptions using same catch block by clubbing them together as shown in example code below.

```
try {……}
catch (IOException | NumberFormatException e)
{
        System.out.println("This has failed to load", e);
}
```

Use of finally block in Exception Handling

The finally block in java is very important in exceptions handling as it is used to put important codes such as clean up code like closing the file, closing the database connection etc. The finally block is always executed whether exception generated or not and whether generated exceptions are handled or not. Therefore, in a finallyblock all the crucial statements are kept regardless if the exception occurs or not.

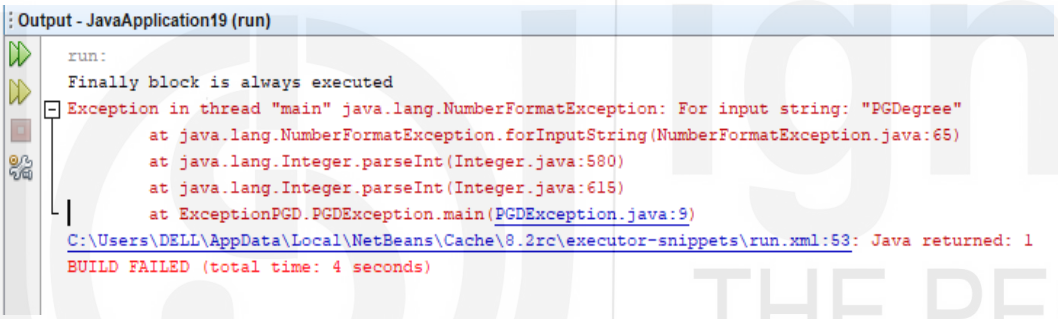
```
try{                    }
catch (Exception e1)
{// Catch Block}
catch (Exception e2)
{// Catch Block}
catch (Exception e3)
{// Catch Block}
finally { important code}
```

Let us see an example of use of finally block. In this program the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and print the given message then the program terminates abnormally.

```
package ExceptionPGD;
public class PGDException
{
  public static void main(String args[])
  {
    try
    {
      int num = Integer.parseInt("PGDegree");
      system.out.println(num);
    }
    finally
    {
      System.out.println(" Finally block  is always executed");
  }
}
```

In the above code since there is no catch block the exception thrown will not be caught and will show NumberFormatException, however the flow of program will not be broken and finally block will be executed.

Output:



7.5 THROW AND THROWS

1. throw is explicitly used to throw an exception but throws is used to declare an exception.
2. Checked exceptions cannot be propagated using only throw but it can be propagated using throws.
3. throw is followed by an instance and throws is followed by a class
4. throw is always used within a method and throws is used with the method signature.
5. throw clause can throw only one exception and not multiple exceptions and throws can declare multiple exceptions.

Syntax of throw :

```
void PGD()
{
  throw new ArithmeticException("Not working");
}
```

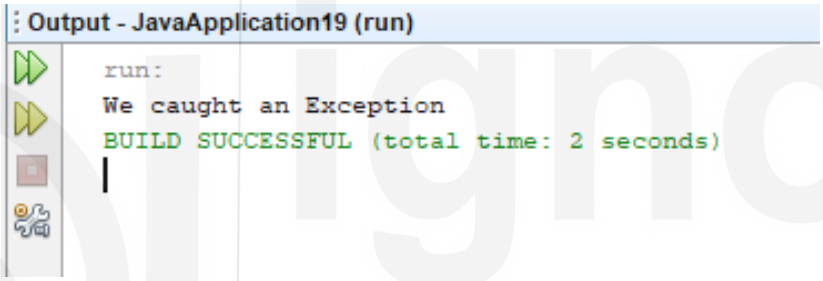
An example which demonstrate use of through clause in programming is given below.

```
package ExceptionPGD;
public class PGDException
```

```
{
    static void avrg()
    {
        try
        {
            throw new ArithmeticException("Demonstrating");
        }
        catch(ArithmeticException e)
        {
            System.out.println("We caught an Exception");
        }
    }
    public static void main(String args[])
    {
        avrg();
    }
}
```

When we execute the program, the following output will be received.

Output:



Throws

Throws is also a keyword which is used to declare the exceptions it does not throw any exception but it specifies that there may occur an exception in the method and it is always used with the method signature. Let us see how.

Syntax:

```
void a_method() throws ArithmeticException { }
```

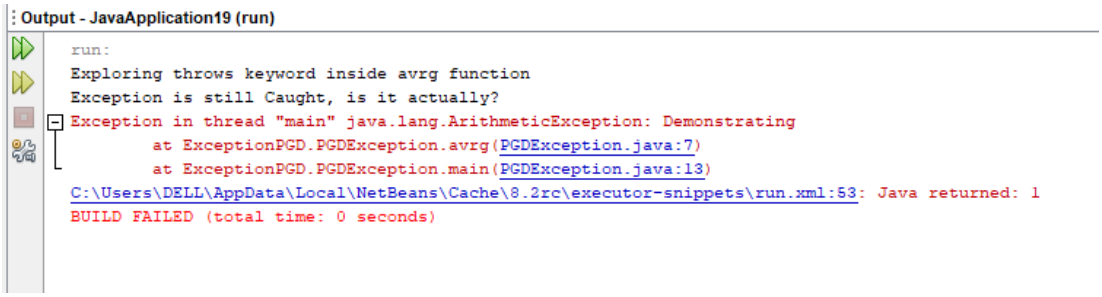
An example to demonstrate use of throws clause.

```
package ExceptionPGD;
public class PGDException
{
        static void avrg() throws ArithmeticException
        {
            System.out.println("Exploring throws keyword inside avrg function");
            throw new ArithmeticException("Demonstrating");
        }
public static void main(String args[])
{
    try
    {
        avrg();
    }
    finally
    {
```

```
        System.out.println("Exception is still Caught, is it actually?");
    }
}
}
```

It will notify that there is an exception in the main method (can you guess what type of Exception is it : Hint: Arithmetic) as we are not handling it via catch in our code but still print the following.

Output:



7.6 FINAL VS FINALLY VS FINALIZE

What willhappen in the situation when you have to execute some statement irrespective of whether it is caught as an exception or not? To handle such situation of finalize () method is used.

Let us understand the three - final, finally and finalize by looking at the differences between these three constructs; so final is a keyword, finally is a block, and finalize is a method. The final is used to apply restrictions on class methods and variables. The finally block is used to place an important code and finalize method is used to perform cleanup processing just before the object is garbage collected. Remember that a class declared as final cannot be inherited and the final method cannot be overridden and the final variable cannot be changed. As we have already learned it, the finally block will be executed whether the exception is handled or not.

7.7 USER DEFINED EXCEPTION

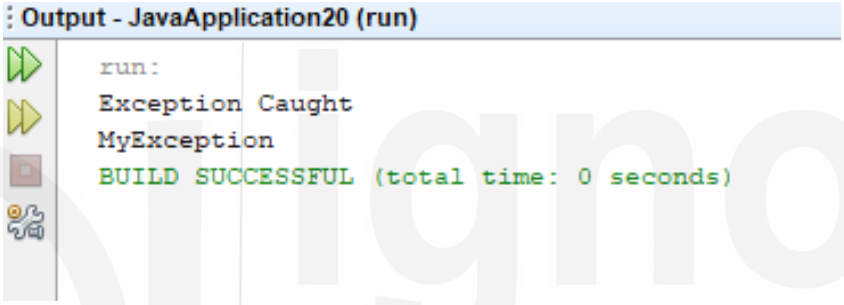
Sometimes the built-in exceptions in Java are not able to describe a certain situation. In such cases a user can also create exceptions and that are called user defined exceptions. There are two important points which are used while creating/defining user defined exception.

- A user defined exception must extend the exception class
- exception is thrown using a throw keyword

```
package javaapplication20;
class MyException extends Exception
{
   public MyException(String s)
   {
      // Call constructor of parent Exception
      super(s);
   }
}
public class PGDException
```

```
{
    public static void main(String args[])
  {
     try
     {
        // Throw an object of user defined exception
        throw new MyException("MyException");
     }
     catch (MyException e)
     {
        System.out.println("Exception Caught");
         // Print the message from MyException object
        System.out.println(e.getMessage());
     }
  }
}
```

Output:

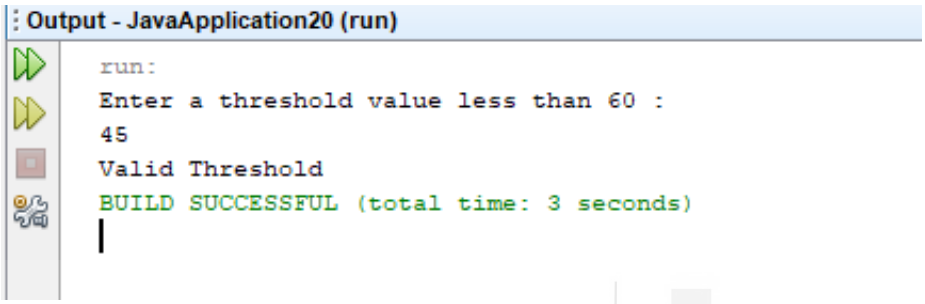


Another Example Program for Custom Exception

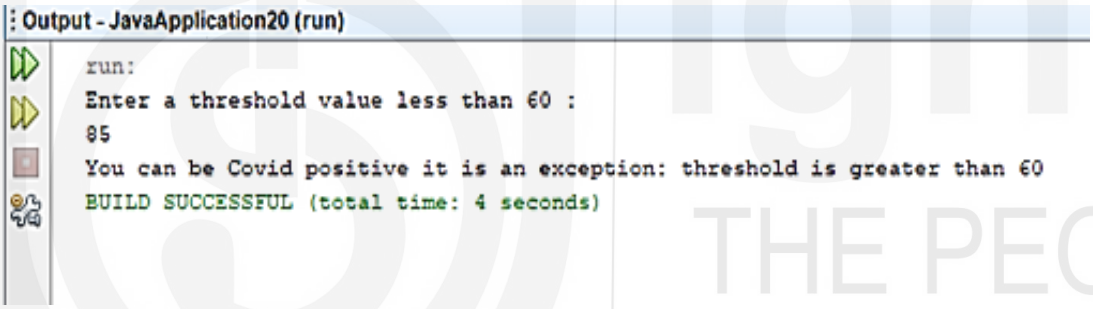
```
package javaapplication20;
import java.util.*;

class PGDException
{
static void ValidateThreshold(int threshold) throws InvalidInputException
{
if(threshold > 60)
{
throw new InvalidInputException("Invalid Threshold");
}
System.out.println("Valid Threshold");
}

public static void main( String args[])
{
Scanner scanner = new Scanner(System.in);
System.out.println("Enter a threshold value less than 60 :");
int threshold = scanner.nextInt();
try
{
ValidateThreshold(threshold);
}
catch(InvalidInputException e)
{
System.out.println("You can be Covid positive it is an exception: threshold is greater
than 60");
}
```

```
}
}
class InvalidInputException extends Exception
{
InvalidInputException(String exceptionText)
{
super(exceptionText);
}
}
```

Output-1 (for input value 45)



Output-2 (for input value 85)



Custom and User defined Exceptions come to our rescue in the rare situations where java already dose not has an exception class defined to handle an exception. These are handy tools which maintain the flow and readability of our code.

☞ Check Your Progress-3

1) Can there be a try block without a catch block ?

……

……

……

……

2) Which is the highest level of Exception Handling classes ?

……

……

……

……

3) Show the pre-defined exception hierarchy in Java.

……

……

……

4) Howdo you create a Custom Exception Classes?

……

……

……

5) What is the difference between java- ClassNotFoundException and NoClassDefFoundError.

……

……

……

6) What will happen when you compile and run the following code?

```
package javaapplication20;

public class PGDException extends Exception
{
String className;
public static void main(String[] args)
{
try
{
PGDException v = new PGDException();
if(v.className.equals("My Exception"
System.out.print("My Exception");
else
System.out.print("Other Exception");
}
catch(Exception e)
{
System.out.print("Exception Caught");
}
catch(NullPointerException ne)
{
System.out.print("Null");
}
}
}
```

……

……

……

……

7) Can an Exception be rethrown ? Is it required for the caller method to catch or re-throw the checked exception?

……

……

……

7.8 SUMMARY

In this unit we began with learning a unique feature of java language which is an statement *assertion* which is primarily used in non-production environment of code to prevent programmers making costly mistakes repeatedly and saw its advantages. Then we learnt about a relatively recent but important feature *annotations* which helps us document a program without writing lengthy comments. Finally, we culminated the unit by learning how *Exception Handling* is done in java andalso learnt how we create our own Exceptions.

7.9 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

☞ **Check Your Progress-1**

1) Assertions are **disabled** by default in java. We can use *java –ea or java -da* to enable or disable assertions respectively. If conditional statements used to check bug conditions reach production even after the bug has been removed they consume CPU cycles and may be cause of bottleneck or resource crunch. This problem does not exist with assertions as they are disabled by default.

2) Assert usually throws "Assertion Error" when the assumption made is wrong. Assertion Error extends from Error class (that ultimately extends from Throwable).

3) If assertions are enabled for the program in which the assertion fails, then it will throw Assertion Error.

4) An assert statement declares a Boolean condition that is expected to occur in a program. If this boolean condition evaluates to false, then an Assertion Error is given at runtime provided the assertion is enabled. If the assumption is correct, then the boolean condition will return true.

5) The Assertion Error thrown by the assert statement is an unchecked exception that extends the Error class. The assertions are not required to declare them explicitly and also there is no need to try or catch them.

6) To assert an exception we declare an object of ExpectedException as follows:

public ExpectedException exception = ExpectedException.none ();

Then we use it's expected () and expectMessage()methods in the test method, to assert the exception, and give the exception message.

☞ **Check Your Progress-2**

1) These are :

- *@Override* – marks that a method is meant to override an element declared in a superclass.
- *@Deprecated* – indicates that element is deprecated and should not be used.

- *@SuppressWarnings* – tells the compiler to suppress specific warnings.
- *@FunctionalInterface* – introduced in Java 8, indicates that the type declaration is a functional interface and whose implementation can be provided using a Lambda Expression

2) Annotations are there in the *java.lang* and *java.lang.annotation* packages.

3) Annotations are a form of an interface where the keyword *interface* is preceded by *@*, and the body contains *annotation type element* declarations that look just like methods:

```
public @interface SelfDefinedAnnotation
{
   String value();
   int[] types();
}
```

To start using it using your code :

```
@SelfDefinedAnnotation(value = "A String type element", types = 1)
public class MyAnnotationUser
{
   @SelfDefinedAnnotation(value = "any new relevant value", types = { 31, 48 })
   public Element nextElement;
```

4) On behalf of the annotation, java compiler or JVM performs some additional operations. By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

5) Object

6) java.text

7) @Retention annotation specifies how the marked annotation is stored.

@Documented annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.)

@Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.

@Inherited annotation indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class superclass is queried for the annotation type. This annotation applies only to class declarations.

@Repeatable annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use.

☞ Check Your Progress-3

1) Yes. (java 7 onwards)

2) Throwable is the highest level of Error Handling classes.

3)
```
//Pre-defined Java Classes
class Error extends Throwable{}
class Exception extends Throwable{}
class RuntimeException extends Exception{}
```

4) class MyOwnRTPCRTestException extends Exception{}

5) Both *ClassNotFoundException* and *NoClassDefFoundError* occur when the JVM can not find a requested class on the classpath.*ClassNotFoundException* is a checked exception which occurs when an application tries to load a class through its fully-qualified name and can not find its definition on the classpath
NoClassDefFoundError is a fatal error. The error occurs when a compiler could successfully compile the class, but Java runtime could not locate the class file.

6) The catch block catching child class must come first in order before the catch block catching the parent class.
NullPointerException is child class of the Exception class (Exception > RuntimeException > NullPointerException), so the catch block with NullPointerException must come before the catch block with Exception class.
The code will give compilation error "Unreachable catch block for Null Pointer Exception. It is already handled by the catch block for Exception".

7) Yes, Exception can be rethrown. Yes, if the method is throwing any of the checked exceptions, the caller must either catch it using try catch block or re-throw it using throws clause. Otherwise, compilation fails.

7.10 REFFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Savitch, Walter, " Java: An introduction to problem solving & programming", Pearson Education Limited, 2019.
- Muneer Ahmad Dar, "Java Programming Simplified: From Novice to Professional", BPB, 2020.
- Lulliana Cosmina, "Java for Absolute Beginners", Apress,2018.
- Yashvant Kanetkar, "Let Us Java", BPB,2019.
- https://www.buggybread.com
- https://www.baeldung.com
- https://www.javacodeexamples.com
- https://docs.oracle.com/en/java/

ignou THE PEOPLE'S UNIVERSITY

UNIT 8 PACKAGES AND INTERFACES

Structure **Page No**

8.0 Introduction 186
8.1 Objectives 186
8.2 Package 187
8.3 Access Rules in Packages 188
8.4 Finding Packages and CLASSPATH 189
8.5 Creating Own Packages 190
8.6 Importing Packages 195
8.7 Basics of Interface in Java 198
8.8 Defining, Implementing and Applying Interface 199
8.8.1 Defining an Interface 199
8.8.2 Implementing Interfaces 201
8.8.3 Applying Interfaces 203
8.9 Extending Interfaces 204
8.10 Default Interface Methods 206
8.11 Issues of Multiple Interfaces 208
8.12 Use of Static Methods in an Interface 210
8.13 Summary 212
8.14 Solutions/ Answer to Check Your Progress 213
8.15 References/Further Reading 215

8.0 INTRODUCTION

Java programming language provides a powerful feature called packages which are groups of related classes and interfaces together in a single unit. Packages are used to manage large groups of classes and interfaces to avoid naming conflicts. The Java API itself is implemented as a group of packages. Java supports two types of packages,i.e. built-in packages and user-defined packages. In this unit, you will learn more about the user-defined packages as well as interfaces.

Similar to the Class, Interface is also one of the important features of java. In this unit, you will learn how to define methods in an interface and how to use those methods in a class. Each Interface is compiled into a separate bytecode file just like a regular class, but you cannot create an instance of the interface. This unit provides detailed information about the interfaces from declaration to its implementation. Using this unit, you can learn how to use static as well as default methods in the interfaces. Apart from this information, you will know about the feature of the interface through which you can use multiple inheritances in a java program. In this unit, examples are given at each step which will help you to understand the concepts and write java program.

8.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the need and the purpose of packages,

- create own package and import packages in program,
- explain the concept of interfaces,
- implement interfaces, and
- work with default and static methods in an interface.

8.2 PACKAGE

In java, a package is a collection of similar types of classes, interfaces and sub-packages. It is just like a folder in a file directory. A package statement identifies the directory path for a given class. For example, when you write an application program in java programming language, there may be many individual classes. You can place related classes into a folder and name this folder for organizing these classes; now, it is called a package. In other words, you can say that the package statement defines a namespace in which classes are stored. It is a good practice to group the related classes in a package. Packages are used to avoid name conflicts and to write better maintainable code.

There are two types of packages in java,i.e. built-in packages and user-defined packages.

Built-in packages

As you are already aware, the Java API is a library of prewritten classes to help in programming. Built-in packages are those packages that Java API provides to simplify the task of java programmer. Some Java API are discussed in detail in Block 3 Unit 4 of this course.

Java provides several built-in packages or predefined packages. The complete list of these built-in packages can be found at the link:https://docs.oracle.com/ javase/8/ docs/api/.

Here are some of the commonly used built-in packages:

- java.lang – it contains fundamental classes to the design of the Java programming language.
- java.io − classes for input and output functions are grouped in this package
- java.util – it contains utility classes

User-Defined Packages

A package that the user creates is known as user-defined package. Besides using built-in packages, whenever the user (programmer) wants to categorize the classes and interfaces and avoid naming conflicts, the user creates a user-defined package for better java programming.

You can create your own packages. For example, like the following :

```
└── javaclasses
    └── IGpack
        └── socis.java
```

You need to use the package keyword to create a User-defined package. The subsequent section of this unit gives more about the creation of own package and how to import built-in packages and user-defined packages in a java program.

8.3 ACCESS RULES IN PACKAGES

Before you get into the depth of the topic Packages in Java, you may need to know about accessing rules in packages. You have already come across java access control mechanism and their access modifier keywords such as public, private and protected while writing your java program in the previous units of this course.You have known that package is a collection of related classes. It can also have interfaces and sub-packages. When you put your classes in a package, what will be the accessibility or scope of class within the package? The discussion on this point is given in this section.

Packages in Java add another dimension to access control. Java provides many security levels that offer the visibility of members within the classes, subclasses, and packages. Classes and packages both define the feature of data encapsulation. Class acts as containers for data and methods, whereas Packages works as containers for classes and other sub packages. The package is naming as well as a visibility control mechanism. Java supports four categories regarding the visibility of the class members between classes and packages:

- Sub classes in the same package
- Sub classes in different packages
- Non-subclasses in the same package
- Classes that are neither in the same package nor subclasses

You can limit the access of classes, methods and variables throughout packages by using access specifiers. The three main access modifiers private, public, and protected provide a range of access required by these categories. A class can have only two access modifier, one is default and another is public. If the class has default access then it can only be accessed within the same package.When a class has public access, it can be accessed from anywhere by any other code outside of the package in which that class is. The following table applies only to members of classes. In the table, 'Yes' means that the data is accessible and 'No' means data cannot be accessible.

Table 8.1: Java access control mechanism for class member

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

You can understand the java access control mechanism in a very simple manner. You can declare anything as **public** then it can be accessed from anywhere. Anything declared as **private** can be seen only within that class. When you do not mention any access modifier, it is called **default** access modifier, and it is visible to sub-classes as well as to other classes in the same package. It means that the scope of this modifier is limited to the package only. If you declare anything as **protected** then code is accessible in the same package and its subclasses present in any other package. This way, java packages provide access control to the classes.

8.4 FINDING PACKAGES AND CLASSPATH

When you develop an application, there may be a number of classes, and for organizing them, you need to create a package. Packages solve many problems related to access control and namespace conflicts perspective. Also, it is important to note that, they become the origin of some difficulties when you compile and run java program. This is because the specific location that the java compiler will consider as the root of any package hierarchy is controlled by CLASSPATH. The CLASSPATH is an environment variable that Application ClassLoader uses to locate and load the .class files. The classpath is the path that the Java runtime environment searches for classes and other resource files.

For example, let us say if you create a class socis under a package named IGpack. The directory name must match your package name exactly. So, you create a directory similar to package name as IGpack and put your java file (for example socis.java) inside that directory. Now, your current working directory is IGpack and compiles your socis.java file. After compiling the socis.java file, the class file is stored in IGpack directory as shown in the figure 8.1.

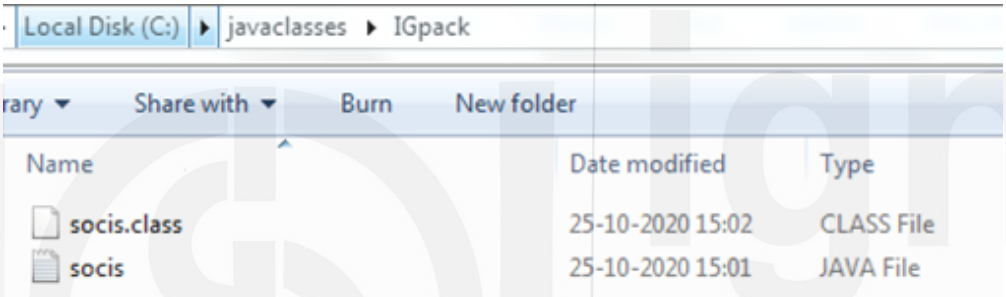


Figure 8.1: Directory structure for socis.java file

When you execute socis.class file through the java interpreter, it generates an error message: "Could not find or load main class socis" ,as shown in the figure 8.2.

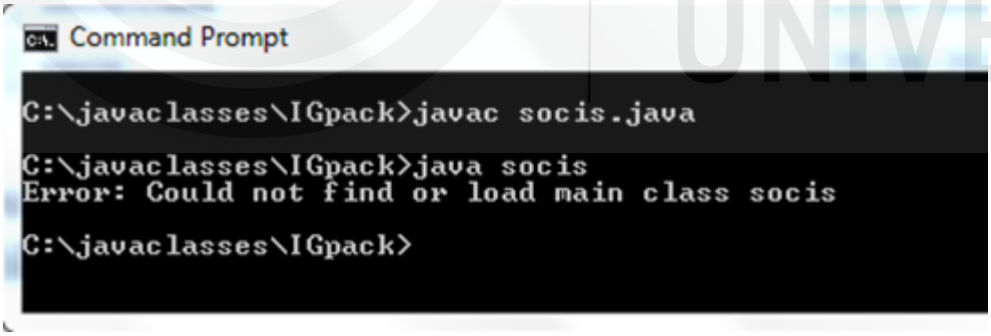


Figure 8.2: Command Prompt for execution of socis file

This is because the class is now stored in a package called IGpack. This class is not a simple class; it is created under the package. You can refer this class with package name like IGpack.socis. If you run with this reference, you will get an error message again. The reason behind these error messages is hidden in your CLASSPATH variable. In this example, IGpack is the name of a directory which is created by you. Actually, there does not exist any IGpack directory in the current working directory. For this reason, you need to set your development class hierarchy to classpath environment variable. CLASSPATH sets the top of the class hierarchy. Then you will be able to run your java file from any directory using the **java IGpack.socis** command. For example, if you are working on your source code in a **c:/javaclasses** directory then set your CLASSPATH to c:\javaclasses;

For classpath variable setting, you can refer environment setup for java development in Unit1 Block 1 of this course. Now, you can run the above example using **java IGpack.socis** command, program will run successfully and show the result as given in figure 8.3.

Figure 8.3: Command Prompt screen for successful running of socis file

8.5 CREATING OWN PACKAGES

In the previous section, you have learned about the packages which comprise the built-in packages and user-defined packages. Built-in packages are part of the java development kit, and users develop User-defined packages to group the related classes, interfaces, and sub-packages.

In this section, you will learn how to create your own package and sub package within the existing packages, along with examples. Java provides package commands to you for creating your own package. It is written as the first statement in a java source file. There can be only one package statement in each source file. You can define more than one file in the same package statement. The general form of the package statement is given below:

package pkg_name;

Here, package is java keyword, and pkg_name is the name of the package. For example, the following statement creating a package called MyFirstPackage.

package MyFirstPackage;

You must remember that each package in Java has its unique name. When you define the package name, it must match with the directory name exactly. *You cannot rename a package without renaming the directory in which the classes are stored.*

You can also create a hierarchy of packages. For this, you can simply separate each package name from the other package name by using a period. The general form of multi-levelled package statement is shown below:

package pkg_name1[.pkg_name2[.pkg_name3]];

The following examples explain how to create your own packages:

Creating a package in java is very easy; simply include a package command followed by the name of the package as the first statement in java source file like the following:

```
package mypack;
public class student
{
```

```
    String studentId;
    String studentName;
}
```

Creation, compilation and execution of Java Package

In the next example, you will learn about **creation, compilation and execution of java package.**

Example-2: Create a class named 'socis' under the package 'IGpack1'. You can follow some simple steps for creation, compilation and execution of java package.

```
package IGpack1;  // creation of package
class socis
{
    public void method1()
    {
        System.out.println("method1 of socis");
    }
    public static void main(String args[])
    {
        socisobj = new socis();
        obj.method1();
    }
}
```

- For testing purpose, you can create a folder 'javaclasses' in C drive. You can write the above source code in a notepad and save as 'testpack.java' in "javaclasses" folder.
- If you are not using any IDE, you need to follow the syntax given below and compile 'testpack.java' file using the javac command:

Figure 8.4: Command Prompt for compiling testpack Java program

- After successfully compilation of 'testpack.java', a class file 'socis' is created under your "javaclasses" folder.

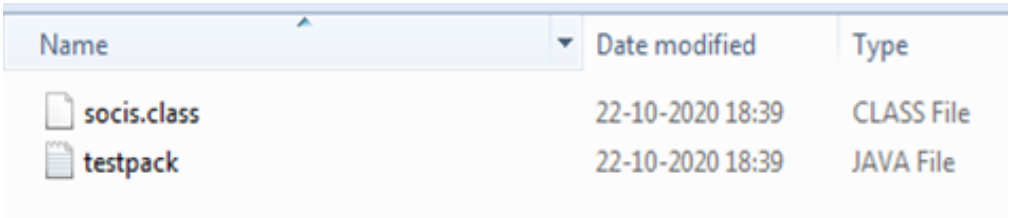


Figure 8.5: File Structure for testpack JAVA file

Compilation of Java Package program

- Now you can compile java program with package statement using the following command at the Command Prompt. This command dictate the compiler to create a package. The -d switch in the command specifies the destination where to put the generated class file. If you want to keep the package within

the same directory, you can use "." (dot). The "." operator represents the current working directory.

javac –d . testpack.java

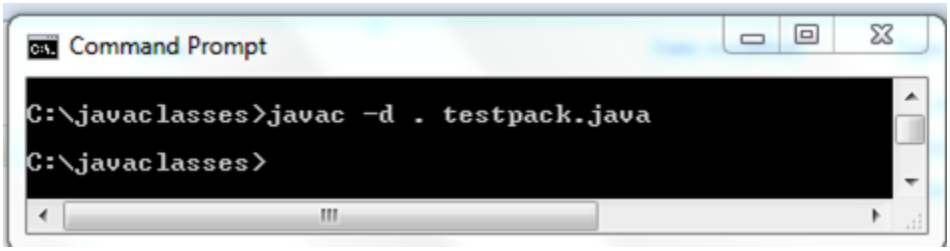


Figure 8.6: Compiling Java Program

- After successfully compilation, a folder with the given package name is created in the specified destination and the compiled class file will be placed in that folder. You can see in following figure 8.7 "IGpack1" named package has been created.

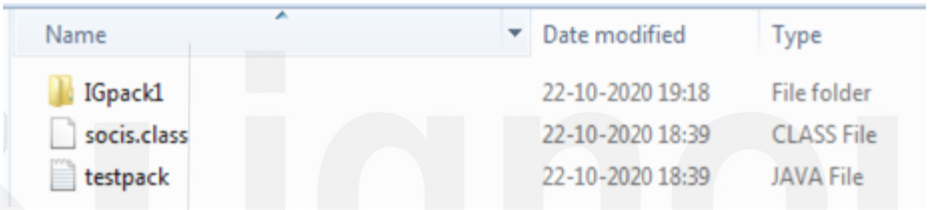


Figure 8.7: File Structure after package creation

- When you open the Java Package 'IGpack1' inside you will find "socis.class" file as shown in figure 8.8.

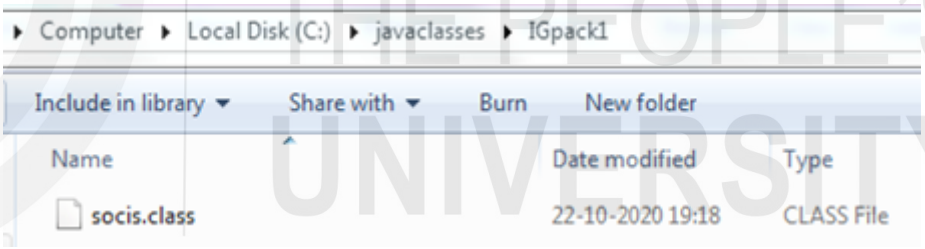


Figure 8.8: File Structure for socis.class file under IGpack1 package

Executing Java Package program

- When you execute socis.class file using the following command, it will display result like in figure 8.9.

java IGpack1.socis

Figure 8.9: Output Screen for Executing Package in Java

In the above example, the current directory is "javaclasses". If you want to create a package in the parent directory, you can take your same file (i.e. testpack.java) and

compile using the following command. See the figure 8.10 for details. In this case, the file will be saved in C Drive. The double dot(..) represents the parent directory.

javac –d .. testpack.java

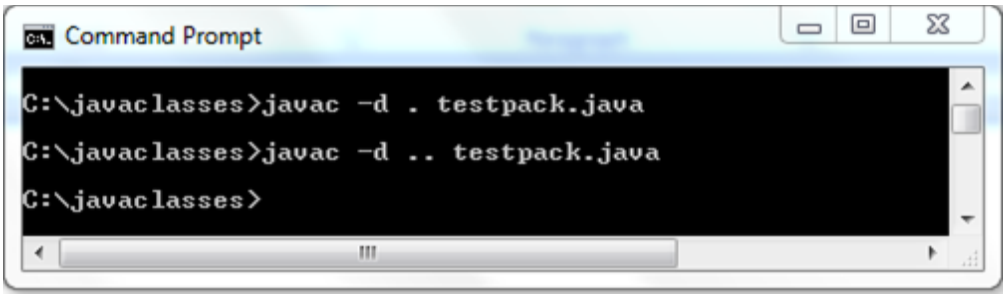


Figure 8.10: Command Prompt for compiling Java program to create package in parent directory

- You can see the following figure 8.11, package IGpack1 is created in C drive. When you click on this folder, it will show "socis.class" file.

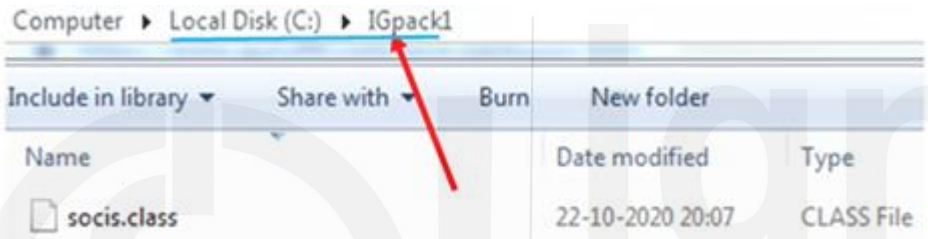


Figure 8.11: File Structure of creating socis.class file in parent directory

Creation of a sub packagewithin your existing java package

You have seenthe creation of own packages, compilation, and execution of Java Package program through the above example. If you want to create a sub package within your existing java package, you can take the above source code (i.e.testpack.java) with a slight modification in the package statement only as like the following:

package IGpack1.IGpack2;

Here, IGpack1 is parent package and IGpack2 is sub-package. Now compile the modified testpack.java file using the following command:

javac –d . testpack.java

After successfully compilation the file, sub package IGpack2 is created (shown in figure 8.12) under the existing package 'IGpack1' and it contains socis.class file as shown in figure 8.13.

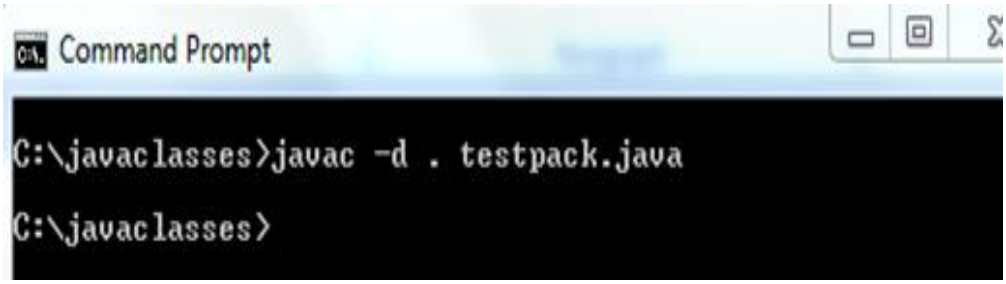


Figure 8.12: Compiling testpack.java file for creation of sub-package

Execution of sub package

For executing the source code of sub-package, you can mention the fully qualified name of the class i.e. main package name followed by the sub-package name followed by the class name as the following:

java IGpack1.IGpack2.socis

This is how the package is executed and gives the output as "method1 of socis" from the code file as shown in figure 8.14

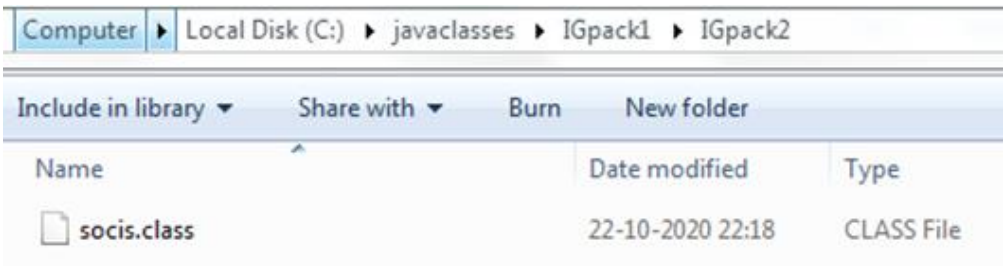


Figure 8.13: File structure for sub-package IGpack2

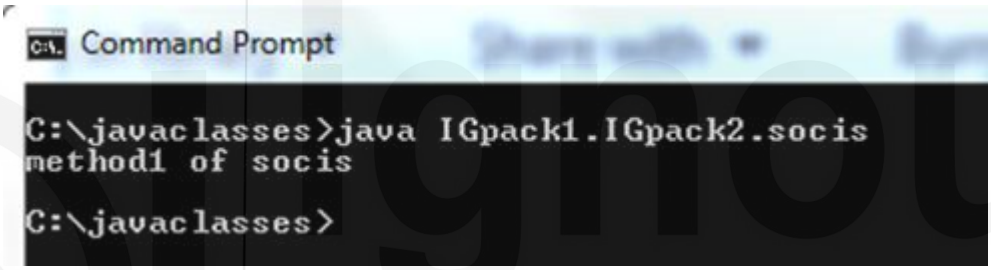


Figure 8.14: Command Prompt for executing of sub-package program

The same idea as discussed is used for creating packages using some IDE. To create your own Java package using NetBeans IDE, right click on the source packages and select New -> Java Package, as shown in figure 8.15.

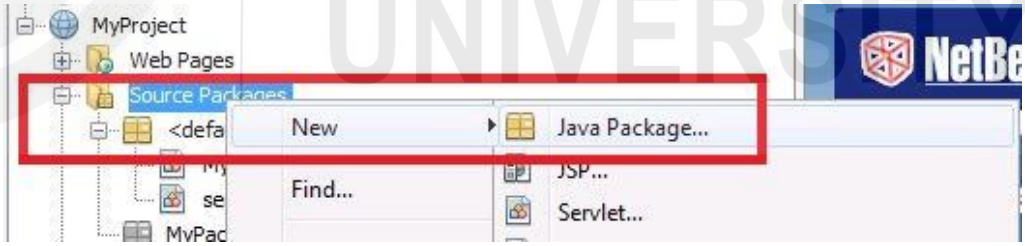


Figure 8.15: Creation of Package in NetBeans

After selection of the above procedure, you can enter name and location of package. you can see in figure 8.16.

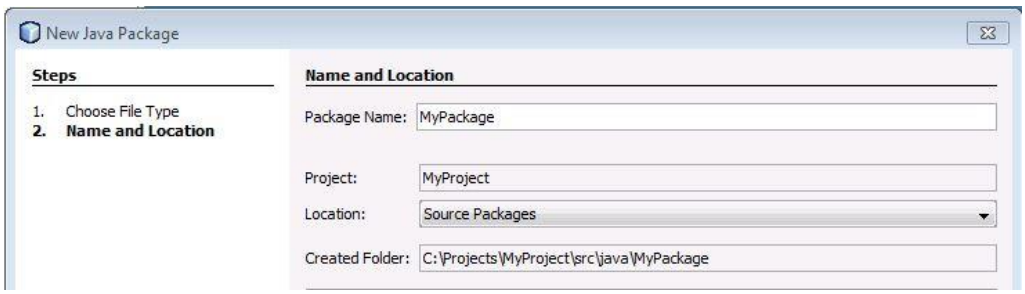


Figure 8.16: Name and Location of Package

Now, you can select the Finish button from the Button Panel.

Figure 8.17: Button Panel

This way you can create your own package. For creating java class under the package, right-click on the package, select New->Java Class.A package declaration is automatically inserted into each new source file it creates.

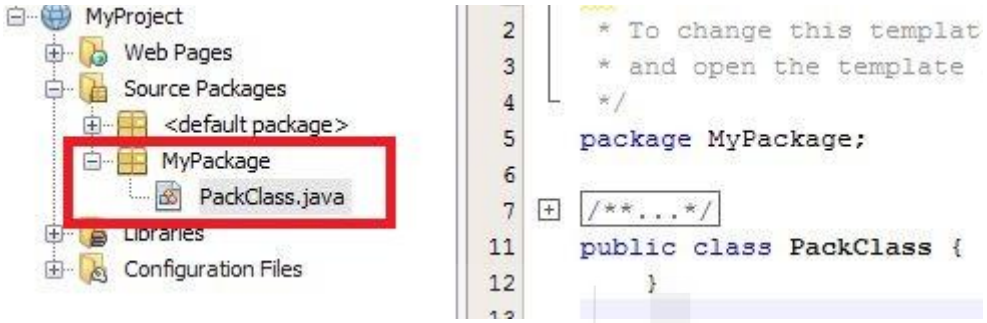


Figure 8.18: Creation of Java class under Package

To create a sub package in NetBeans, right-click on the parent package, select New-> Java Package. Also give a name to new sub package, you can see example in figure 8.19:

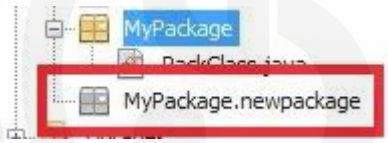


Figure 8.19: Creation of Sub-package in NetBeans

To create a Java sub class under the sub-package, right click on sub-package and select New-> Java Class and then give a name to new sub class. you can see in figure 8.20 for creation of 'subclass.java' file under the 'newpackage'.

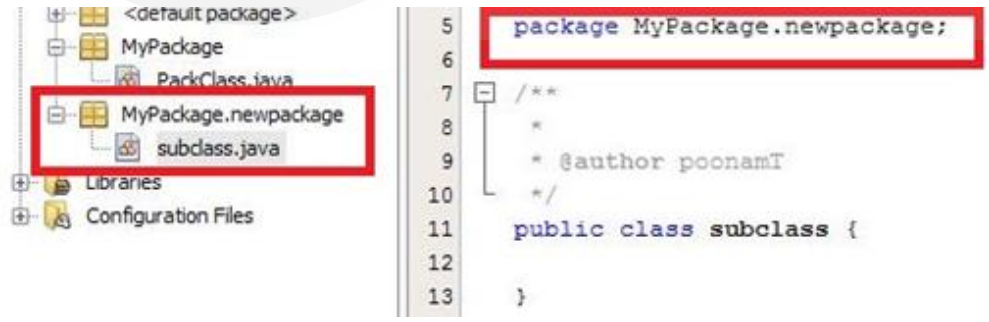


Figure 8.20: Creation of sub-class under the sub-package

This section clarified to you the creation of your own package and sub package in java. You have also learnt compilation as well as execution of package and sub-package program.

8.6 IMPORTING PACKAGES

This section describes how to import built-in packages as well as user-defined packages in your java source file.

Java provides import statement for including packages in java source file. Remember that in java programming, the import statement is written directly after the package statement (if it exists) and before the class definition. You can write more than one import statements. The general form of the import statement is as follows:

import package1[.package2].(classname|*);

You can import any specific package member or import all the types contained in a particular package.

For example,

```
import java.util.*;   // Import the whole package
import java.util.Calendar;  // Import a single class;
```

In the above example, java.util is a standard java package while Calendar is an abstract class of the java.util package. It provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc.

The above example shows you how to import built-in packages in java program. In a similar manner, you can use import statement to include your created packages in java source code. Let's discuss user-defined packages with the help ofan example.

Example: Create a class called Addition inside a package named MyPackage and save it as "Addition.java" in "javaclasses" folder which is already created in the previous section.

```
package MyPackage;
public class Addition
{
   public int add(int a, int b)
    {
        return a+b;
    }
}
```

Now let us see how to use this package in another program. Write the following code and save it as TestPackage.java

```
import MyPackage.Addition;
public class TestPackage
{
     public static void main(String args[])
     {
          Addition obj = new Addition();
          System.out.println(obj.add(100, 200));
     }
}
```

Now, compile both the files and run TestPackage file, using the following steps. Also the same process are shown in figure 8.15.

Step 1: Compile Addition.java file using **javac Addition.java** command at thecommand prompt and it creates Addition.class

Step 2: Now, you can use javac –d . Addition.java and it createsAddition.class file inside MyPackage

Step 3: Compile TestPackage.java file and it creates TestPackage.class

Step 4: Run TestPackage with java interpreter and it shows the addition of two numbers

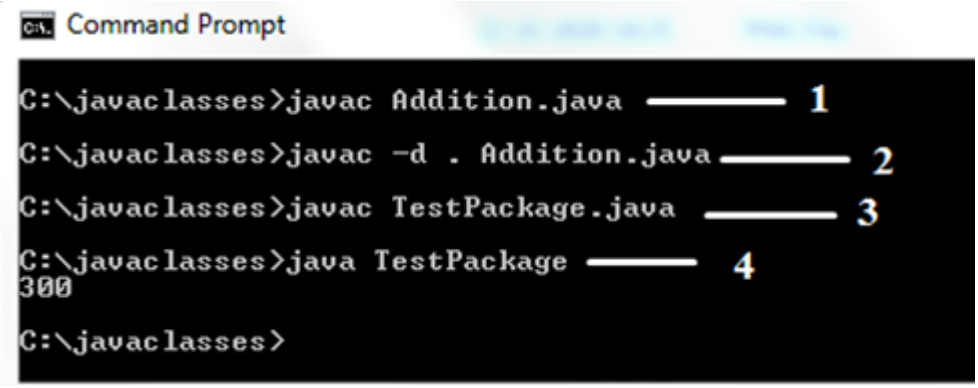


Figure 8.21: Command Prompt for compiling and executing Java Program

The file structure of the above example is shown in the following figure 8.22:

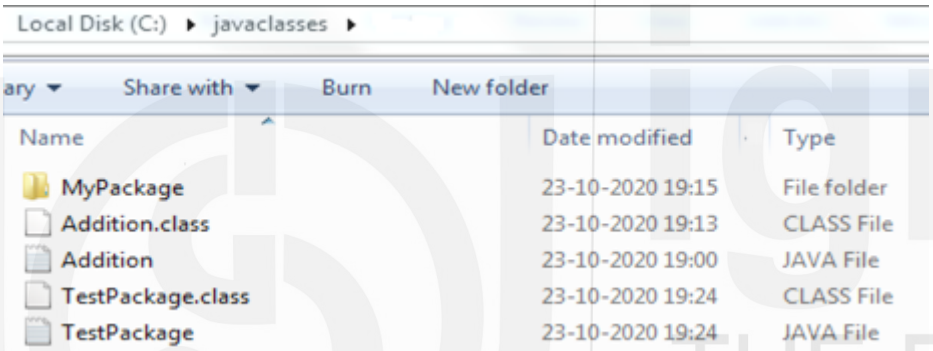


Figure 8.22: File structure of the above importing example

To use the class Addition, you have imported the package MyPackage. In the above program you have imported the package as MyPackage.Addition in TestPackage.java; this only imports the Addition class. However, if you have several classes inside package MyPackage , you can import the package as shown below.

import MyPackage.*;

☞Check Your Progress-1

1) What are Packages? What are the advantages of packages? Write the name of the default package in JDK.

……

……

……

……

2) What are different types of packages in Java?

……

……

……

……

3) How would you compile a source code of Java that is created as a package? Also, write command for executing Java Package program.

……

……

……

4) How are packages imported in java program?

……

……

……

8.7 BASICS OF INTERFACE IN JAVA

Both Class and Interface are important concepts that build the foundation stone for java programming.

You have gone through the concept of classes in Unit 4 Block 1 of this course. In this section, you will learn about the basic features of interfaces in java programming languages. In java, an interface is syntactically similar to classes, but they lack instance variables, and their methods are just declared without having any body. It means that the Interface can contain only constants declaration, method declaration, default methods, static methods.

The following table shows some differences between classes and Interface:

Table 8.2: Difference between Class and Interface

	Class	**Interface**
Method	Abstract and concrete methods are defined.	Only abstract methods are defined
Member specifier	Members of a class can be defined as public, private, protected or default.	By default, all the members are public
attributes and behaviours	A class describes the attributes and behaviours of an object.	An interface describes behaviours that a class implements.

The interface contains one or more method declarations without their implementations. All the methods of an interface are abstract methods. The variables in an interface are public, static and final. Once it is defined, any number of classes can implement an interface. The Interface keyword is used to create an interface. An interface is a powerful mechanism for defining behaviour among unrelated classes.

As you already know, a class with the abstract keyword in its declaration is called abstract class. An abstract class can have both abstract methods,i.e. methods without a body, and regular methods with implementations. You have already studied the concept of abstract classes in Unit 2 Block 2 of this course. You can lay down some comparison between an interface and abstract class as under:

- Interface can be implemented using ‘implements’ keyword, whereas an abstract class can be inherited using the ‘extends’ keyword.

- Interface can only have public members, while abstract class can have any type of members like public, private etc.
- Variables declared in a Java interface is by default final, whereas an abstract class may contain non-final variables.
- Interface is completely abstract and cannot be instantiated. A Java abstract class also cannot be instantiated but can be invoked if a main() exists.
- An interface can extend another Java interface only, while an abstract class can extend another Java class and implement multiple Java interfaces.

The next sections give you more details on defining, implementing and applying interface. You will also know about Default Interface Methods, Issues of Multiple Interfaces, and the use of Static Methods in an Interface. In addition to this, you can extend an interface using *extends clause*. So, you can go thoroughly run your program correctly.

8.8 DEFINING, IMPLEMENTING AND APPLYING INTERFACE

In the previous section, you have studied the basics of Interfaces. Now, this section will present you with how to define constants and methods in an interface as well as how a class can implement this interface. The syntax of creating an interface is very close to that for creating a class with few exceptions. The most significant difference is that none of the methods in interface may have a body. The interface can be seen as a prototype for a class.

8.8.1 Defining an Interface

Interface keyword is used to declare an interface. The interface has the following general syntax:

<accessibility modifier>**interface**<interface-name><**extends** interface-clause>

```
{
  // interface declaration
  <constant declarations>
  <method prototype declarations>
  <nested interface declarations>
}
```

Here, access modifier is either **public** or not used. When you do not mention any

access modifier then it is treated as default access modifier, and it is visible to other members of the package in which it is declared. When it is declared as public, the interface can be used by anyone. The interface-name is the name of interface, and it can be a valid identifier. The interface declaration part can contain member declarations which cover constant declarations, method prototype, nested interface declarations.

Here is a simple example of declaring an interface:

```
public interface SimpleInterface
{
   public String xyz = "IGNOU";
   public void method1();
}
```

As you can see, an interface is declared by using the Java **interface** keyword. The above interface example contains one variable and one method. The variable can be accessed directly from the interface, like this:

System.out.println(SimpleInterface.xyz);

Constants in Interfaces

You can define constant in an interface. The constants declaration is considered to be public, static and final. An interface constant can be accessed by any user/client (i.e. interface or a class) using its fully qualified name irrespective of whether the user/client implements or extends its interface. However, if a client is an interface that extends this interface or a class that implements this interface then the client can access constants directly without using the fully qualified name.

Interface Methods

In javaan interface contains one or more method declarations. All methods in an interface are public, even if you leave out the public keyword in the method declaration. As mentioned earlier, the interface cannot specify any implementation for these methods. It is the job of the classes implementing the interface to specify an implementation for the methods of the interface.

For defining Interface in Java using Netbeans IDE, start NetBeans software and create your project as JavaApplication and enter name of the project and select Finish button from the button panel. Now, expand Source Packages and right-click on the your own defined Packageand select New->Java Interface.

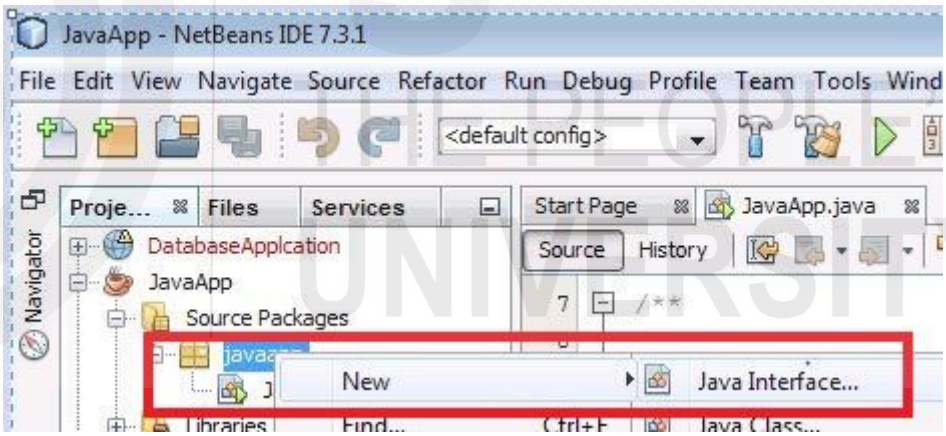


Figure 8.23: Creation of Interface in NetBeans

Now, Enter name and location of the interface.

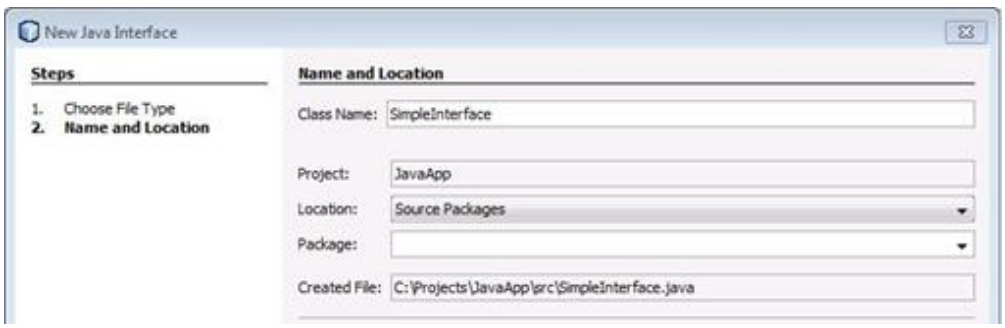


Figure 8.24: Name and Location of Interface

This way you can create an Interface in Java using NetBeans.

8.8.2 Implementing Interfaces

As yet you know that the methods in the interface are declared with an empty body. Once it is defined, a class can implement an interface for accessing these methods. It is compulsory for a class that implements an interface to all the methods declared in the interface. Any number of interfaces can be implemented by a class using the**implements** keyword. The syntax for implementing interfaces is given below.

<*access specifier*> class [extends *superclass-name*]
[implements *interface-name*[,*interface-name*…]]
{
// class body
}

For example the following program shows you how a class implements an interface. The two methods are defined in interface -ImpleInterface, and the class has to implement all the methods declared in the ImpleInterface.

```
interface ImpleInterface
{
  public void method1();
  public void method2();
}
class Impleclass implements ImpleInterface
{
    public void method1()
    {
        System.out.println("implementation of method1");
    }
   public void method2()
    {
        System.out.println("implementation of method2");
    }
    public static void main(String arg[])
    {
        ImpleInterface imp = new Impleclass();
        imp.method1();
        imp.method2();
    }
}
```

Now, you can compile and run the program. After successfully running, it will give the output as like the following figure 8.25:

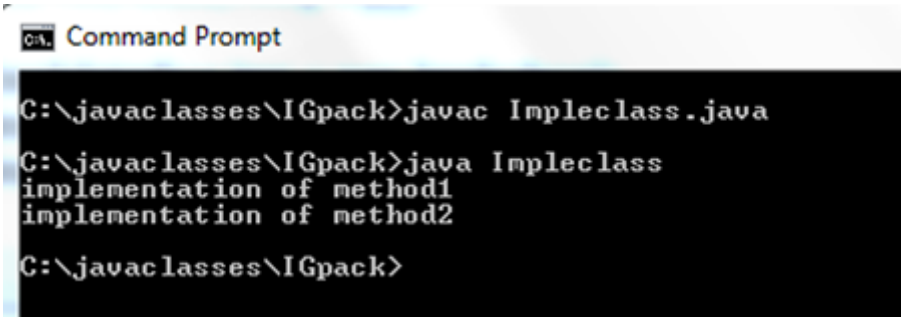


Figure 8.25: Compiling and executing of Implementing Interface Example

The file structure of the above program is shown in figure 8.26:

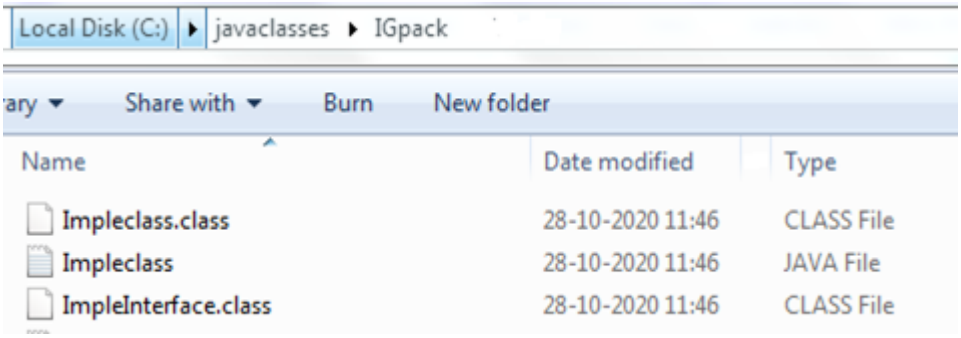


Figure 8.26: File structure for Implementing Interface Example

Now you can implement interface using the above example in NetBeans. For this you can write the code in the created

Interface such as 'SimpleInterface.java' like the following Figure 8.27:

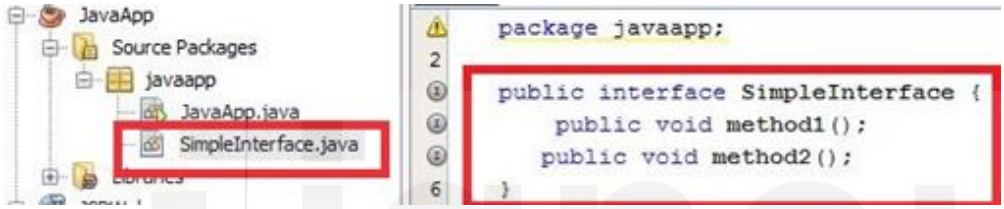


Figure 8.27: Example of Interface in NetBeans

Now implement the Interface named 'SimpleInterface' in Java Class like the following Figure 8.28:

Figure 8.28: Implementaion of Interface using NetBeans

Now, you can compile the java class as well as Interface using F9 and right-click on the Java Class program and select 'Run File' or Shift+F6 to run the program and after successfully running, it will give the output as like the following figure 8.29

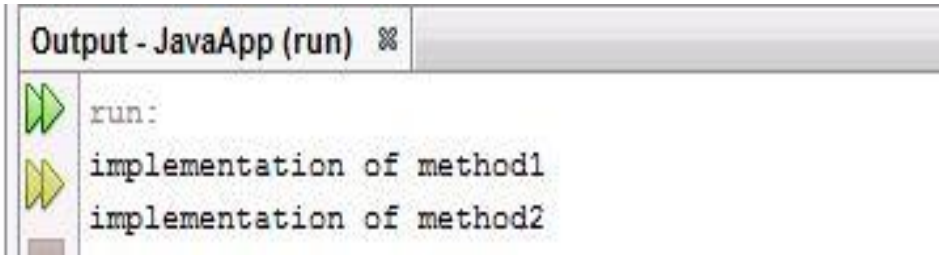


Figure 8.29: Output of the above Java Program

Accessing Implementations through Interface References

You have seen in the above example, there are two methods in the interface which are implemented by the class. In the Java program, you would have noticed that variable **imp** is declared to be of the interface type **ImpleInterface**, it is assigned an instance of **Impleclass**. The variable imp can be used to access method1() and method2(). You can access the methods declared by its interface declaration using interface reference variable. You cannot use interface reference variable to access non-interface method.

8.8.3 Applying Interfaces

In the preceding sections, you have been explained about the declaration and implementation of interface. This section will explain you on how to use both of them with a real example.

Following example will help you in understanding the concept of declaring and implementing interface by the class. In this example, an interface named Area is defined which contains constant and method which is implemented by two classes such as Rectangle & Circle and they will compute area as per their shape.

// **Example for Implementing Interfaces**
//**InterfaceTest.java**

```
interface Area                          //interface defined
{
        final static float pi=3.14f;          // constant declare
        float compute(float x, float y);   // method declaration
}

class Rectangle implements Area         //interface Area implemented
{
public float compute(float x, float y)
        { return (x * y); }
}
 class Circle implements Area                // interface Area implementation
{
public float compute(float x, float y)
        { return (pi * x * y); }
}

class InterfaceTest
{
      public static void main(String args[])
      {
         Rectangle rect = new Rectangle();
         Circle cir  = new Circle();
         Area area;                                  // interface object
         area=rect;
         System.out.println("Area of Rectangle =" + area.compute(10,20));
         area=cir;
         System.out.println("Area of Circle =" + area.compute(10,20));
      }
}
```

After successfully compilation, it will show output as shown in following figure 8.30:

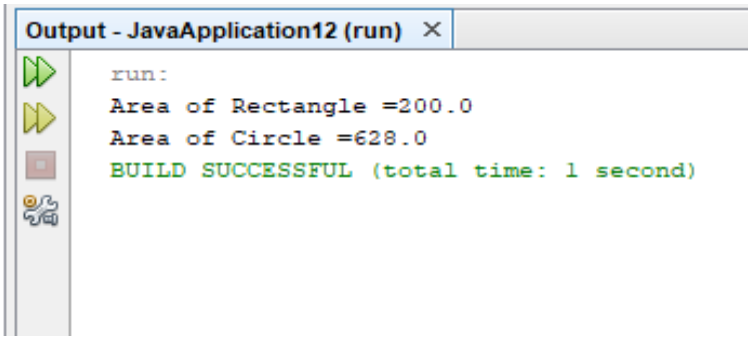


Figure 8.30: Command Prompt Showing Output of above Example

The file structure of the above example is shown in following figure 8.31:.

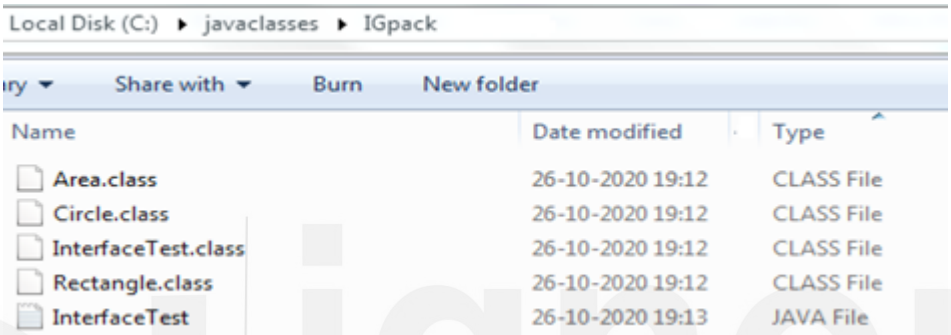


Figure 8.31: File Structure of the above Example

8.9 EXTENDING INTERFACE

An interface can extend other interface by using the keyword **extends**. The syntax is similar to inheriting classes. When a class implements an interface that inherits other interface then the class must implement all methods defined within the interface inheritance chain.

For Example: The following program demonstrates the use of extending interface:

```
interface X1 {
                    void method1();
                    void method2();
                  }
// interface X2 includes method1() and method2() of interface X1

interface X2 extends X1
{ void method3();  }   // adds method3()

// Class must implement all the methods of X1 and X2
class TestClass implements X2
{
     public void method1()
                    {System.out.println("method 1"); }
     public void method2()
                    {System.out.println("method 2"); }
     public void method3()
                    {System.out.println("method 3"); }

}
class InterfaceExtend
```

```
{
    public static void main(String arg[])
    {
        TestClassobj = new TestClass();
        obj. method1();
        obj. method2();
        obj. method3();
    }
}
```

After compiling and running the above program, it will shows output as shown in following figure 8.32:

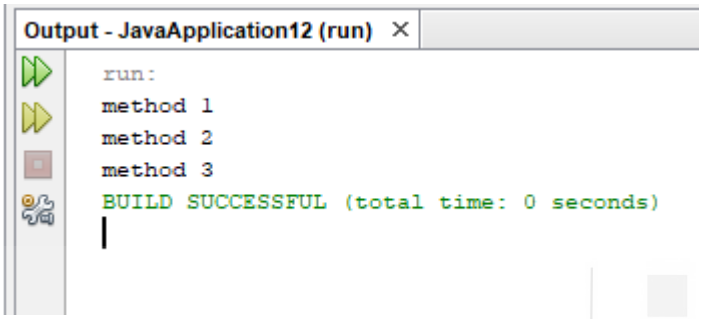


Figure 8.32: Output screen for Extending Interface Example

The file structure of the above Extending Interface Example is shown in figure-8.33

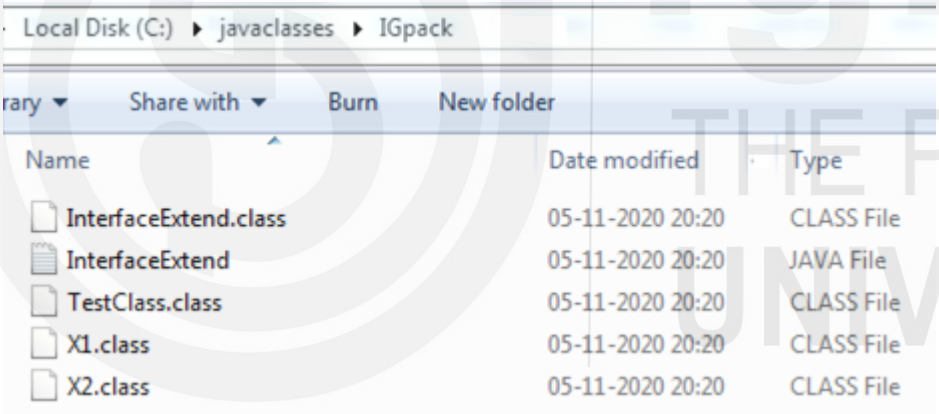


Figure 8.33: File Structure for Extending Interface Example

☞Check Your Progress-2

1) Explain the concept of an interface.

……

……

……

……

2) Can you extend interfaces in Java? Explain with example.

……

……

……

……

3) Which of these field declarations are valid for an interface?

Select the two correct answers.

a) public static int area = 40;

b) private final static int area = 40;

c) final static int area = 40;

d) int area;

……

……

……

4) What is the difference between interface and abstract class ?

……

……

……

5) Write a java program to find an area of rectangle using Interface. For this program, create a class Rectangle that implements an interface 'FindArea' which contains an abstract method 'Area'. You can specify the length and width values in a program.

……

……

……

8.10 DEFAULT INTERFACE METHODS

In the preceding sections, you have learned that all the methods of interfaces are public and abstract by default. In this section, you will learn how to declare and use default methods. You can also define default as well as static methods in the interface. In the section 8.12, you will find out about the use of static methods in the interface.

If you want to add new methods in the interface, it will require change in all the implementing classes. The solution of this situation is default method. The default method allows the java developers to add new methods in the existing interfaces without affecting the classes that implement these interfaces. It is not mandatory to provide implementation for default methods of interface by the implementing class. You can override these methods in the classes that implement these interfaces.

Default methods are those methods that are defined in the interface with the keyword default. There is no need to specify the public modifier in default methods, it is implicitly defined public. For example:

```
interface defaultInterface
{
default void method1()
     {
System.out.println("This is default method");
}
}
```

Example: The following program demonstrates the use of default method in Interface.

In this program, default method named method2() and another regular method named method1() are defined in an interface which are being called in the main() method of the Implementation Class 'example'.

```
interface defaultInterface
{
    public void method1();
    default void method2()      // declaration of default method
    {
        System.out.println("This is default method");
    }
}

// Implementation Class
class example implements defaultInterface
{
    public void method1()
    {
        System.out.println("This is regular method");
    }

    public static void main(String arg[])
    {
        example obj = new example();
        obj.method1();
        obj.method2();  //calling the default method of interface

    }
}
```

When you compile and run the above java program, it will show output as the following figure 8.34.

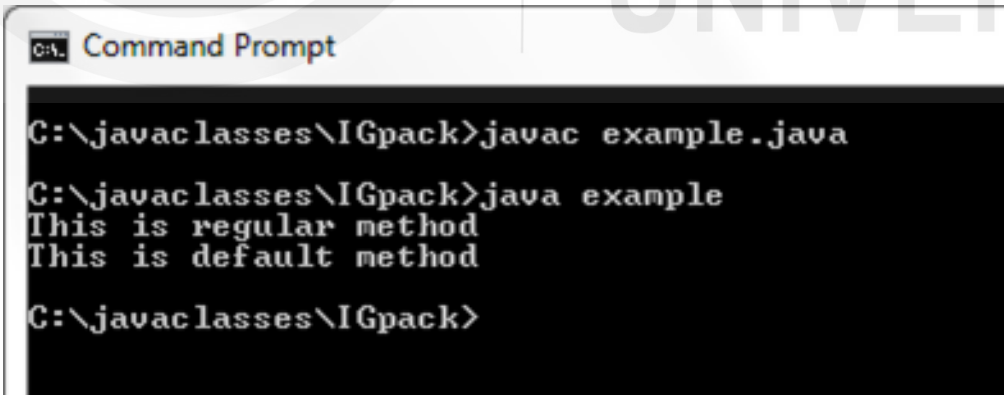


Figure 8.34: Command Prompt Screen for example of default method in Interface.

The file structure of the above example program is as follows:

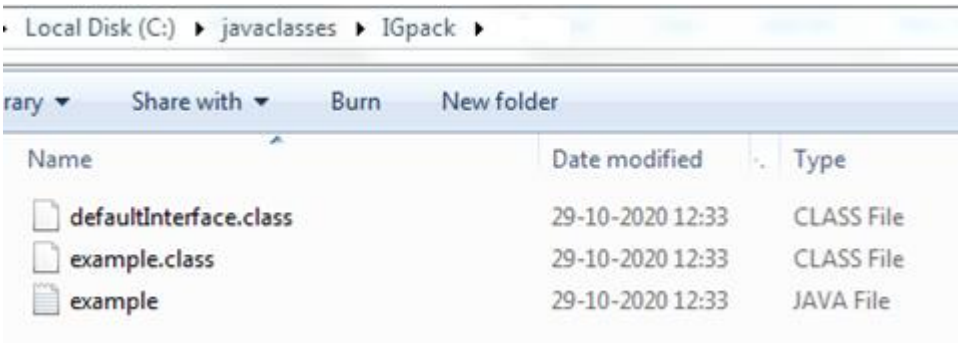


Figure 8.35: File structure of the above example

8.11 ISSUES OF MULTIPLE INTERFACES

You have been gone through about the inheritance in Unit 1 Block 2 of this course. You already know that in multiple inheritance a class inherits properties of more than one parent class. *Java is object oriented language but does not support multiple inheritances* in *the case of class.* On the other hand, there is another mechanism for achieving multiple inheritances through the Interfaces because a class can implement multiple interfaces in java.

Consider the following example through which you can understand the issues of multiple inheritances.

```
// First Parent class
class P1
{
   void method1()
   {
System.out.println("method of Class P1");
   }
}

// Second Parent Class
class P2
{
   void method1()
   {
System.out.println("method of Class P2");
   }
}

// Error : Demo class is inheriting from multiple classes
class Demo extends P1, P2
{
  public static void main(String args[])
  {
     Demo d = new Demo();
     d.method1();
  }
}
```

Output of the above program is compilation error as shown in figure 8.36:

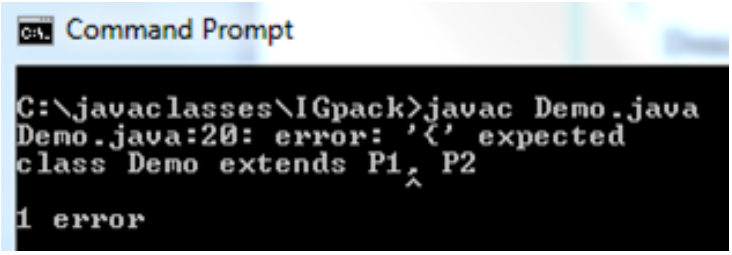


Figure 8.36: Output of the above program

You have seen in the above program, both the parent class P1 and P2 have methods with the same signature. Upon calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

Java provides solution for the above problem in the form of multiple interfaces and class can implement two or more interfaces.

For example, if both the implemented interfaces contain default methods with the same method signature then implementing class should explicitly specify which default method is to be used or it should override the default method.

```
interface P1
{
     default void display()
{
System.out.println("Default method: Interface P1");
  }
}

interface P2
{
default void display()
{
System.out.println("Default method: Interface P2");
 }
}

// Implementation class
class TestMulti implements P1, P2
{
   // Overriding default display() method
   public void display()
{
 /* use super keyword to call the display method of P1 interface */
P1.super.display();

/* use super keyword to call the display method of P2 interface */
 P2.super.display();
}

   public static void main(String args[])
   {
TestMulti t = new TestMulti();
t.display();
   }
}
```

When you compile the above Java program, it will show output looks like the following figure 8.37

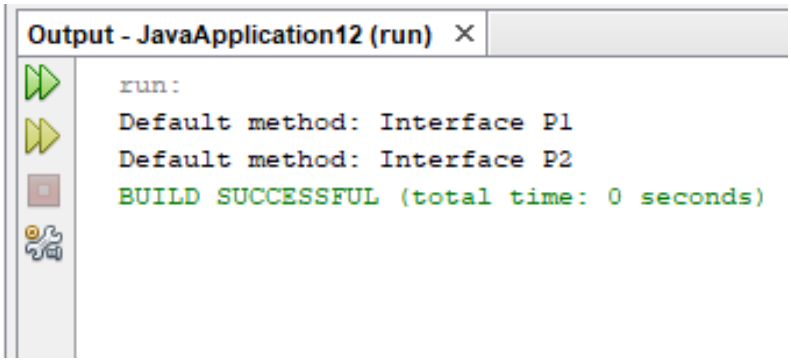


Figure 8.37: Command Prompt Screen for test multiple inheritance through interfaces program example

8.12 USE OF STATIC METHODS IN AN INTERFACE

In the preceding section, you have gone through the default method in the interface. This section will give you a demonstration of the use of static methods in an interface.

As the name static indicates that once it is defined, you can use it but cannot change it. When you use static method in an interface then calling class cannot change in static method. You can declare it with the static keyword at the beginning of the method signature and they provide an implementation.

The interface static method is similar to the default method of interface but it cannot be overridden in implementation Classes. The static method is dissimilar to other regular methods which are defined in the Interface; the static method contains the complete definition of the function and since the definition is completed, the said method becomes static. Therefore, these methods cannot be overridden in the implementation class. Interface name should be instantiated with it(static method) to use a static method as it is a part of the Interface only.

Example: The following Java program demonstrates the use of the static method in Interface.

```
interface staticInterface
{
     //regular method
      public void method1();

     // default method
     default void method2()
     {
     System.out.println("The default method is define in interface"); }
          // static method
     static void method3()
     { System.out.println("This is Static Method");   }
 }
// Implementation Class
class StaticExample implements staticInterface
{
     public void method1()
```

```
    { System.out.println("This is regular method");  }

    public static void main(String arg[])
    {
        staticexampleobj = new staticexample();

        // Calling the abstract method of interface
        obj.method1();

        // Calling the default method of interface
        obj.method2();

        // Calling the static method of interface
        staticInterface.method3();
    }
}
```

When you compile and run the above Java Program, it will show output as like the following figure 8.38:

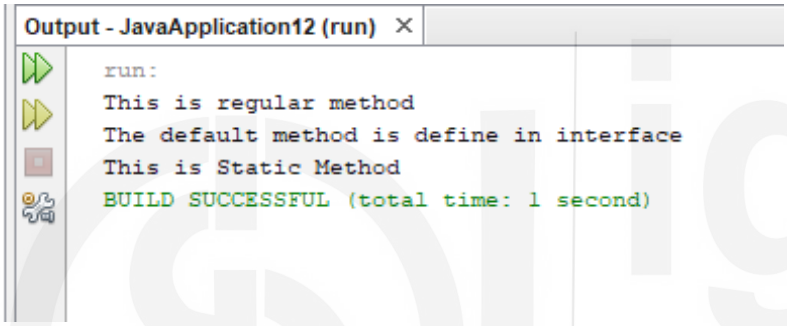


Figure 8.38: Output Screen for use of Static method in Interface

The file structure of the above java program is looks like the following figure 8.39:

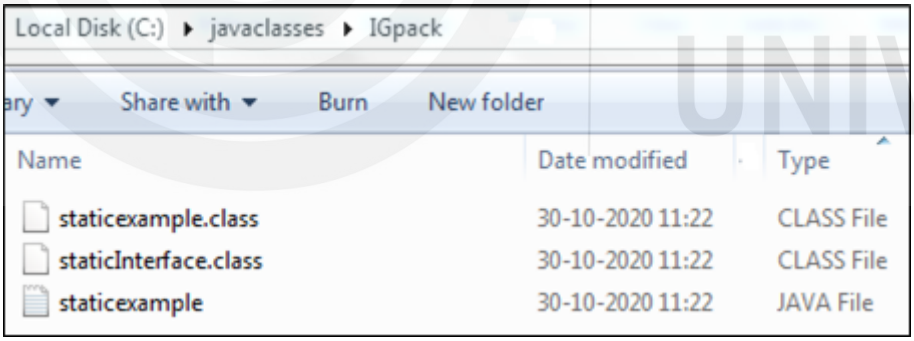


Figure 8.39: File structure of the above java program

☞Check Your Progress-3

1) What are the default methods in interface?

……

……

……

2) What is the use of static methods in interfaces?

……

……

3) What is the difference between static and default methods in interfaces?

…….

…….

…….

4) What is incorrect with the following interface?

```
public interface A
{
  void method1(int avalue)
  {
  System.out.println("Hello! Student");
   }
}
```

…….

…….

…….

5) Write a Java Program to add two numbers using static method in interface. You can specify input values in a program.

…….

…….

…….

8.13 SUMMARY

In this unit you have learned mainly two important features namely packages and interfaces. The role of packages is to make groups of related classes and interfaces. So, a package is collection of classes, interfaces and sub packages. The package is a grouping mechanism in which related class files are grouped and made available to other applications and other parts of the same application. It provides a way to organize related classes and interfaces to avoid naming conflicts. You can put classes that you developed in packages and distribute the package to others. Java language itself comes with a rich set of packages which are standard packages. This unit make you able to differentiate between built-in packages and user-defined package. You are also able to create your own package and importing packages in Java program. This unit has also explained to you the interfaces. Now you are able to create interface and use them in a class. With interfaces, you can obtain the effect of multiple inheritances. With the introduction of default methods, you can add additional features to the interfaces without affecting the end-user classes. You have also learned about the use of static methods in interface. Now you are fully able to use packages and interfaces in Java program.

8.14 SOLUTIONS/ANSWER TO CHECK YOUR PROGRESS

☞Check your Progress-1

1) The package is a grouping mechanism in which related class files are grouped and made available to other applications and other parts of the same application. So, the package is a collection of related classes and interfaces. The package declaration should be the first statement in a Java class.

   Packages offer several advantages like the following:

   - You can define their own packages to bundle a group of classes and interfaces, etc.
   - Using Package, you can avoid naming conflicts.
   - It is a good practice to group related classes implemented by you so that a programmer can easily determine the related classes, interfaces, enumerations, and annotations.
   - Using packages, it is easier to provide access control and locate the related classes.

   The JDK includes the default package name as java.lang package. This package provides the fundamental classes that are necessary to design a basic Java program. The important classes are Object, which is the root of the class hierarchy and Class, instances of which represent classes at run time.

2) Two types of packages in Java are Standard or Built-in packages and User-defined packages. Built-in packages are those packages which Java API provides to simplify the task of Java programmer. The packages which the users create are called as User-defined package.

3) The Java command for compiling Java source code using the following syntax:

   javac –d . filename.java

   For example: javac –d . student.java

   For executing package programs, the fully qualified name of a class is <package>.<classname>. For example: java javapackagename.student

4) Java has an import keyword, which is used to access java package and its classes into the java program. You can import any specific package member or import all the types contained in a particular package. The import statement is written directly after the package statement (if it exists). You can write more than one import statements.

☞Check your Progress-2

1) Interface can contain only constants declaration, method declaration, default methods, static methods and nested types. The interface contains one or more method declarations without their implementations. Once it is defined, any numbers of classes can implement an interface. The Interface keyword is used to create an interface. An interface is a powerful mechanism for defining behaviour among unrelated classes.

2) An interface can extend other interface using the keyword extends. The syntax is similar to inheriting classes. When a class implements an interface that inherits other interface, then the class must implement all methods defined within the interface inheritance chain. For example, please refer to section 8.9

of this unit.

3) The correct answers are (a) and (c). Fields in interfaces declare named constants and are always public, static and final. None of these modifiers are mandatory in a constant declaration. All named constants must be explicitly initialized in the declaration.

4) The difference between interfaces and abstract classes are as under:

- Interface can be implemented using 'implements' keyword whereas abstract class can be inherited using 'extends' keyword.
- Interface can only have public members, while abstract class can have any type of members like public, private etc.
- Variables declared in a Java interface is by default final, whereas an abstract class may contain non-final variables.
- Interface is completely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated but can be invoked if a main() exists.
- An interface can extend another Java interface only, while an abstract class can extend another Java class and implement multiple Java interfaces.

5)
```
// create an interface
interface FindInterface
{
      void Area(int length, int width);
}

// implement the FindInterface interface
class Rectangle implements FindInterface
{
      // implementation of abstract method of interface
        public void Area(int length, int width)
      {
       System.out.println("Area of the rectangle = " + (length * width));
      }
}
class Test
{
      public static void main(String[] args)
      {
         // create an object
            Rectangle r = new Rectangle();
         r.Area(8, 9);
       }
}
```

☞Check your Progress-3

1) When you want to add new functionality to an existing interface then you can use default methods. For creating default methods, use the default keyword and add the definition for the method. Default methods can be provided to an interface without affecting implementing classes as it includes an implementation. If each added method in an interface is defined with implementation, then no implementing class is affected. An implementing class can override the default implementation provided by the interface.

2) You can have static methods in an interface (with body), and it cannot be override in implementation Classes. If your interface has a static method you need to call it using the name of the interface, just like static methods of a class.

3) When you want to add new functionality to an existing interface without breaking the rules then you can use static methods. Static methods in an interface are similar to default methods, but they cannot be overridden in implementation Classes. The static methods are declared by using the static keyword. You can access static methods by using the interface name.

4) It has a method implementation in it. Only default and static methods have implementations.

5)
```
interface staticInterface
{
      // static method
    static void add()
    {
      int x, y, z;
      x=2;
      y=3;
      z=x+y;
      System.out.println("Static Method in an Interface");
      System.out.println("X + Y =" + z);
    }
 }
// Implementation Class
class staticexample implements staticInterface
{
public static void main(String arg[])
{
    // Calling the static method of interface
    staticInterface.add();

}
}
```

8.15 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" Vol. 2. Pearson Education, 2013.
- Prasanalakshmi, "*Advanced Java Programming*", CBS Publishers 2015
- Sagayaraj, Denis, Karthik and Gajalakshmi ,"*Java Programming – for Core and Advanced Users*", Universities Press 2018 .
- Khalid A. Mughal and Rolf W. Rasmussen, " Aprogramer's Guide to Java Certification", Addison-Wesley Professional, 2003.
- https://www.w3schools.com/java/
- https://docs.oracle.com/javase/tutorial/java/package/packages.html
- The Complete Reference Java – Patrick Naughton and Herbert Schildt

ignou
THE PEOPLE'S
UNIVERSITY