

---

# **UNIT 1 INHERITANCE**

---

<b>Structure</b>	<b>Page no.</b>
1.0 Introduction	
1.1 Objectives	
1.2 Basics of Inheritance	
1.3 Advantages of Inheritance	
1.4 Member Access and Inheritance	
1.5 Types Of Access Specifiers	
1.6 Types of Inheritance	
1.7 Use of Super Keyword	
1.7.1 To call the superclass constructor	
1.7.2 To access the member of the superclass	
1.8 Creating Multilevel Class Hierarchy	
1.9 Demonstrating Order of Constructors Execution	
1.10 Use of Final Keyword	
1.11 Summary	
1.12 Solutions/Answer to Check Your Progress	
1.13 References/Further Reading	

---

## **1.0 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. The 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.

---

## **1.1 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.

---

## **1.2 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

## Inheritance

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.

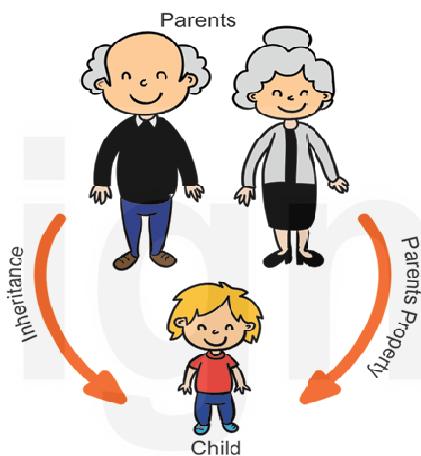


Figure 1: Inheritance- parents to child

Figure 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 minimise 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.

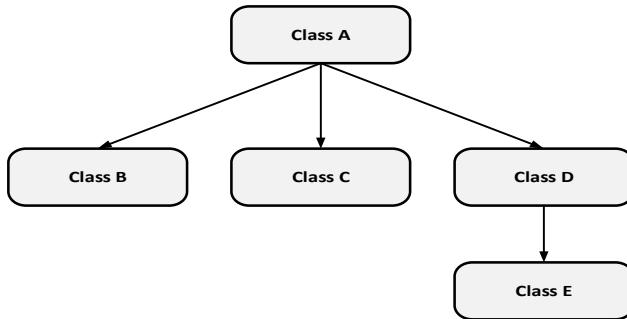


Figure 2: Class hierarchy

In figure 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 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 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.

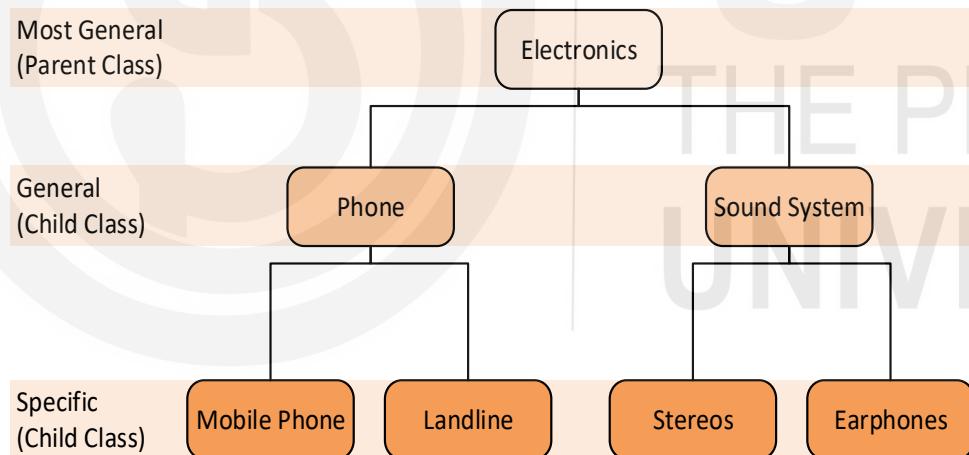


Figure 3: Hierarchy and generality of classes

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 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.

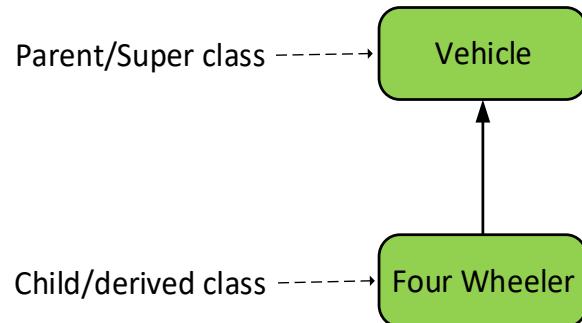


Figure 4: Super and Sub-class relation

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

```

### Output:

```

InheritanceExample >
Output - JavaApplication7 (run) X
run:
Vehicle is Running
It is four-wheeler
BUILD SUCCESSFUL (total time: 1 second)

```

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().

## 1.3 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 like 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 and also in the next unit of this block.

## 1.4 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 like, 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 below given example.

```
/*--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:



```
Output - InheritanceExample (run) ×
run:
Value of i is:15 and Valuee of j is: 30
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - j has private access in inheritanceexample.A
    at inheritanceexample.B.sum(InheritExp.java:27)
    at inheritanceexample.InheritExp.main(InheritExp.java:36)
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 3 seconds)
```

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.

---

## 1.5 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 1.

**Table 1 Summary of Access Specifier**

	High restriction ----→ Low restriction			
	Private	Default	Protected	Public
<b>Inside Class</b>	YES	YES	YES	YES
<b>Inside Package</b>	NO	YES	YES	YES
<b>Outside Package subclass</b>	NO	NO	YES	YES
<b>Outside Package</b>	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:

```
Jtest.Property > running >
Refactoring Output - JavaApplication7 (run) X
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - running has private access in Jtest.Property
    at Jtest.Car_Main.main(Car_Main.java:14)
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 2 seconds)
```

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");
    }
}
```

Inheritance,  
Polymorphism and  
Packages

```
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**

Q1. 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
```



Q2. Find out the output of the program given below.

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

        return i;
    }
}

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

        return methodA(i);
    }
}

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

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

Q3. Why you can 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 = new Ram();  
}
```

Q4. Is the code given below written correctly? If yes then what will its output?

```
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();  
    }  
}
```

## 1.6 TYPES OF INHERITANCE

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

**Table 2 Types of Inheritance**

Name of Inheritance	Block Diagram	Syntax
Single Inheritance	<pre> graph TD     ClassA[Class A] --&gt; ClassB[Class B]   </pre>	<pre> public class A {     ----- } public class B extends A {     ----- }   </pre>
Multi-Level Inheritance	<pre> graph TD     ClassA[Class A] --&gt; ClassB[Class B]     ClassB --&gt; ClassC[Class C]   </pre>	<pre> public class A {     ----- } public class B extends A {     ----- } public class C extends B {     ----- }   </pre>
Hierarchical Inheritance	<pre> graph TD     ClassA[Class A] &lt;--&gt; ClassB[Class B]     ClassA &lt;--&gt; ClassC[Class C]   </pre>	<pre> public class A{     ----- } public class B extends A{     ----- } public class C extends A{     ----- }   </pre>

<b>Multiple Inheritance</b>	<pre> graph TD     ClassA[Class A] --&gt; ClassB[Class B]     ClassC[Class C] --&gt; ClassB   </pre>	<pre> public class A {     ----- } public class B {     ----- } public class C extends A, B {     ----- }   </pre> <p>//Java does not support multiple inheritances</p>
-----------------------------	--	---

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.

## 1.7 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.

### 1.7.1 Calling the 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;
    }
}
  
```

## Inheritance

```
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 the program:**

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 even in a multileveled hierarchy also.

### **1.7.2 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");
    }
}

```

## Inheritance

```
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.

## 1.8 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 4. Percentage can be a subclass of Marks, which is further a subclass of Student.

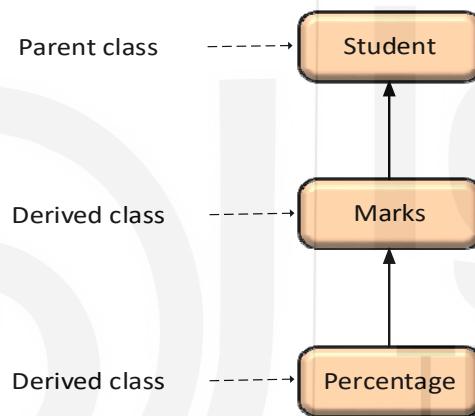


Figure 5 Multi-level class hierarchy

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
  
```

## Inheritance

```
{  
    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 forming 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

Q1. 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();
    }
}
```

Q2. 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);
    }
}
```

## Inheritance

```
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;
}
```

Q3. What do you mean by Generalized and Specialized classes in Java? Explain it with the help of a program.

---

## 1.9 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 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();
}
}

```

Inheritance,  
Polymorphism and  
Packages

#### **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.

---

## **1.10 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 can not 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
{
```

## Inheritance

```
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: can not 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 can not 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 can not 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

Q1. What do you mean by method hiding in Java?

---



---



---



---

Q2. The code given below shows a compile-time error. Can you suggest the appropriate corrections?

## Inheritance

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

class Y extends X
{
    public Y()
    {
        System.out.println("Child Class.");
    }
}
```

---

Q3. 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);
    }
}
```

---

Q4. Can you guess? How many public classes a Java program can have?

---

---

---

---

## 1.11 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 been discussed.

## 1.12 SOLUTIONS/ANSWER TO CHECK YOUR PROGRESS

### Check your progress 1:

**Answer 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<sub>1</sub>* and class *Parent<sub>2</sub>* 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.

**Answer 2:** The output of the program is: 200

### Answer 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 can not be instantiated outside the package, even though if it has a public constructor.

### Answer 4:

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

### **Check your progress 2:**

**Answer 1.** The output of the program is given below.

You are in class A.

You are in class B

Here, Now in class C

### **Answer 2**

The output of the program is:

21

32

25

### **Answer 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:**

#### **Answer 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.

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

class X

### Inheritance

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

### Answer 3:

Welcome to Base class

Now it is Derived class

Now it is Derived class

**Answer 4:** A Java program can have maximum only one public class.

---

## 1.13 REFERENCES/FURTHER READING

---

- ... 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”, Apress, Berkeley, CA, 2020.

# UNIT 2 POLYMORPHISM

Structure	Page no.
2.0 Introduction	
2.1 Objectives	
2.2 Introduction to Polymorphism	
2.3 Advantages of Polymorphism	
2.4 Types of Polymorphism	
2.5 Method of Overloading	
2.6 Method of Overriding	
2.7 Abstract Class	
2.8 Application of Abstract Class	
2.9 Summary	
2.10 Solutions/Answer to Check Your Progress	
2.11 References/Further Reading	

## 2.0 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 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 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 taking many roles.

Another example may be the saving of two contact numbers of the same person. What 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.

## 2.1 OBJECTIVES

After reading this unit, you will be able to:

- ... Explain the basics of polymorphism,
- ... Write the program on method overloading 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.

## 2.2 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.

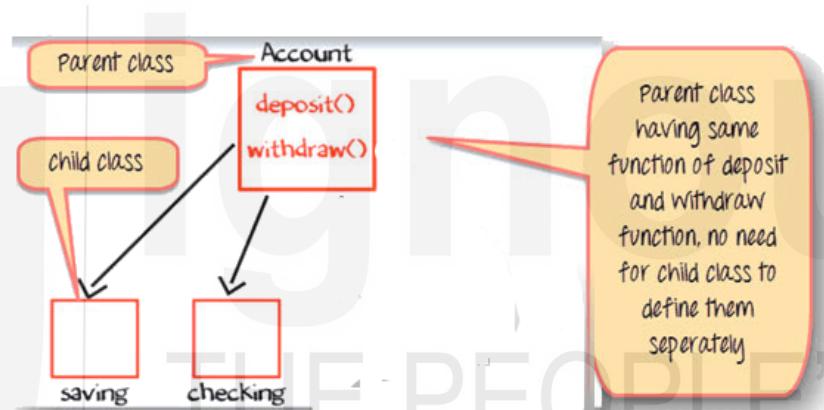


Figure 2 A scenario of inheritance

In figure 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.

The concept of polymorphism in Java makes it possible to perform a single action in different ways with the idea of reusability. In figure 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.



Figure 3: Single action in different ways

Let us discuss the programming aspects of polymorphism of Java with the help 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( ).

#### Programming Example 1:

```
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.

### 2.3 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.

## 2.4 TYPES OF POLYMORPHISM

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

- ... Static Polymorphism
- ... Dynamic Polymorphism

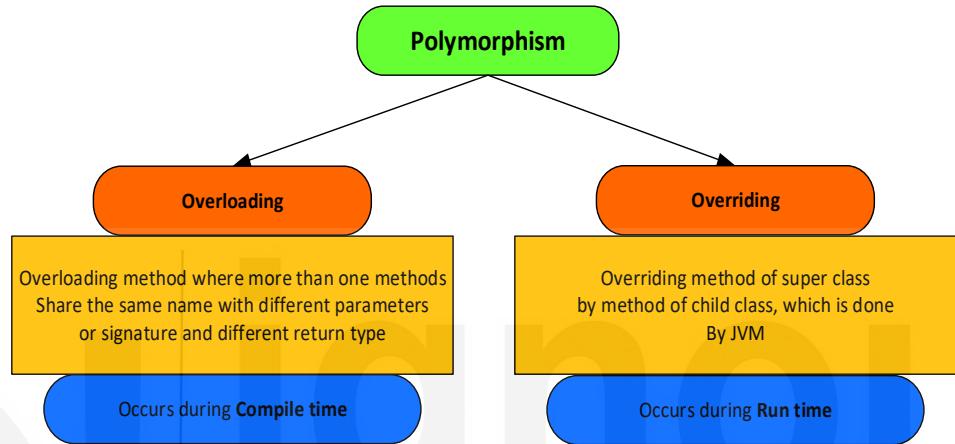


Figure 4. Types of Polymorphism

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.

## 2.5 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 parameters 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)

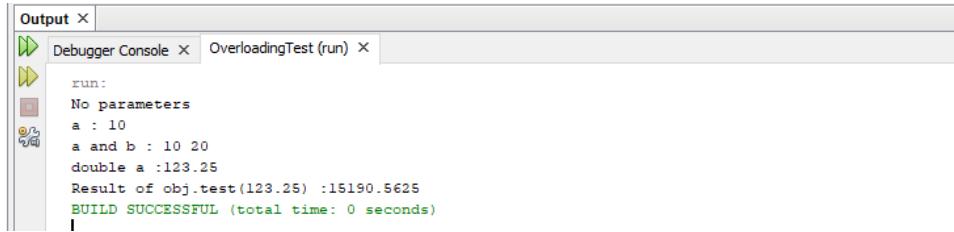
3. There is a different sequence of data types of the parameters.

**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[])  
{  
DemoOverload obj = 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:**



The screenshot shows an 'Output' window from an IDE. It has tabs for 'Debugger Console' and 'OverloadingTest (run)'. The 'OverloadingTest (run)' tab is active and displays the following text:  
run:  
No parameters  
a : 10  
a and b : 10 20  
double a :123.25  
Result of obj.test(123.25) :15190.5625  
BUILD SUCCESSFUL (total time: 0 seconds)

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**

Q1. What is method overloading?

-----  
-----  
-----  
-----

Q2. What is compile-time polymorphism?

-----  
-----  
-----  
-----

Q3 Write a program to demonstrate the overloading of three methods with the same name.

-----  
-----  
-----  
-----

---

## 2.6 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);
```

ignou  
THE PEOPLE'S  
UNIVERSITY

```
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 of the program:**

```
run:
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
rea for Figure is undefined.
Area is 0.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

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 bounded with class. Hence, it cannot be overridden.
5. The access level of methods cannot be more restrictive than the overridden method's access level.

---

## 2.7 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%

## Polymorphism

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.

## 2.8 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 4 of this course. If you need to know more about interface in detail please refer unit 4 of this block then return back here to proceed further. Before discussing the application, let's have a look on the comparative chart 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 as 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 implements many java interfaces.	3. Only one java interface can be extended.
4. Members of abstract class can 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.

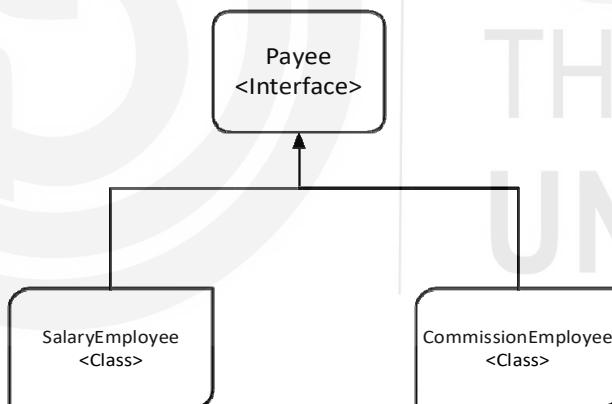


Figure 5. A scenario of interface implementation

/\*-----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.

## Polymorphism

```
/*-----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, 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.

## Polymorphism

```
/*-----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)
    {
        PaymentSystem paymentSystem = new PaymentSystem();
        CommissionEmployee raviSingh = new CommissionEmployee("Ravi Singh", 5000, 301.0);
        paymentSystem.addPayee(raviSingh);
        CommissionEmployee sksingh = new CommissionEmployee("Sunil Singh", 6000, 545.0);
        paymentSystem.addPayee(sksingh);
        SalaryEmployee maryBrown = new SalaryEmployee("Mary Brown", 7000, 500.0);
        paymentSystem.addPayee(maryBrown);
        SalaryEmployee susanWhite = 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 of the program:**

```
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,

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

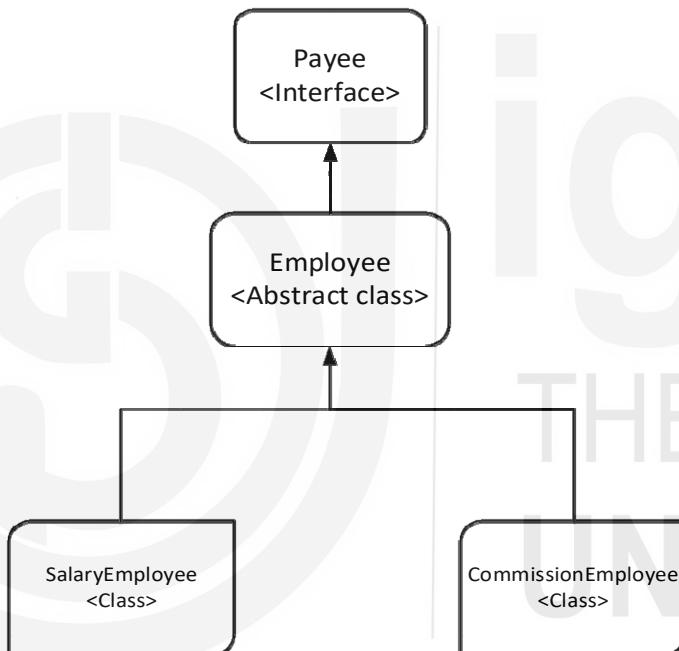


Figure 6 A Scenario to interface class

**CHECK YOUR PROGRESS-2**

Q1. Why do we use polymorphism in Java?

---

---

---

---

Q2. What do you mean by polymorphic parameters? Explain it with the help of suitable program.

Q3 Find the answers to the program given below.

Its 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);
    }
}
```

---

---

---

Q3 Find the answers of the program given below.

Its return type is different, while method 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);  
}  
}
```

---

---

---

---

## 2.9 SUMMARY

In this unit, we have discussed the important concept known as polymorphism. This unit also discussed the types of polymorphism. Also, 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 have discussed, code reusability concept using abstract class instead of using the implement method of interface.

## 2.10 SOLUTION/ANSWER TO CHECK YOUR PROGRESS

### Check you progress 1:

#### 1. Answer 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. Answer 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. Answer 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( String args[]){  
    Summation ob = 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 advantages 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. Let's have a look at the example given below, in which *content* is defined as string once and 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 argument list cannot be defined in a same class.

3. Run this program you will observe that it will throw a compilation error, because you can not 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.

Inheritance,  
Polymorphism and  
Packages

## 2.11 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 3 ASSERTIONS, ANNOTATIONS AND EXCEPTION HANDLING**

---

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Assertion and its use
- 3.3 Annotations
  - 3.3.1 Built-In Java Annotations
- 3.4 Exception Handling
  - 3.4.1 Checked and Unchecked Exceptions
  - 3.4.2 Using try, catch and finally method to handle exception
- 3.5 Throw and Throws
- 3.6 Final vs Finally vs Finalize
- 3.7 User Defined Exception
- 3.8 Summary
- 3.9 Solutions/ Answer to Check Your Progress
- 3.10 References/Further Reading

---

### **3.0 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 over time in the language itself. In the third and final part of this unit 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 do 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 to learn that how apt and wide coverage of exceptions is provided in java. Towards the end you will learn to define your own Exception.

---

### **3.1 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,
- Difference between throw and throws,
- Use final finally and finalize keywords, and
- Define your own exception methods.

---

## 3.2 ASSERTIONS AND ITS USE

---

We all would have heard about the role of a proofreaders in 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, isn't it? 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 isn't 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 them, to 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 **AssertionError** to be thrown. The execution of AssertionError 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

Lets 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 ( barring invocation of a method that is declared void.)

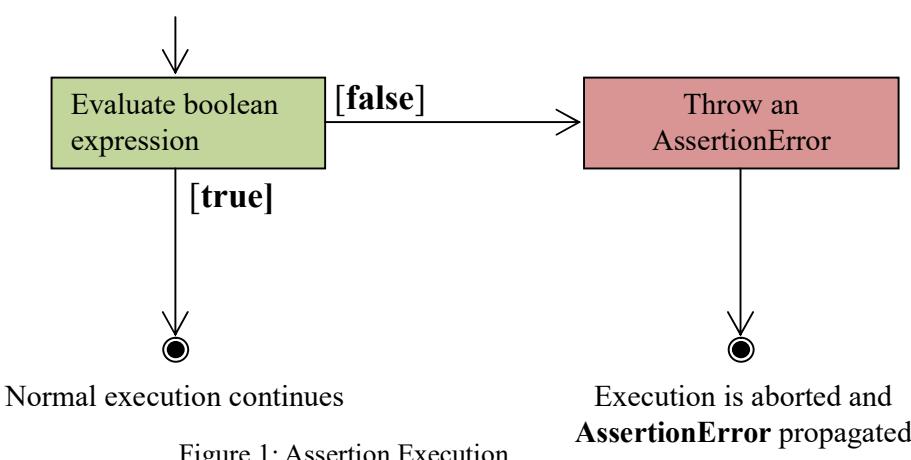


Figure 1: Assertion Execution

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 :

```
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_connexion ()  
{  
    Connection conn = getConnection ();  
    assert conn == null;  
}
```

The above assert, also can be 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 ( remember assertions 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 utilised in 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 and 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 dayOfWeek – the parameter passed to switch-case has a value other than the valid days, an AssertionError 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?

---

---

---

---

### 3.3 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 to find 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.

### 3.3.1 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.

```
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:

The screenshot shows an IDE interface with a code editor and a console window. The code editor contains the following Java code:

```
1 class Animal{
2     void eatSomething(){
3         System.out.println("eating something");
4     }
5 }
6
7 class Dog extends Animal{
8     @Override
9     void eatsomething(){System.out.println("eating foods");}//should be eatSomething
10 }
11
12 class TestAnnotation1{
13     public static void main(String args[]){
14         Animal a=new Dog();
15         a.eatSomething();
16     }
17 }
```

The IDE highlights the misspelling "eatsomething" in red. The console window below shows the output of the program:

```
<terminated> Test (1) [Java Application] /usr/lib/jvm/java-ibm-x86_64-80/jre/bin/javaw (13-Sep-2020, 2:36:12 pm)
eating something
```

Output once the correction is made :

The screenshot shows the same IDE interface after the spelling mistake has been corrected. The code editor now displays:

```
1 class Animal{
2     void eatSomething(){
3         System.out.println("eating something");
4     }
5 }
6
7 class Dog extends Animal{
8     @Override
9     void eatSomething(){System.out.println("eating foods");}//should be eatSomething
10 }
11
12 class TestAnnotation1{
13     public static void main(String args[]){
14         Animal a=new Dog();
15         a.eatSomething();
16     }
17 }
```

The IDE no longer highlights the misspelling. The console window shows the corrected output:

```
<terminated> Test (1) [Java Application] /usr/lib/jvm/java-ibm-x86_64-80/jre/bin/javaw (13-Sep-2020, 2:37:54 pm)
eating foods
```

```
class Viral
{
    void testVirus(){System.out.
}

class Covid extends Viral
{
    @Override
    void testvirus(){System.out.
```

```
}
```

```
class VerifyAnnotation1
{
    public static void main(String args[])
    {
        Viral v=new Covid();
        v.testVirus();
    }
}
```

Output when there is a spelling mistake:

Replace

Output once the correction is made :

Example 2 :

```
class ViralTest
{
    public void conductTest()
    {
        System.out.println("This is test for presence of Vital 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 annotation 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, therefore avoid use of such methods.

Example 1:

```
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 2:**

```
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 ?

---

---

---

---

---

### 3.4 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 the 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 a programmer 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.

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 2 given below show the Throwable class and its subclasses.

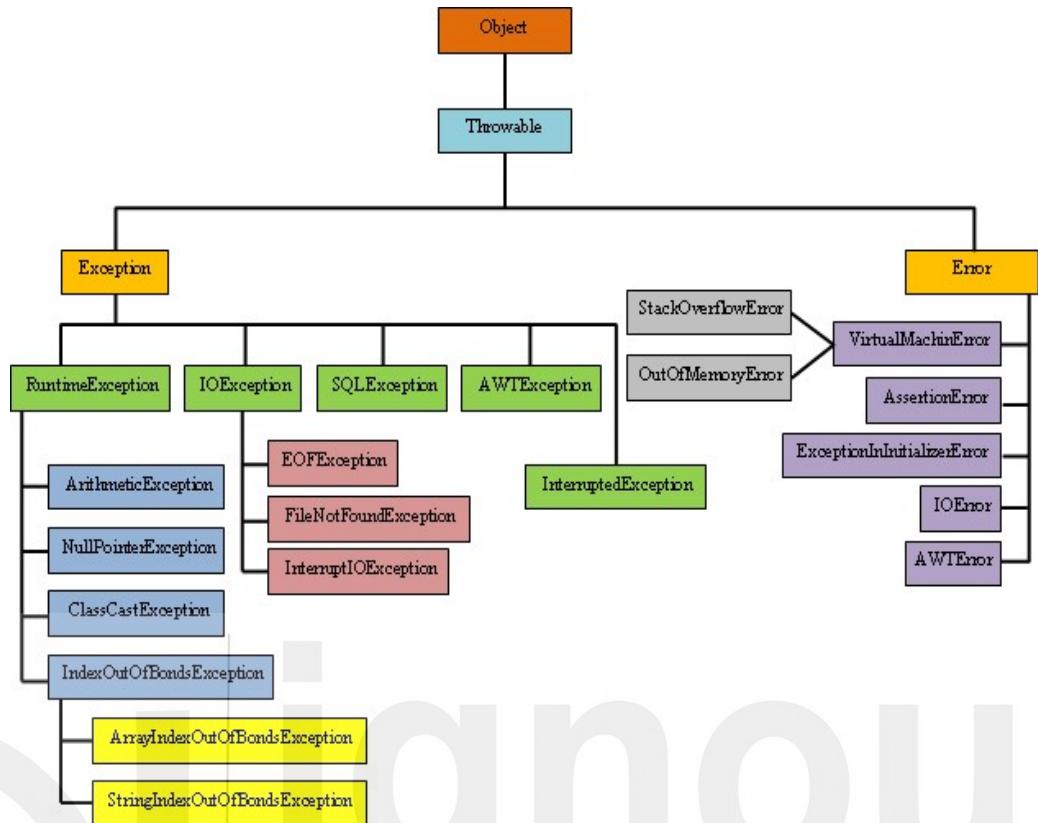


Figure 2: Throwable Class Hierarchy in Java

### Exception class and its Subclasses in Throwable class Hierarchy

Here one important and interesting question arises that how Java virtual machine handles 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 handing it to the runtime system is called '**throw'ing** an exception, then by using **try, catch and finally clauses** these exceptions can be handled in Java. This is how Java virtual machine handles exceptions internally.

In figure 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.

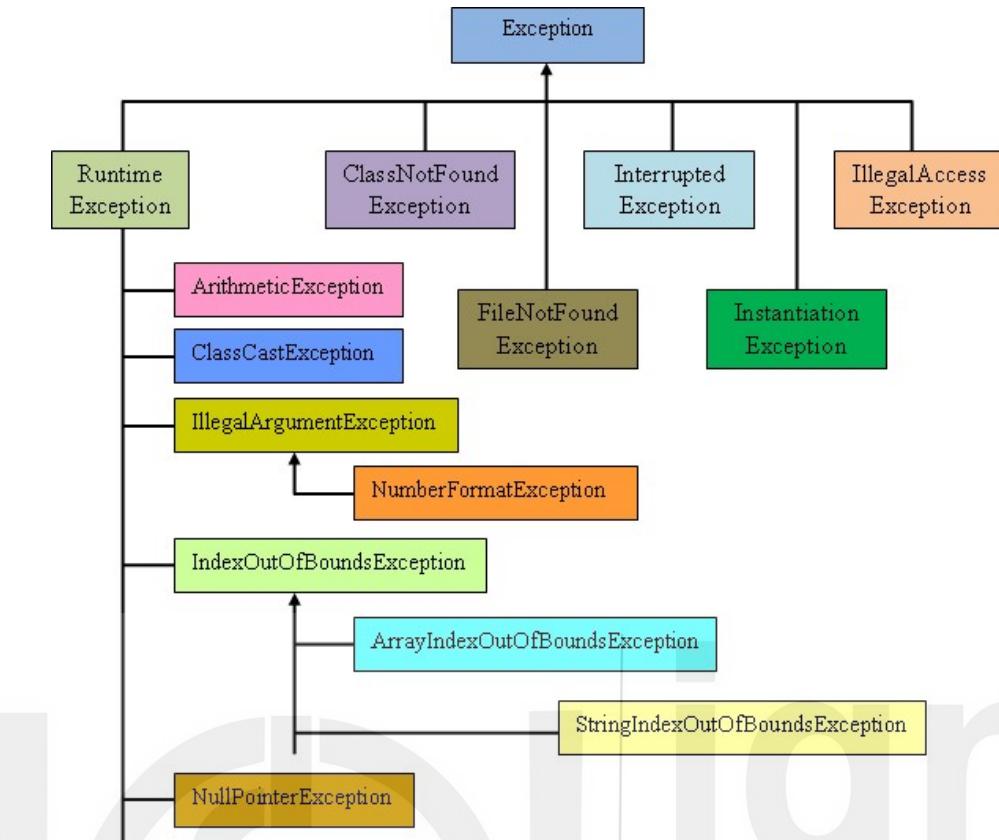


Figure 3: Exception Class and its subclasses

### 3.4.1 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 do not 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 PGDEception. Inside the class the first thing that we are going to do is that 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 PGDEception
{
    public static void main(String args[])
    {
        string str = null;
        system.out.println(str.length());
    }
}
```

### Output



```
Output - JavaApplication19 (run) ×
run:
Exception in thread "main" java.lang.NullPointerException
at ExceptionPGD.PGDEception.main(PGDEception.java:12)
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

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.

### 3.4.2 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

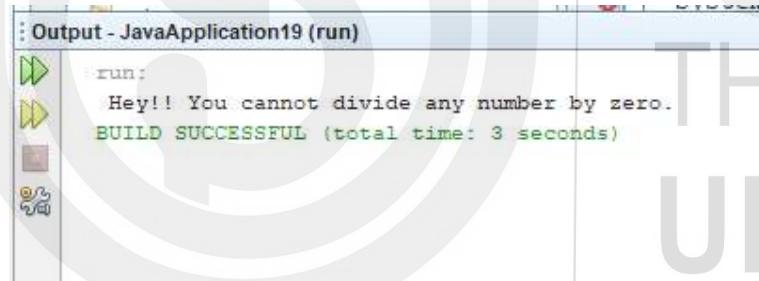
we try to divide a number by zero then it says we cannot divide the number by zero because it raises arithmetic exception a base arithmetic anomaly.

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's run the program and see the output.

```
package ExceptionPGD;

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

#### Output:



```
: Output - JavaApplication19 (run)
run:
Hey!! You cannot divide any number by zero.
BUILD SUCCESSFUL (total time: 3 seconds)
```

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.

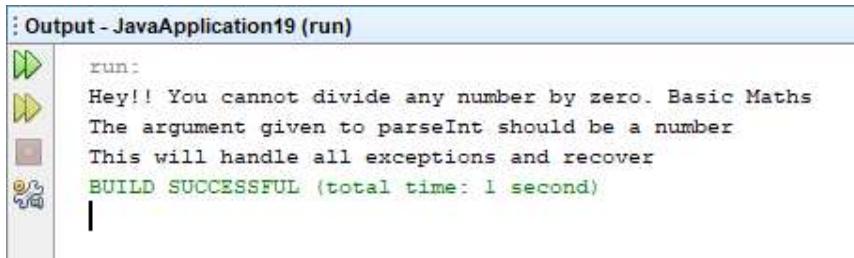
Built-in exceptions are the exceptions which are available in Java libraries and they are suitable to explain certain error situations like arithmetic exception, array index out of bound and class not found exception, runtime exception, number format exception etc.

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 PGDEception
{
    public static void main(String args[])
    {
        try
        {
            int a=90, b=0;
            int c=a/b;
            System.out.println("Resultant =" +c)
        }
        catch(ArithmetricException 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:**



```
: Output - JavaApplication19 (run)
run:
Hey!! You cannot divide any number by zero. Basic Maths
The argument given to parseInt should be a number
This will handle all exceptions and recover
BUILD SUCCESSFUL (total time: 1 second)
```

Now, let us 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.

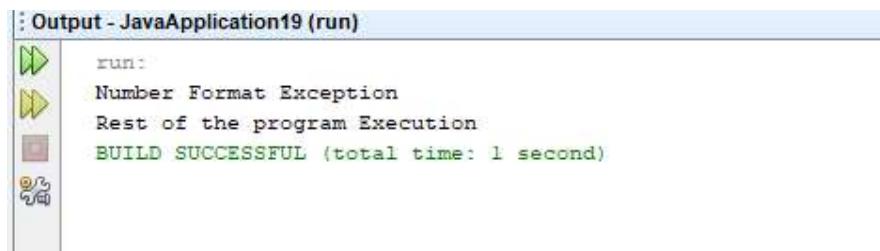
So here we have one try block and have different catch blocks. Now what we should practice while there is a need to handle such different types of exceptions while writing our code is to handle them from sub class exception to super class exception (specialized to generalized). Lets understand this with 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.

Lets see an example of catching multiple exceptions:

```
Package ExceptionPGD;
public class PGDEception
{
    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:**



```
: Output - JavaApplication19 (run)
run:
Number Format Exception
Rest of the program Execution
BUILD SUCCESSFUL (total time: 1 second)
```

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 finally block all the crucial statements are kept regardless of 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 PGDEception
{
    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:

```
: Output - JavaApplication19 (run)
run:
Finally block is always executed
Exception in thread "main" java.lang.NumberFormatException: For input string: "PGDegree"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at ExceptionPGD.PGDException.main(PGDException.java:9)
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 4 seconds)
```

## 3.5 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");
}
```

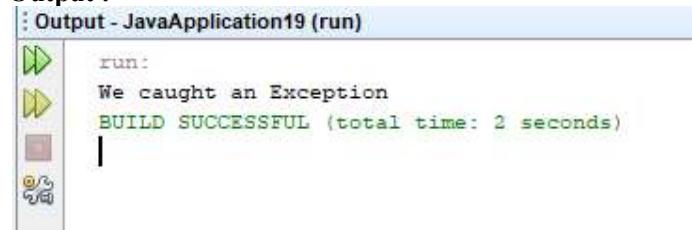
An example which demonstrate use of through clause in programming is given below.

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

```
{  
    avrg();  
}  
}
```

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

**Output :**



```
run:  
We caught an Exception  
BUILD SUCCESSFUL (total time: 2 seconds)
```

**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:

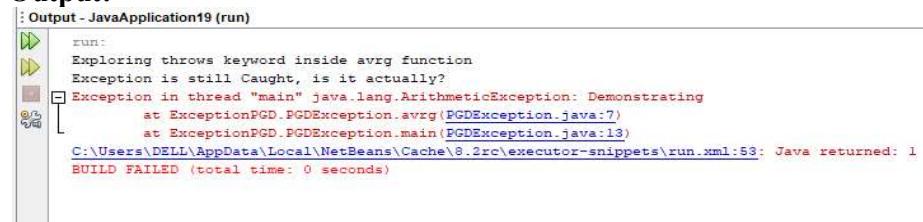
```
void a_method() throws ArithmeticException { }
```

An example to demonstrate use of throws clause.

```
package ExceptionPGD;  
public class PGDEception  
{  
    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:**



```
run:  
Exploring throws keyword inside avrg function  
Exception is still Caught, is it actually?  
Exception in thread "main" java.lang.ArithmetricException: Demonstrating  
    at ExceptionPGD.PGDEception.avrg(PGDEception.java:7)  
    at ExceptionPGD.PGDEception.main(PGDEception.java:13)  
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

## 3.6 FINAL VS FINALLY VS FINALIZE

What will happen 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.

## 3.7 USER DEFINED EXCEPTION

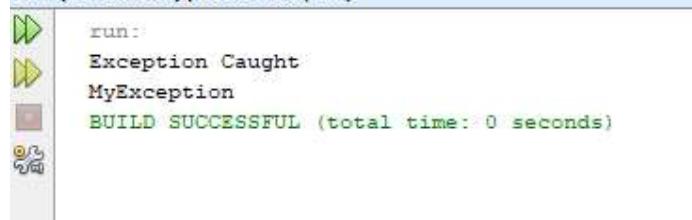
Sometimes the built-in exceptions in Java are not able to describe a certain situation. In such cases a user can also create exceptions and that are called user defined exceptions . There are two important points which are used while creating/defining user defined exception.

- 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 PGDEception
{
    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 :**  
: Output - JavaApplication20 (run)



```
run:  
Exception Caught  
MyException  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Another Example Program for Custom Exception

```
package javaapplication20;  
import java.util.*;  
  
class PGDEception  
{  
    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 - JavaApplication20 (run)
  run:
  Enter a threshold value less than 60 :
  45
  Valid Threshold
  BUILD SUCCESSFUL (total time: 3 seconds)
```

### Output-2 ( for input value 85)

```
: Output - JavaApplication20 (run)
  run:
  Enter a threshold value less than 60 :
  85
  You can be Covid positive it is an exception: threshold is greater than 60
  BUILD SUCCESSFUL (total time: 4 seconds)
```

Custom and User defined Exceptions come to our rescue in the rare situations where java already does not have 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. How to 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 PGDEception extends Exception
{
    String className;
    public static void main(String[] args)
    {
        try
        {
            PGDEception v = new PGDEception();
            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.
- 
- 
- 
-

## 3.8 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 and also learnt how we create our own Exceptions.

## 3.9 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 “AssertionError” when the assumption made is wrong. AssertionError 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 AssertionError.
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 AssertionError is given at runtime provided the assertion is enabled. If the assumption is correct, then the boolean condition will return true.
5. The AssertionError 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 MyOwnRTPCCTestException 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 *NullPointerException*. 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.

---

### 3.10 REFERENCES/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/>

THE PEOPLE'S  
UNIVERSITY

---

# **UNIT 4 PACKAGES AND INTERFACES**

---

## **Structure**

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Package
- 4.3 Access Rules in Packages
- 4.4 Finding Packages and CLASSPATH
- 4.5 Creating Own Packages
- 4.6 Importing Packages
- 4.7 Basics of Interface in Java
- 4.8 Defining, Implementing and Applying Interface
  - 4.8.1 Defining an Interface
  - 4.8.2 Implementing Interfaces
  - 4.8.3 Applying Interfaces
- 4.9 Extending Interfaces
- 4.10 Default Interface Methods
- 4.11 Issues of Multiple Interfaces
- 4.12 Use of Static Methods in an Interface
- 4.13 Summary
- 4.14 Solutions/ Answer to Check Your Progress
- 4.15 References/Further Reading

---

## **4.0 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.

---

## **4.1 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

## 4.2 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 is 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, 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.

### 4.3 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 previous units of this course. As yet, 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 are concerned:

- ... 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 1: Java access control mechanism for class member**

	<b>Public</b>	<b>Private</b>	<b>Protected</b>	<b>No Modifier</b>
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.

## 4.4 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. While packages solve many problems related to access control and namespace conflicts perspective. Also, it is important to note that, they become 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 1.

Local Disk (C:) ▶ javaclasses ▶ IGpack			
Category	Share with	Burn	New folder
Name		Date modified	Type
 socis.class		25-10-2020 15:02	CLASS File
 socis		25-10-2020 15:01	JAVA File

Figure 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 2.

```
C:\ Command Prompt
C:\javaclasses\IGpack>javac socis.java
C:\javaclasses\IGpack>java socis
Error: Could not find or load main class socis
C:\javaclasses\IGpack>
```

Figure 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 figure3.



```
Command Prompt
C:\javaclasses>java IGpack.socis
method1 of socis
C:\javaclasses>
```

Figure 3: Command Prompt screen for successful running of socis file

## 4.5 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[])  
    {  
        socis obj = 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 4: Command Prompt for compiling testpack Java program

- .. After successfully compilation of ‘testpack.java’, a class file ‘socis’ is created under your “javaclasses” folder.

Name	Date modified	Type
socis.class	22-10-2020 18:39	CLASS File
testpack	22-10-2020 18:39	JAVA File

Figure 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
```

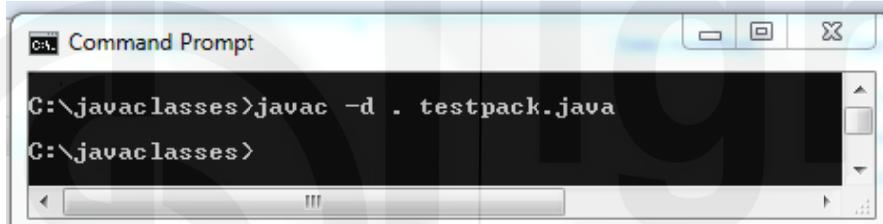


Figure 6: Command Prompt for compiling Java program to create package in current working directory

- .. 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 7 “IGpack1” named package has been created.

Name	Date modified	Type
IGpack1	22-10-2020 19:18	File folder
socis.class	22-10-2020 18:39	CLASS File
testpack	22-10-2020 18:39	JAVA File

Figure 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.

Computer ▶ Local Disk (C:) ▶ javaclasses ▶ IGpack1		
Include in library	Share with	Burn
Name	Date modified	Type
socis.class	22-10-2020 19:18	CLASS File

Figure 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 9.

```
java IGpack1.socis
```

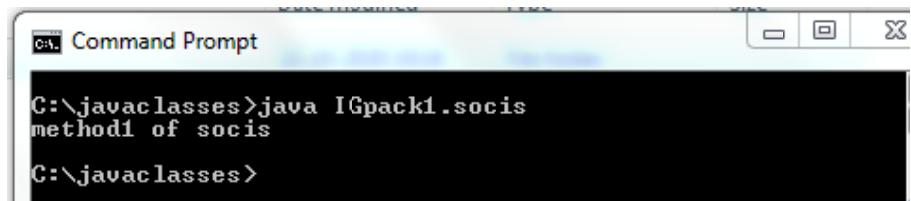


Figure 9: Output Screen for Executing Package in Java

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

- ... You can see the following figure-11, package `IGpack1` is created in C drive. When you click on this folder, it will show “`socis.class`” file.

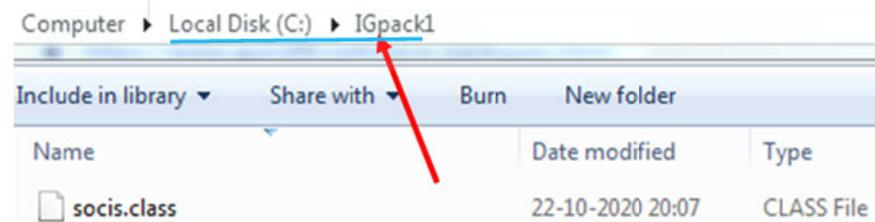


Figure 11: File Structure of creating `socis.class` file in parent directory

## Creation of a sub package within your existing java package

You have seen the 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-12) under the existing package 'IGpack1' and it contains socis.class file as shown in figure-13.

```
C:\ Command Prompt
C:\javaclasses>javac -d . testpack.java
C:\javaclasses>
```

Figure 13: Compiling testpack.java file for creation of sub-package

Computer ▶ Local Disk (C:) ▶ javaclasses ▶ IGpack1 ▶ IGpack2		
Include in library ▼	Share with ▼	Burn
Name	Date modified	Type
socis.class	22-10-2020 22:18	CLASS File

Figure 12: File structure for sub-package IGpack2

### 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-14

```
C:\ Command Prompt
C:\javaclasses>java IGpack1.IGpack2.socis
method1 of socis
C:\javaclasses>
```

Figure 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 15.

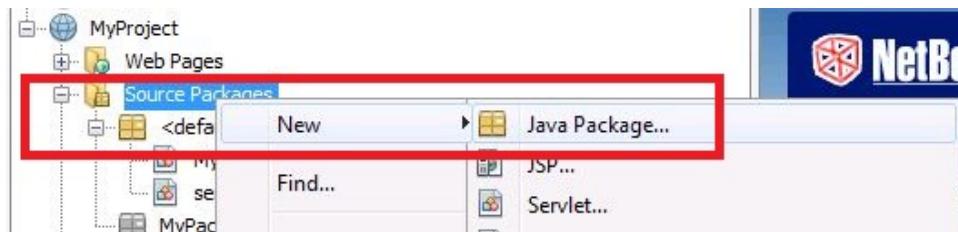


Figure 15: Creation of Package in NetBeans

After selection of the above procedure, you can enter name and location of package. you can see the figure 16.

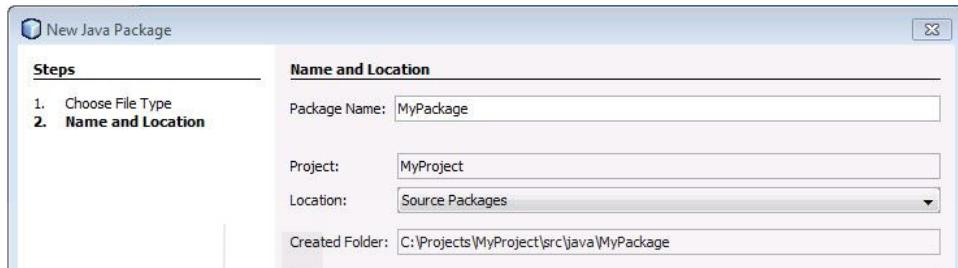


Figure 16: Name and Location of Package

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



Figure 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.

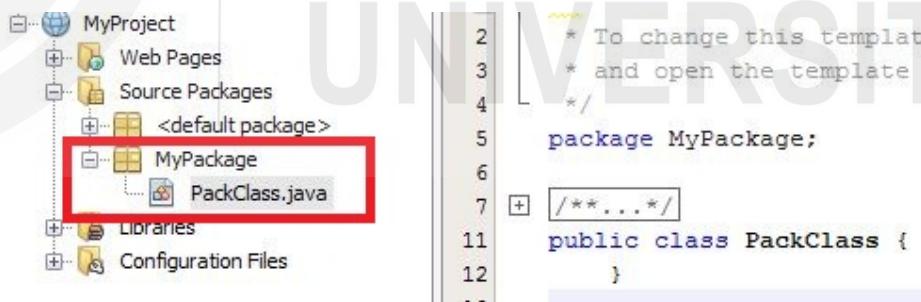


Figure 18:Creation of Java class under Package

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 figure-19:



Figure 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-20 for creation of ‘subclass.java’ file under the ‘newpackage’.

The screenshot shows a file tree on the left and a code editor on the right. The file tree shows a package named 'MyPackage' containing a file 'ParkClass.java'. Inside 'MyPackage', there is a sub-package named 'MyPackage.newpackage' which contains a file 'subclass.java'. The code editor shows the following Java code:

```

5 package MyPackage.newpackage;
6
7 /**
8 * 
9 * @author poonamT
10 */
11 public class subclass {
12 }
13

```

Figure 20: Creation of sub-class under the sub-pakage

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

## 4.6 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 of 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 15.

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

**Step 2:** Now, you can use **javac -d . Addition.java** and it creates Addition.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

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The user has entered the following commands and their results are numbered 1 through 4:

- 1: C:\javaclasses>javac Addition.java
- 2: C:\javaclasses>javac -d . Addition.java
- 3: C:\javaclasses>javac TestPackage.java
- 4: C:\javaclasses>java TestPackage 300

The file structure of the above example is shown in following figure22:

Figure 21: Command Prompt for compiling and executing Java Program

File	Share with	Burn	New folder
Name		Date modified	Type
MyPackage		23-10-2020 19:15	File folder
Addition.class		23-10-2020 19:13	CLASS File
Addition		23-10-2020 19:00	JAVA File
TestPackage.class		23-10-2020 19:24	CLASS File
TestPackage		23-10-2020 19:24	JAVA File

Figure 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 like 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 to compile a source code of Java that is created as a package? Also, write command for executing Java Package program.

---

---

---

4. How packages are imported in java program?

---

---

---

---

## 4.7 BASICS OF INTERFACE IN JAVA

---

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.

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

	<b>Class</b>	<b>Interface</b>
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 extends an interface using *extends clause*. So, you can go thoroughly and run your program correctly.

---

## 4.8 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.

### 4.8.1 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 treats 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 java an 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 Package and select New->Java Interface.

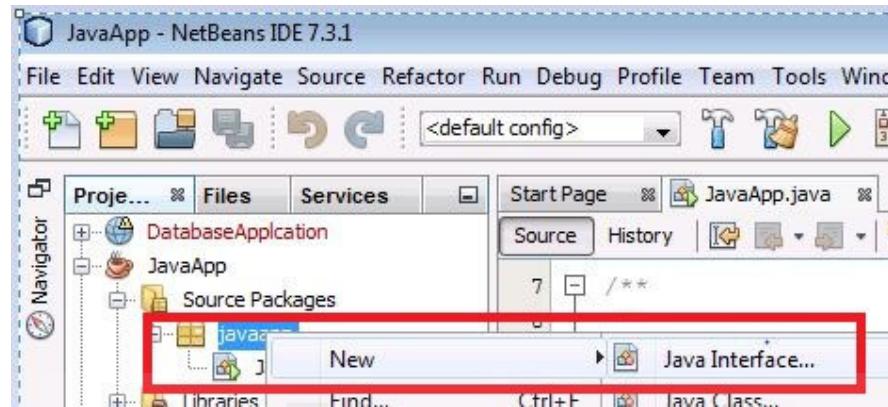


Figure 23:Creation of Interface in NetBeans

Now, Enter name and location of the interface.

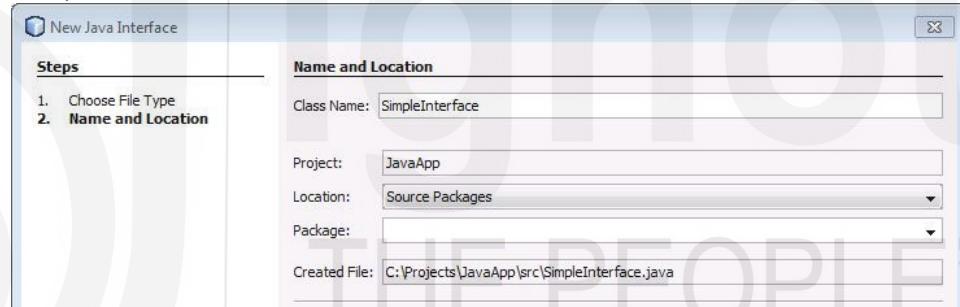


Figure 24:Name and Location of Interface

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

#### 4.8.2 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 .

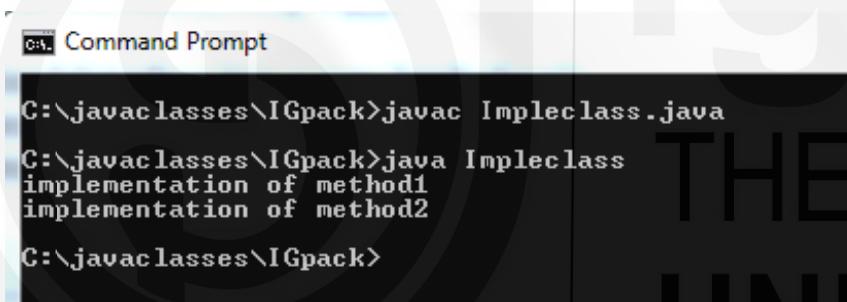
```
interface ImpleInterface
```

```

{
    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 25:



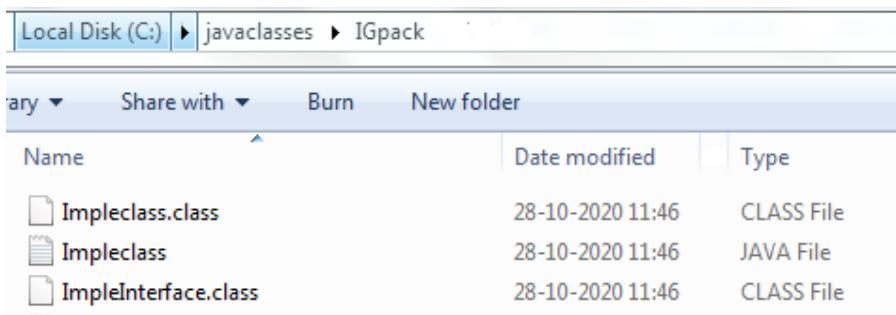
```

C:\Command Prompt
C:\javaclasses\IGpack>javac Impleclass.java
C:\javaclasses\IGpack>java Impleclass
implementation of method1
implementation of method2
C:\javaclasses\IGpack>

```

Figure 25: Command Prompt for compiling and executing of Implementing Interface Example

The file structure of the above program is shown in figure 26:



Local Disk (C:) \ javaclassees \ IGpack		
File	Share with	Burn
Name	Date modified	Type
Impleclass.class	28-10-2020 11:46	CLASS File
Impleclass.java	28-10-2020 11:46	JAVA File
ImpleInterface.class	28-10-2020 11:46	CLASS File

Figure 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-27:

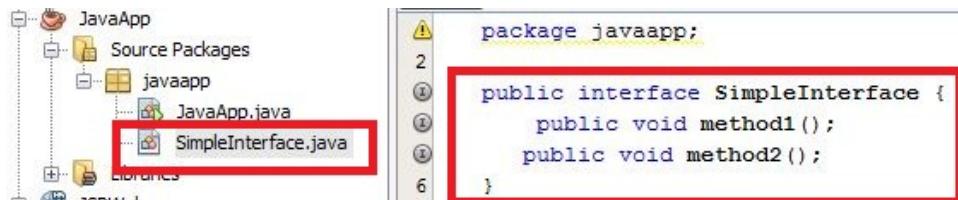


Figure 27: Example of Interface in NetBeans

Now implement the Interface named ‘SimpleInterface’ in Java Class like the following Figure-28:

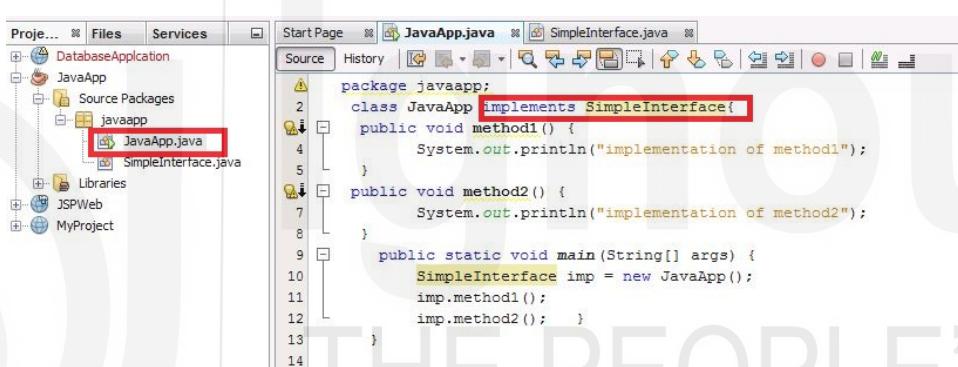


Figure 28: Implementaion of Interface using NetBeans

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-29

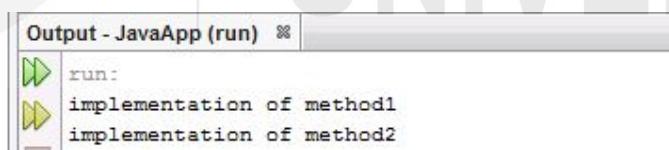


Figure 29: Output of the above Java Program

### 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 **SimpleInterface**, it is assigned an instance of **JavaApp**. 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.

#### 4.8.3 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 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-30:

```
Output - JavaApplication12 (run) ×
run:
Area of Rectangle =200.0
Area of Circle =628.0
BUILD SUCCESSFUL (total time: 1 second)
```

The file structure of the above example is shown in following figure 31:.

Local Disk (C:) ▶ javaclasses ▶ IGpack		
File	Share with	Burn
Name	Date modified	Type
Area.class	26-10-2020 19:12	CLASS File
Circle.class	26-10-2020 19:12	CLASS File
InterfaceTest.class	26-10-2020 19:12	CLASS File
Rectangle.class	26-10-2020 19:12	CLASS File
InterfaceTest	26-10-2020 19:13	JAVA File

Figure 31: File structure of the above example

## 4.9 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[])
    {
        TestClass obj = new TestClass();
        obj.method1();
```

```

        obj. method2();
        obj. method3();
    }
}

```

After compiling and running the above program, it will shows output as shown in following figure-32:

```

Output - JavaApplication12 (run) X
run:
method 1
method 2
method 3
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 32: Output screen for Extending Interface Example

The file structure of the above Extending Interface Example is shown in figure-33.

Local Disk (C:) ▶ javaclasses ▶ IGpack			
Category ▾	Share with ▾	Burn	New folder
Name		Date modified	Type
InterfaceExtend.class		05-11-2020 20:20	CLASS File
InterfaceExtend		05-11-2020 20:20	JAVA File
TestClass.class		05-11-2020 20:20	CLASS File
X1.class		05-11-2020 20:20	CLASS File
X2.class		05-11-2020 20:20	CLASS File

Figure 33: File Structure for Extending Interface Example

## ☛ Check Your Progress-2

- Explain the concept of an interface.

---



---



---



---

- Can you extend interfaces in Java? Explain with example.

---



---



---



---

- 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.

---

---

---

## 4.10 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 4.12, you will find 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 34.

```
C:\ Command Prompt
C:\javaclasses\IGpack>javac example.java
C:\javaclasses\IGpack>java example
This is regular method
This is default method
C:\javaclasses\IGpack>
```

Figure 34: Command Prompt Screen for example of default method in Interface.

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

Local Disk (C:) ▶ javaclasses ▶ IGpack ▶			
Name	Date modified	...	Type
defaultInterface.class	29-10-2020 12:33		CLASS File
example.class	29-10-2020 12:33		CLASS File
example	29-10-2020 12:33		JAVA File

Figure 35: File structure of the above example

## 4.11 ISSUES OF MULTIPLE INTERFACES

You have been gone through about the inheritance in Unit 1 Block 2 of this course. You already know that **Multiple Inheritance** is a feature of object oriented programming where a class can inherit 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-36:

```
C:\javaclasses\IGpack>javac Demo.java
Demo.java:20: error: ',' expected
class Demo extends P1, P2
                                ^
1 error
```

Figure 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");
    }
}

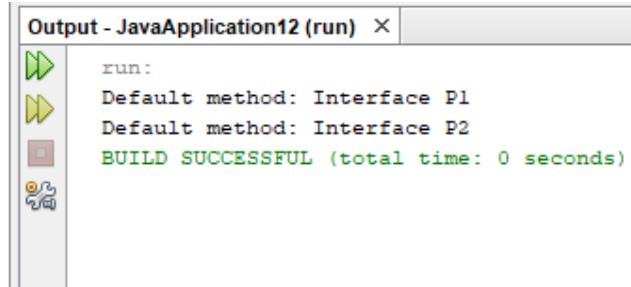
// Implementation class
class TestMulti implements P1, P2
{
    // Overriding default display() method
    public void display()
    {
        /* use super keyword to call the display method of P1 interface */
        P1.super.display();

        /* 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 37



The screenshot shows the 'Output - JavaApplication12 (run)' window. It contains a toolbar with icons for run, stop, and others. The main area displays the command 'run:' followed by the output of the program: 'Default method: Interface P1' and 'Default method: Interface P2'. Below this, a green message says 'BUILD SUCCESSFUL (total time: 0 seconds)'.

Figure 37: Command Prompt Screen for test multiple inheritance through interfaces program example

---

## 4.12 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 and 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[])
    {
        staticexample obj = 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 38:

```

Output - JavaApplication12 (run) ×
run:
This is regular method
The default method is define in interface
This is Static Method
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 38: Output Screen for use of Static method in Interface

The file structure of the above java program is looks like the following figure-39:

Local Disk (C:) > javaclasses > IGpack			
Category	Name	Date modified	Type
	staticexample.class	30-10-2020 11:22	CLASS File
	staticInterface.class	30-10-2020 11:22	CLASS File
	staticexample.java	30-10-2020 11:22	JAVA File

Figure 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.

---

---

---

---

### 4.13 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 makes 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 is 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.

## 4.14 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 4.9 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();

}
```

---

## 4.15 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, “ A programer’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

THE PEOPLE'S  
UNIVERSITY

---

# **UNIT 1 MULTITHREADED PROGRAMMING**

---

## **Structure**

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Multithreading
- 1.3 Java thread Model
- 1.4 Creating Threads in Java
  - 1.4.1 The Thread Class
  - 1.4.2 The Main Thread
  - 1.4.3 Creating Child Threads
- 1.5 Thread Life Cycle
- 1.6 Creating Multiple Threads
- 1.7 Using `isAlive()` and `join()`
- 1.8 Thread Priority
- 1.9 Synchronization
- 1.10 Interthread Communication
- 1.11 Suspending, Resuming, and Stopping Threads
- 1.12 Obtaining a Thread State
- 1.13 Using Multithreading in problem solving
- 1.14 Summary
- 1.15 Solutions/ Answer to Check Your Progress
- 1.16 References/Further Reading

---

## **1.0 INTRODUCTION**

---

Multithreading is one of the important features provided by java programming language. A thread represents a single sequence of execution that can independently execute in an application. Uses of threads in programs are good for maximizing the resources utilization of the system on which applications are running. Multithreaded programming is very useful in developing many types of applications, such as network applications, gaming applications and Internet applications. In this unit, you will learn the concept of multithreading and how they are used in applications development using java. This unit also explains the working of threads, thread properties, thread synchronization, and interthread communication.

---

## **1.1 OBJECTIVES**

---

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

- ... understand concepts of multithreading,
- describe the java thread model,
- create and use threads in your programs,
- write programs to describe how to set the thread priorities,
- use the concept of thread synchronization in programming, and
- use inter-thread communication in programs.

---

## **1.2 MULTITHREADING**

---

Most modern computer systems have the capability of performing more than one job. It is like having more than one computer to perform your jobs. Conceptually, in basic terms, this is called multitasking.

Multitasking can be performed either at the Process level, or it can be executed at Thread level.

- ... Process based multitasking: A process is a program in execution. This type of multitasking lets your computer run two or more programs simultaneously.

### Multiprocessing

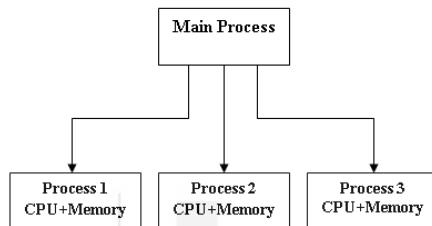


Figure 1(a): Multiprocessing

- ... Thread based multitasking: Thread is a part of a single program that can run concurrently. In thread-based multitasking, one program can perform two or more tasks simultaneously.

### Multithreading

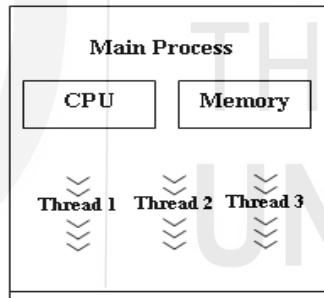


Figure 1(b): Multithreading

### Parts of a Program

Any computer program in memory has four different types of spaces. These are stack, heap, variable- space and program code space. These are allocated by the memory management module of the operating system based on the optimization algorithm used by the operating system.

- ... The stack is used for static memory allocation.
- ... The heap is used for dynamic memory allocation.
- ... The variable space is used for storing all the variables declared.
- ... The code space includes all the instructions written in the code.

## Heavy-weight and Light-weight Processes

### Multithreaded Programming

A program will occupy all four spaces as a unit. In process-based multitasking, if another program resides in the memory, it will occupy all four spaces as a separate unit. That is why process-based multitasking is known as heavy-weight multitasking. On the other hand, a thread is a part of a process. All threads of a process use the same four spaces that the program is allocated by overlaying the space with their variables and code space. Since they occupy the same space, they are called light-weight processes, as can be seen in figure 2.

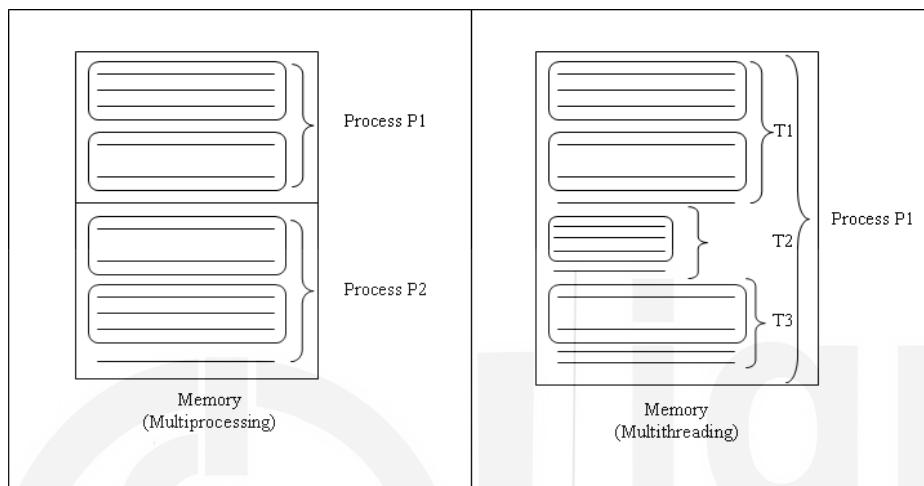


Figure 2: Multiprocessing and Multithreading

As apparent from the figure 2, process-based multitasking uses individual memory spaces, whereas the threads use the single memory space of a process, justifying the name light-weight processes.

#### ☛ Check your Progress-1

- 1) What are different memory spaces occupied by a program when it is in execution?

.....  
.....  
.....

- 2) Why is a thread called a light-weight process?

.....  
.....  
.....

- 3) What are the advantages of multithreading?

.....  
.....

## 1.3 JAVA THREAD MODEL

A thread has the following three components:

- ... CPU –Thread Class is designed to encapsulate the virtual CPU in it. When a thread is constructed, the code and data that define the thread specified by the object are passed to the constructor of the thread class.
- ... Data – Data may/may-not be shared by multiple threads
- ... Code – Shared by multiple threads, independent of data.

Java is a very strong language as far as support of multithreading is concerned. It is having a very rich set of methods available for multithreaded programming. Further in this unit, you will learn how to write program using concepts of multithreading in java.

In java, a thread can be in various states. These states are identified as :

- ... New state: The main thread starts when the program starts executing.
- ... Ready State: It is ready to occupy CPU as soon as it is possible by the scheduler.
- ... Running State: The Thread is in running state when it has the CPU. It may leave this state for doing I/O or by executing wait(), sleep() methods, etc.
- ... Blocked State: A thread moves to blocked state for several reasons. Few notable ones are performing I/O, acquiring locks, resources etc.
- ... Dead State: A thread moves to dead state when it completes its execution, i.e., the execution of main() method completes.

A thread life cycle can be depicted by combining the thread states, as shown in figure 3.

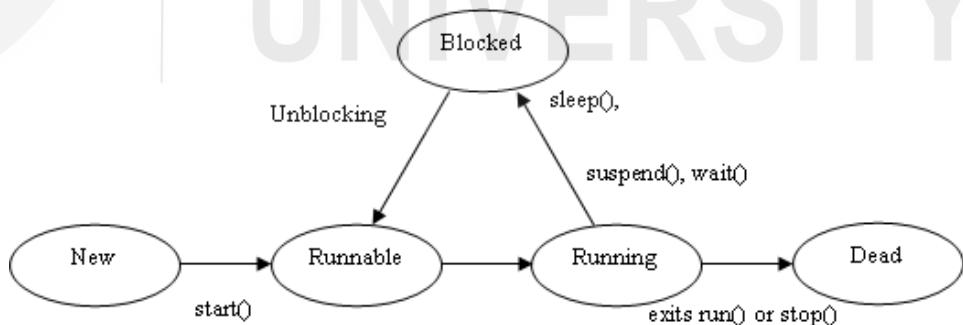


Fig 3: Thread Life Cycle

## 1.4 CREATING THREADS IN JAVA

There are two ways to create a thread in java. The java Multithreading system is built upon the following:

- ... Thread Class and its methods
- ... Interface Runnable

The **java.lang.Thread** class allows you to create and control child thread. The Thread class encapsulates the thread of CPU; The thread reference thus obtained helps to know the status of the main thread.

### 1.4.1 The Thread Class

The Thread class defines several constructors and methods to create and manage threads

#### The Thread Constructors

Few Thread class constructors are listed below:

Constructor	Description
<b>Thread()</b>	This allocates a new <b>Thread</b> object.
<b>Thread(String name)</b>	This constructor takes the name of the child-thread as the parameter and constructs the Thread object-instance
<b>Thread(Runnable target)</b>	This allocates a new <b>Thread</b> object
<b>Thread(Runnable target, String name)</b>	This allocates a new <b>Thread</b> object.

There are several other constructors with ThreadGroup Parameter to create thread objects, but it is beyond the scope of our discussion in this course.

#### The Thread Class Methods

Thread Class defines several methods to manage the threads. Some commonly used methods of Thread class are listed below.

Method	Description
<b>getName()</b>	Gets the name of the thread
<b>getPriority()</b>	Gets the priority of the thread
<b>isAlive()</b>	Determining if the Thread is still Alive
<b>join()</b>	Waits for a thread to terminate
<b>run()</b>	Entry point for thread
<b>sleep()</b>	Suspends a thread for a period
<b>start()</b>	Start a thread by calling its run method.

### 1.4.2 The Main Thread

Whenever a java program runs, one thread begins running immediately. This thread is called the main thread of the program. All child threads are born out of the main thread. Also, all the child threads will finish their execution before the main thread. The main thread is always the last thread to complete its execution. When the main thread finishes, the program terminates.

The main thread is created automatically by the system when your program starts running. It could be controlled by the thread reference. The reference is obtained by the `currentThread()` method, which is a public, static function of the `Thread` Class.

Now let us see a program that demonstrates the working of the main thread.

```
import java.lang.Thread;
public class Currthread
{
    public static void main(String[] args)
    {
        Thread t=Thread.currentThread();
        System.out.println("Name of the current thread is:"+t.getName());
        t.setName("Parent Thread");
        System.out.println("New Name of the current thread is:"+t.getName());
        System.out.println("Is thread alive?:"+t.isAlive());
        System.out.println("Priority of the current thread:"+t.getPriority());
        t.setPriority(8);
        System.out.println("New Priority of the current thread:"+t.getPriority());
    }
}
```

```
run:
Name of the current thread is:main
New Name of the current thread is:Parent Thread
Is thread alive?:true
Priority of the current thread:5
New Priority of the current thread:8
BUILD SUCCESSFUL (total time: 0 seconds)
```

In the above program, you can see the use of methods `setName()`, which is used to set the name of that on which it is called. Also, methods `setPriority()` and `getPriority()` are used in this program. For thread priority, we will learn in the later section of the unit.

### 1.4.3 CREATING CHILD THREADS

As mentioned earlier , there are two ways to create thread instances.

- ... By implementing the `Runnable` interface
- ... By extending the `Thread` class

Now let us see these ways of creating threads one by one.

#### 1.4.3.1 First Way to create thread by implementing Runnable interface

To implement `Runnable` interface, a class needs to implement a single method called `run()`.

```
public void run()
{
```

.....

}

Inside run() method you can define the code that you want from your child thread to execute. The run() method can call other methods, use other classes, declare variables, just like the main thread. Basically the run() method establishes the entry point for another concurrent thread of execution within your program.

After you create a class that implements Runnable interface, you will instantiate an object of Thread from that class. An example of creating a heading by implementing the Runnable interface is given below.

```
import java.lang.Thread;
class mythread implements Runnable
{
    Thread t;
    String name;
    static int a, b;
    mythread(String n,int a1,int b1)
    {
        a=a1;
        b=b1;
        t=new Thread(this,n);//Child Thread created
        t.start();//Child thread now ready to run
    }
    public void run()
    {
        int c;
        c=a-b;
        System.out.println("child thread :");
        System.out.println("subtraction is :" +c);
    }
}
public class Childthread
{
    public static void main(String[] args)
    {
        int a=9;int b=5;
        mythread t1=new mythread("child thread",a, b);
        System.out.println("addition is :" +(a+b));
    }
}
```

### Output:

```
Output - childthread (run)
run:
addition is :14
child thread :
subtraction is :4
BUILD SUCCESSFUL (total time: 0 seconds)
```

#### 1.4.3.2 Second way to create child threads is to extend the Thread class itself.

The Thread class defines several constructors.

The constructor **Thread(String name)** uses the child-thread name as the parameter. Using the constructor, the new thread is created. The child thread created does not run till a call to start() method is called.

The start() method makes a call to run() method ( This is done automatically). Once the call to run() is made, the child thread is ready to run and is put in the scheduling queue.

```
//A program in java to extend Thread class.
public class myThread extends Thread
{
    static int a,b;
    String n;
    myThread(String n1, int a1, int b1)
    {
        super(n1);
        a=a1;
        b=b1;
        start();
    }
    public void run()
    {
        System.out.println("SUM is " + (a+b));
    }
    public static void main(String args[])
    {
        myThread i1 = new myThread("Thread is ",10,5);
        System.out.println("Main Thread " + (a-b));
    }
}
```

#### Output

```
Output
run:
Main Thread 5
SUM is 15
BUILD SUCCESSFUL (total time: 0 seconds)
```

## ☛ Check your Progress-2

- 1) Discuss the difference between Runnable and the Running state of the thread.

.....  
.....  
.....  
.....

- 2) Create a child thread by implementing the Runnable interface wherein the child thread does string concatenation, and the main thread changes the string to uppercase.

.....  
.....  
.....  
.....

- 3) Repeat the above program by making child thread by extending the thread class.

.....  
.....  
.....

---

## 1.5 CHILD THREAD LIFE CYCLE

A newly born/child thread is created when the constructor is called. The child thread starts running only when a call to start() method is done. Calling start() method places the child thread in a runnable state. This means, it is ready to execute the run() method and is available for scheduling. This does not mean that the thread runs immediately. It executes the run method only after the CPU is available .

- ... Ready state : After the call to start() method is made, the thread is in ready state. It is ready to occupy CPU as soon as it is possible by the scheduler.
- ... Running State: The Thread is in running state when it has the CPU and is executing its run() method. It may leave this state for doing I/O or by executing wait(),sleep() methods, etc.
- ... Blocked State : A thread moves to blocked state for several reasons. Few notable ones are performing I/O, acquiring locks, resources etc.
- ... Dead State : A thread moves to dead state when it completes its execution, i.e., the execution of run() method.

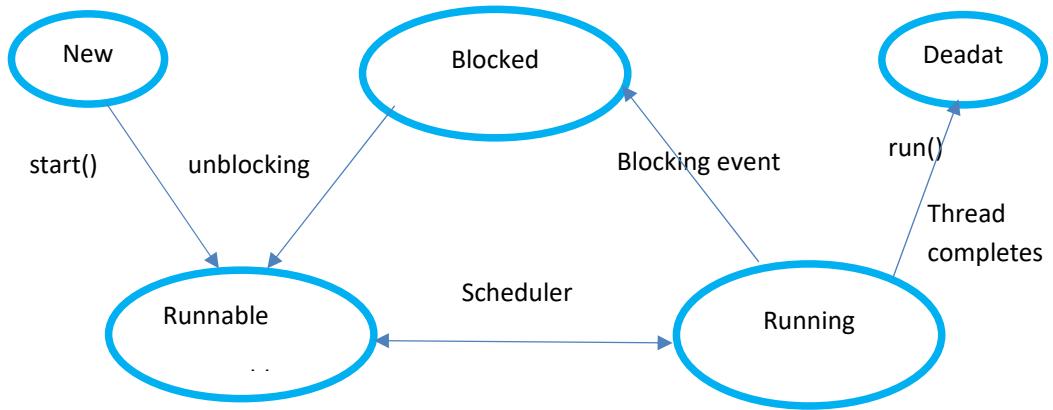


Figure 4:Life cycle of Child Thread

Java threads follow the **preemptive scheduling** where a thread with lower priority relinquishes the CPU when preempted by a thread of higher priority. The model of a preemptive scheduler is such that many threads might be runnable, but only one is running.

A thread can relinquish the CPU for a variety of other reasons besides being preempted by a higher priority thread:

- ... The thread code can execute a `Thread.sleep()` method call deliberately asking the thread to pause for a fixed period.
- ... The thread might have to perform I/O, access a shared resource, access a network resource, and cannot continue until the resource becomes visible.

All threads that are runnable are kept in a pool according to their priority. When CPU becomes available, the thread with the highest priority is given the CPU.

Since java threads are not time-sliced therefore you must ensure that the code for your threads gives other threads a chance to execute from time to time. This is achieved by calling the sleep call at various intervals or by code controlled programming.

`Thread.sleep()` method can pause a thread for a specific period and are interruptible. The `sleep()` is a static method in the `Thread` class. Because it operates on the current thread, it is called `Thread.sleep( int milliseconds )` where `milliseconds` is the time for which the thread is made inactive.

When a child thread completes the `run()` method's execution, it terminates and then cannot run again.

## 1.6 CREATING MULTIPLE CHILD THREADS

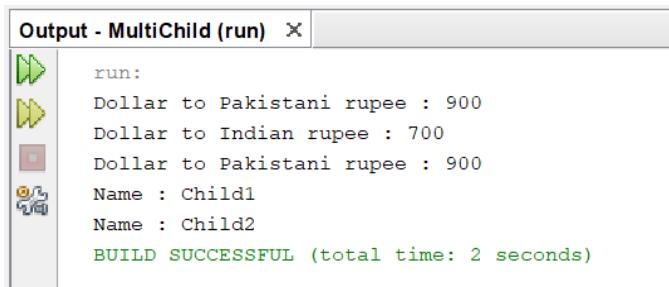
More than one child thread can be created and can be programmed to perform different tasks. In the following example, the main thread converts the dollar to the Indian rupees. The child thread1 converts the dollar to the Pakistani rupee, and childthread2 converts to the Nepali rupee.

Note: The function `getName()` is used to distinguish between the child threads.

```
import java.lang.Thread;
```

```
public class MultiChild implements Runnable
{
    String name;
    static int dollar,rupee;
    Thread t;
    MultiChild(String n,int d)
    {
        t=new Thread(this,n);
        dollar=d;
        rupee=0;
        t.start();
    }
    public void run()
    {
        if(t.getName().equals(name))
        {
            System.out.println("Name child1: "+t.getName());
            rupee = dollar*80;
            System.out.println("Dollar to Nepali rupee : "+ rupee);
        }
        else
        {
            System.out.println("Dollar to Pakistani rupee : "+ dollar*90);
            System.out.println("Name : "+t.getName());
        }
    }
    public static void main(String args[])
    {
        MultiChild t1 = new MultiChild("Child1",10);
        MultiChild t2 = new MultiChild("Child2",10);
        System.out.println("Dollar to Indian rupee : "+ dollar*70);
    }
}
```

### Output:



```
Output - MultiChild (run) ×
run:
Dollar to Pakistani rupee : 900
Dollar to Indian rupee : 700
Dollar to Pakistani rupee : 900
Name : Child1
Name : Child2
BUILD SUCCESSFUL (total time: 2 seconds)
```

## 1.7 USING ISALIVE() AND JOIN()

There are some methods that help to control the working of a thread.

**Testing a Thread State:** A thread can be in an unknown state. Using the method *isAlive()*, you can determine if a thread is still viable(Running/Runnable/Blocked). This method returns true for the thread that has started but has not completed its task(i.e still running).

```
public final boolean isAlive()
```

This method returns true if this thread is alive; false otherwise. The following java program shows the use of *isAlive()* method.

```
//Program
```

```
Class RunnableClass implements Runnable
{
    public void run()
    {
        for(int i = 0; i < 3 ; i++)
        {
            System.out.println(Thread.currentThread().getName() + " i - " + i);
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                System.out. println(" Exception Caught");
                e.printStackTrace();
            }
        }
    }
}
public class Example
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new RunnableClass(), "t1");
        Thread t2 = new Thread(new RunnableClass(), "t2");
        t1.start();
        t2.start();

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());
        System.out.println("Processing finished");
    }
}
```

### Output:

```
t1 Alive - true
t2 Alive - true
Processing finished
t3 i - 0
t1 i - 0
```

```
t2 i - 0
t3 i - 1
t1 i - 1
t2 i - 1
t2 i - 2
t1 i - 2
t3 i - 2
```

**Joining a Thread:** The `join()` method causes the calling thread to wait until the thread on which the `join` method is called terminates.

```
public final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. There are three overloaded `join` functions.

- ... **public final void join()** - Waits indefinitely for this thread to die.
- ... **public final void join(long milliseconds)** - Waits at most milliseconds for this thread to die. A timeout of 0 means to wait forever.
- ... **Public final void join(long milliseconds, int nanoseconds)** - Waits at most milliseconds plus nanoseconds for this thread to die.

The following is a java program that shows the use of the `join()` method.

```
//program
Class RunnableClass implements Runnable
{
    public void run()
    {
        for(int i = 0; i < 5 ; i++)
        {
            System.out.println(Thread.currentThread().getName() + " i - " + i);
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
public class JoinExample
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new RunnableClass(), "t1");
        Thread t2 = new Thread(new RunnableClass(), "t2");
        t1.start();
        t2.start();

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        try
        {
```

```

        t1.join();
        t2.join();

    }

    catch (InterruptedException e)
    {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    System.out.println("t1 Alive - " + t1.isAlive());
    System.out.println("t2 Alive - " + t2.isAlive());

    System.out.println("Processing finished");
}
}

```

**Output:**

```

t1 Alive - true
t2 Alive - true
t2 i - 0
t1 i - 0
t1 i - 1
t2 i - 1
t1 i - 2
t2 i - 2
t1 Alive - false
t2 Alive - false
Processing finished

```

## 1.8 JAVA PRIORITY ENVIRONMENT

Java threads operate in a Preemptive/priority- based scheduling environment. This means that a thread with higher priority forces/preempts a thread with lower priority to release the CPU. This is called context switching. Two threads of equal priority are further scheduled by the secondary scheduling algorithm of the operating system. The default priority assigned to the thread is 5(NORM-PRIORITY). The range lies between MIN-PRIORITY( value=1) and MAX-PRIORITY( value =10). The priority of a thread is an int value . The priority can be set using the method:

*final void setPriority(int level)*

where the level specifies the new priority setting for the calling method. The value of the level must be in range 1 to10.

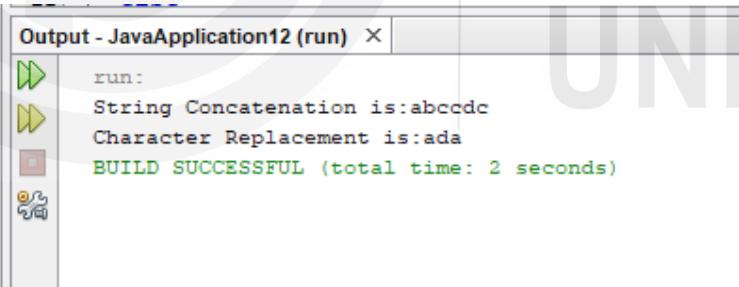
Following program demonstrate the use of `setPriority( int level)` method.

```

//Program
import java.lang.String;
public class Priority implements Runnable

```

```
{  
    String name;  
    Thread t;  
    String s3, s4;  
    Priority(String n, String s1, String s2, int prio)  
    {  
        name = n;  
        s3 = new String(s1);  
        s4 = new String(s2);  
        t = new Thread(this, n);  
        t.setPriority(prio);  
        t.start();  
    }  
  
    public void run()  
    {  
        if(t.getName().equals("ch1"))  
            System.out.println("String Concatenation is:" + (s3+s4));  
        else  
            System.out.println("Character Replacement is:" + s4.replace('c','a'));  
    }  
}  
public static void main(String args[])  
{  
    String s1 = new String("abc");  
    String s2 = new String("cdc");  
    Priority th1 = new Priority("ch1", s1, s2, 7);  
    Priority th2 = new Priority("ch2", s1, s2, 2);  
}  
}
```

**Output:**

```
Output - JavaApplication12 (run) ×  
run:  
String Concatenation is:abccdc  
Character Replacement is:ada  
BUILD SUCCESSFUL (total time: 2 seconds)
```

**☛ Check your Progress-3**

- 1) Enlist the ways in which a child thread is created. Which one is preferred method and why?

.....  
.....  
.....  
.....

- 2) Discuss the macros for priorities within Java thread environment.

- 3) Discuss the utility of isAlive() method in the Java thread environment

## 1.9 SYNCHRONIZATION OF THREADS

When two or more threads want access to the shared resources, they need some way to ensure that the resource is used by only one thread that has exclusive access to it at a time. The synchronization keyword helps to achieve this. Synchronization in java is achieved by using the concept of object lock flag.

The communication with the lock flag is achieved via the keyword I and allows exclusive access to code with shared data. When the thread reaches the synchronized statement, it examines the object passed as an argument. The object is examined to obtain the lock flag from it before continuing. After taking the lock flag from the object, the thread continues executing the shared code(if not, the thread waits in the object lock flag pool). When all other threads try to execute the same synchronized statement, they also try to obtain the lock flag from the object, which is not present. These threads then join a pool of waiting threads on object's lock pool. When the previous thread finishes executing the synchronized method, the lock flag is returned and then a thread waiting on lock pool is given the flag, and then this thread starts executing the synchronized statement.

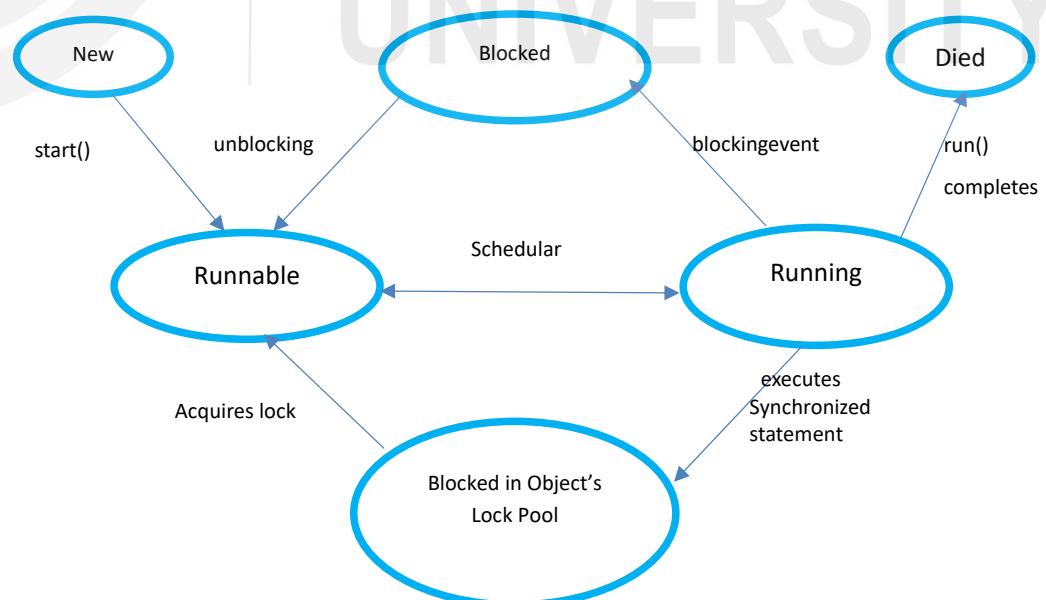
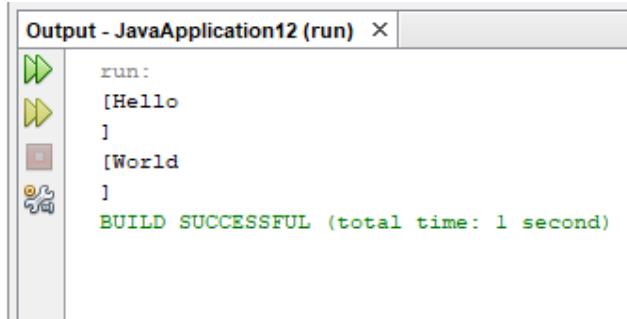


Fig 5: States of Thread through its life cycle with synchronized statement

The following java program illustrates the concept of synchronization.

```
//program
class Callme
{
    void call(String msg)
    {
        System.out.println("[ " + msg);
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("] ");
    }
}
class Caller implements Runnable
{
    Callme target;
    String msg;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run()
    {
        synchronized(target)
        {
            target.call(msg);
        }
    }
}
class Synch
{
    public static void main(String[] agrs)
    {
        Callme target = new Callme();
        Caller obj1 = new Caller(target, "Hello");
        Caller obj2 = new Caller(target, "World");
    }
}
```

**Output:**



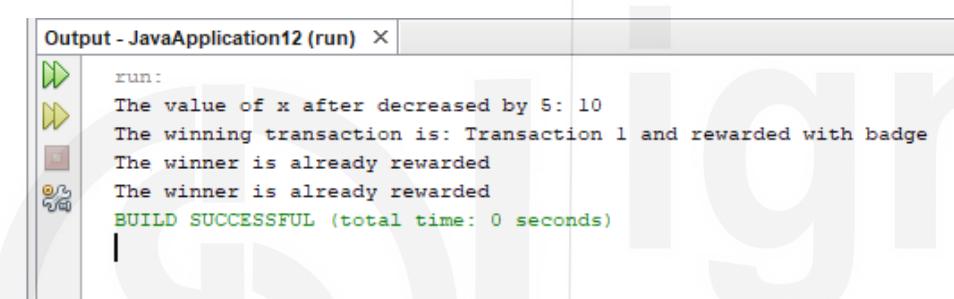
```
Output - JavaApplication12 (run) ×
run:
[Hello
]
[World
]
BUILD SUCCESSFUL (total time: 1 second)
```

Let us see one more java program in which each transaction decrement the value of given object say x by -5. Here it is assumed 3 transactions t1,t2 and t3 and 3 child threads. They starts simultaneously with the same priority, and each one tries to decrements the value of x. The system tries to return the transaction, which completes its execution first.

```
//Program
class Decrement
{
    public void decrementValue(int x)
    {
        System.out.println("The value of x after decreased by 5: "+(x-5));
    }
}
class Decrementar implements Runnable
{
    static boolean winner=false;
    String name;
    int x;
    Thread t;
    Decrement d;
    Decrementar(Decrement d1,String n,int num)
    {
        this.d=d1;
        name=n;
        x=num;
        t=new Thread(this,name);
        t.start();
    }
    public void run()
    {
        synchronized(d)
        {
            if(!winner)
            {
                d.decrementValue(x);
                System.out.println("The winning transaction is: "+name+" and rewarded with badge");
                winner=true;
            }
            else
            {
                System.out.println("The winner is already rewarded");
            }
        }
    }
}
```

```
        }
    }
}
public class decrementTransaction
{
    public static void main(String args[])
    {
        Decrement d=new Decrement();
        new Decrementar(d,"Transaction 1",15);
        new Decrementar(d,"Transaction 2",11);
        new Decrementar(d,"Transaction 3",19);
    }
}
```

### Output:



The screenshot shows the 'Output - JavaApplication12 (run)' window in an IDE. It displays the following text:  
run:  
The value of x after decreased by 5: 10  
The winning transaction is: Transaction 1 and rewarded with badge  
The winner is already rewarded  
The winner is already rewarded  
BUILD SUCCESSFUL (total time: 0 seconds)

## 1.10 INTERTHREAD COMMUNICATION

Three final methods namely `wait()`, `notify()`, `notifyAll()` are provided by `java.lang.Object` class for inter-thread communication. When a `wait` call is issued on object X by a thread, it pauses its execution until another thread issues a `notify` call on the same object X. The prerequisite for a thread to call `wait/notify` is that the thread must possess lock flag for that particular object. Hence, it is understood that `wait` and `notify` can only be called from within a synchronized block of the object.

When a thread executes a synchronized code on a particular object and calls a `wait()`, the thread is placed in wait-pool for that object. Additionally, the thread that calls `wait` releases that object's lock flag and goes to sleep until some other thread enters the same synchronized code on the same object and calls `notify()`. When a `notify` call is executed on a particular object, the first thread that calls `wait` on the same object is moved from the object's wait pool to the object's lock pool where threads stay until object's lock flag becomes available.

On the other hand, when `notifyAll()` call is issued, it moves all thread waiting on that object's wait-pool and puts them into the lock-pool. Only from the lock pool, when the highest priority thread obtains the lock flag, can continue running, where it left off when it called to `wait`.

Method	Description
public final void wait() throws InterruptedException	waits until object is notified
public final void wait (long timeout) throws InterruptedException	waits for the specified amount of time
public final void notify()	Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
public final void notifyAll()	Wakes up all threads that are waiting on this object's monitor.

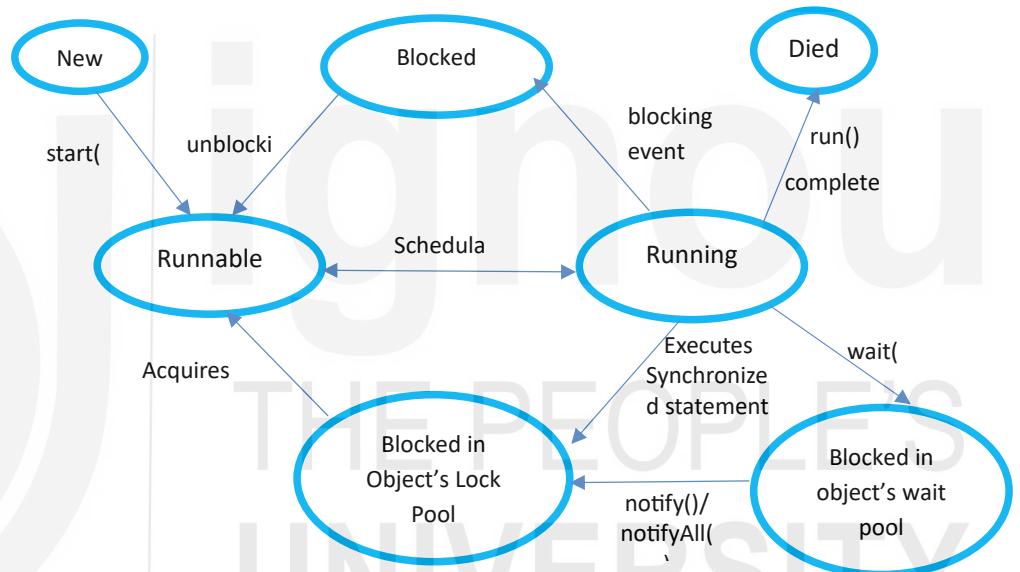


Fig 6: States of Thread through its life cycle with synchronized statement with wait/notify

The following java program shows the concept of interthread communication using the producer-consumer problem for one unit of food. You are advised to execute this program and watch the output.

```

//Program
class Q
{
    int num;
    boolean valueSet = false;

    synchronized int get()
    {
        while(!valueSet)
        try
        {
            wait();
        }
    }
}
  
```

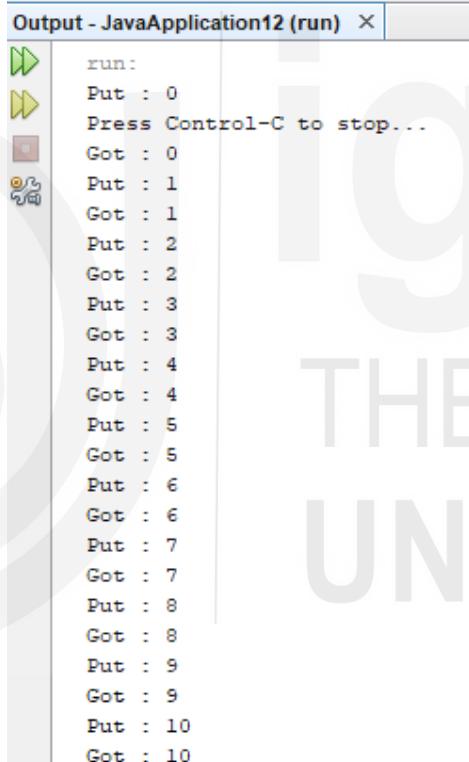
```
catch(InterruptedException e)
{
    System.out.println("InterruptedException caught..!!!");
}
System.out.println("Got : " + num);
valueSet = false;
notify();
return num;
}
synchronized void put(int num)
{
    while(valueSet)
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
            System.out.println("InterruptedException caught..!!!");
        }
    this.num = num;
    valueSet = true;
    System.out.println("Put : " + num);
    notify();
}
class Producer implements Runnable
{
    Q que;
    Producer(Q que)
    {
        this.que = que;
        new Thread(this, "Producer").start();
    }
    public void run()
    {
        int n = 0;
        while(true)
        {
            que.put(n++);
        }
    }
}
class Consumer implements Runnable
{
    Q que;
    Consumer(Q que)
    {
        this.que = que;
        new Thread(this, "Consumer").start();
    }
    public void run()
    {
        while(true)
        {
```

ignou  
THE PEOPLE'S  
UNIVERSITY

```
        que.get();
    }
}
}

class PCFixed
{
    public static void main(String args[])
    {
        Q que = new Q();
        new Producer(que);
        new Consumer(que);
        System.out.println("Press Control-C to stop...");
    }
}
```

Output:



```
run:  
Put : 0  
Press Control-C to stop...  
Got : 0  
Put : 1  
Got : 1  
Put : 2  
Got : 2  
Put : 3  
Got : 3  
Put : 4  
Got : 4  
Put : 5  
Got : 5  
Put : 6  
Got : 6  
Put : 7  
Got : 7  
Put : 8  
Got : 8  
Put : 9  
Got : 9  
Put : 10  
Got : 10
```

---

## 1.11 SUSPENDING, RESUMING AND STOPPING THREAD

---

These are deprecated methods by Java 2.0. While the suspend(), resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads to pause, restart and stop the execution of the thread.

**suspend()**

The suspend( ) method of the Thread class was deprecated by Java 2 several years ago. This was done because suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that the thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

**resume()**

The resume( ) method is also deprecated. It does not cause problems but cannot be used without the suspend( ) method as its counterpart.

**stop()**

This method of the Thread class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

## 1.12 OBTAINING A THREAD STATE

As discussed earlier, a thread can exist in several states. You can get the current state of the thread by calling the getState() method defined by the Thread class.

`Thread.State getState()`

It returns a value of type Thread. State indicates the state of thread at the time at which the call was made. State is an enumeration defined by Thread.

The following table enlists the values that can be returned by getState().

State-Value	Meaning
BLOCKED	A thread has suspended execution because it is waiting to acquire a lock
NEW	A thread is created and has not begun execution
RUNNABLE	A thread that is currently executing and has access to cpu
TERMINATED	A thread that has completed execution
TIMED-WAITING	A thread that has suspended execution for a specified period because it is waiting for some action to occur
WAITING	A thread that has suspended execution because it is waiting for some action to occur ...e.g. wait() and join()

## 1.13 USING MULTITHREADING IN PROBLEM SOLVING

Before we conclude this unit, let us see a program that book a seat using synchronized statement.

```
// java program for a seat reservation in railway
```

```
//package railway;
class reservation
{
    public void reserve(String s)
    {
        System.out.println("seat is reserved by "+s);
    }
}
class customer implements Runnable
{
    reservation r;
    String name;
    Thread t;
    static boolean seat=true;
    public
    customer(reservation r1,String s)
    {
        r=r1;
        name=s;
        t=new Thread(this,s);
        t.start();
    }
    public void run()
    {
        synchronized(r)
        {
            if(seat)
            {
                r.reserve(name);
                seat=false;
            }
            else
            {
                System.out.println(name+" has to wait for seat");
            }
        }
    }
}
public class Railway1
{
    public static void main(String[] args)
    {
        reservation r11=new reservation();
        customer c1=new customer(r11,"customer1");
        customer c2=new customer(r11,"customer2");
    }
}
```

**Output:**

```
Output - JavaApplication12 (run) X
run:
seat is reserved by customer1
customer2 has to wait for seat
BUILD SUCCESSFUL (total time: 0 seconds)
```

#### ☛ Check your Progress-4

- 1) Discuss the concept of synchronization.

.....  
.....  
.....  
.....

- 2) Explain the utility of inter-thread communication.

.....  
.....  
.....  
.....

- 3) Discuss any two problem scenarios where threads can be utilized.

.....  
.....  
.....  
.....

---

## 1.14 SUMMARY

---

This unit described the concepts of multithreading in Java programming. The unit begins with the concept of the main thread, its creation and control methods. Different states of threads are discussed in detail, from its creation to the death state. This unit also explained the process of creating child threads using Thread class and Runnable interface. Subsequently the concept of preemptive java environment with thread's priority are explained. This unit demonstrated the use of the concept of synchronization, creating synchronous methods and inter-thread communication. The concept of object locks used to control access to shared resources also has been explained.

---

## 1.15 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

---

### **➤ Check your Progress-1**

- 1) A program in execution is assigned memory by the operating system. The memory assigned is divided into the following spaces: stack, heap, variable space, and code space
  - ... The stack is used for static memory allocation.
  - ... The heap is used for dynamic memory allocation.
  - ... The variable space is used for storing all the variables declared.
  - ... The code space includes all the instructions written in code.
- 2) The thread process is a light-weight process as a child process continues to share the memory space of the parent thread. No separate memory space is allocated, unlike in fork, where a child process gets a replica of all in a separate memory space.
- 3) Advantages of Multithreading are they being light weight process. They perform multiple tasks while operating in the same memory space.

### **➤ Check your Progress 2**

- 1) A thread is said to be in a runnable state when it is ready to run but has not been assigned the CPU and is waiting in the scheduling queue. The thread is said to be runnable when it is allotted the CPU and is executing.
- 2) Create a child thread by implementing the Runnable interface wherein the child thread does string concatenation, and the main thread changes the string to uppercase.

```

import java.lang.Thread;
class mythread implements Runnable
{
    Thread t;
    String name;
    static String a, b;
    mythread(String n, String a1, String b1)
    {
        a=a1;
        b=b1;
        t=new Thread(this, n); //Child Thread created
        t.start(); //Child thread now ready to run
    }
    public void run()
    {
        System.out.println("child thread :");
        a.concat(b);
        System.out.println("After concatenation :" + a);
    }
}
public class Childthread
{
    public static void main(String[] args)
    {

```

```
String one="hello"; String two="world";
mythread t1=new mythread("child thread", one, two);
System.out.println("Main Thread ");
System.out.println("hello in uppercase looks like :" +toUpperCase(one));
}
}
```

**Output:**

```
Child thread
helloworld
Main Thread
HELLO
```

- 3) Repeat the above program by making child thread by extending the thread class.

```
//Program

public class myThread extends Thread
{
    static String a,b;
    String n;
    myThread(String n1, String a1, String b1)
    {
        super(n1);
        a=a1;
        b=b1;
        start();
    }
    public void run()
    {
        System.out.println("child thread :");
        a.concat(b);
        System.out.println("After concatenation :" +a);
    }
    public static void main(String args[])
    {
        myThread i1 = new myThread("Thread is ","hello", "world");
        System.out.println("Main Thread ");
        System.out.println("hello in uppercase looks like :" +toUpperCase("hello"));
    }
}
```

**Ouptput:**

```
Child thread
helloworld
Main Thread
HELLO
```

☛ **Check your Progress-3**

- 1) A child thread in Java can be created either by extending the thread class or by implementing the runnable interface. The second method by implementing the Runnable interface is preferred as it is felt that classes should be only extended if added functionality is required in the generalized version of the class. Hence, while making the child thread, generally, the thread class is usually not enhanced. Hence, keeping in the philosophy of the OOPs concepts, the Runnable interface method is added.
  - 2) Java scheduling environment is priority based. A thread with higher priority can take the CPU control from a thread with a lower priority. By default, each thread is associated with a priority called normal priority. The priority value can vary between the MIN-PRIORITY to MAX-PRIORITY. The macros associated with Java Priority environment are as follows:
    - i. NORM-PRIORITY=Value 5
    - ii. MIN-PRIORITY = Value 1
    - iii. MAX-PRIORITY=Value 10
  - 3) The `isAlive()` method helps a thread to know about the status of the called thread. The status returned is true if the called thread is still running else false. The syntax of the method is `boolean isAlive()`. This is generally used by the main thread to see the status of the child threads.
- ☛ Check your Progress-4**
- 1) Synchronization is a mechanism that ensures exclusive access to the shared code. This is like having a write lock on the sharable data in the database environment. In java ‘s thread environment, it is achieved through the object’s lock flag. Only the thread that has the object’s lock flag can execute the sharable code exclusively. All other threads needing to execute the sharable code will have to wait till the thread with the lock flag finishes execution.
  - 2) Interthread communication enables Threads to communicate or coordinate with each other with the help of methods like `wait()`, `notify()` and `notifyAll()`. These methods enable a thread to help get the waiting threads out of the object’s lock pool and other wait pools to ensure optimum utilization of CPU cycles.
  - 3) The threads can effectively solve the following problems:
    - i. Client -Server Simulation
    - ii. Database transactions

## 1.16 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 .

# UNIT 3 I/O STREAMS

Structure	Page No.
3.0 Introduction	
3.1 Objectives	
3.2 Introduction to I/O Class and Interfaces	
3.3 File Class and its Methods	
3.4 The AutoCloseable, Closeable and Flushable Interfaces	
3.5 I/O Exceptions	
3.6 Introduction to Stream Classes	
3.7 Byte Stream Classes	
3.8 Character Stream Classes	
3.9 Serialization	
3.10 Summary	
3.11 Solution /Answers to Your Progress	
3.12 References /Further Reading	

## 3.0 INTRODUCTION

Input refers to any piece of information required for completion of execution of a program ( generally provided by users) while output refers to the information that the program provides to the user. For the development of an application, both input and output are essential. Computer programs can enhance their usefulness when they are able to interact with the rest of the world, which consists of users, machines, computers, etc., by some means. Interaction here refers to input, output, or I/O operations. Previous chapters mainly focus on only one type of interaction i.e. interaction with a user either through command line prompt or through a graphical user interface. But it is only one means of computer program and world interaction in which the user acts as both source and destination of the information. Similarly, one other type of input/output, i.e. Text IO, is used for reading and writing text from a file. In addition to Text IO, Java features a much powerful and flexible input/output framework enabling diverse ways of I/O interaction (socket, files, etc.). Java also features networked communication thereby increasing the program's usefulness.

Since in java, every I/O interaction is carried out via program interface. The following are the points that the java input/output interface must address.

1. Identification of end of the line in a file when a different convention is used by the different operating systems.
2. Handling different encoding systems used by the same operating system
3. Handling of file naming and path naming conventions as they may be different for a different system.

These all create a challenge for Java developers to counter these issues for Java users while performing input/output operations from an external source/destination. Java addresses these issues by integrating the `java.io.*` package enabling users to worry-free input/output operations.

In this unit, we elaborate on what are streams, different types of streams, how to read data from the input stream, how to send data over the output stream, etc. The unit also discusses the concept of I/O classes and interface, File class, Byte stream, Character stream, and serialization in detail for proper understanding.

### 3.1 OBJECTIVES

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

- ... explain a clear view of the Java input/output hierarchy,
- ... use I/O class and interface in your program,
- ... use File class for handling files and directories,
- ... use I/O class for reading and writing data,
- ... Use different types of I/O stream, i.e. Byte stream and Character stream,
- ... Handling of I/O Exception in the programs, and
- ... use the concept of Serialization.

### 3.2 INTRODUCTION TO I/O CLASS AND INTERFACES

In java to perform I/O operation, the concept of stream is used. Stream is a sequential flow of input or output data. Stream enables java programs to serially accept input from various data sources such as socket, file, buffer, etc. Stream also helps the programmer to create a file and send data over the internet. The Input stream is used to read data from the input source, while the Output stream helps to write data to a destination. Here source and destination can be anything that generates, consumes and stores the data such as program, disk, peripheral device, etc. The operation of I/O in java is shown in fig.1.

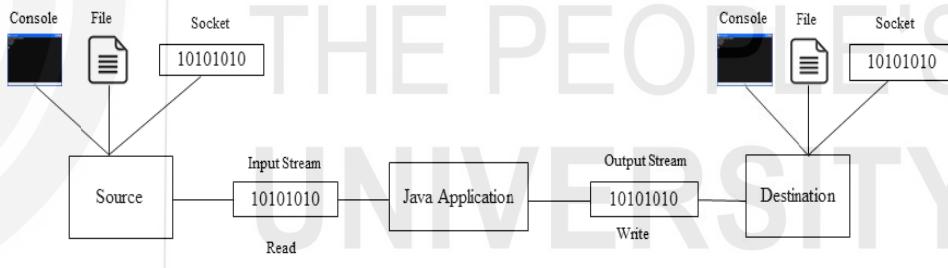


Figure 1: Java I/O

The section discusses different classes and interfaces defined in package `java.io`. The package mainly classifies the I/O class in 5 different class hierarchy sets: `InputStream`, `OutputStream`, `Reader`, `Writer`, and `File`. `Input Stream` and `Output Stream` are used for reading and writing byte data, while `Reader` and `Writer` classes are used for reading and writing the character data. Based on types of data (byte, file, character) and form of I/O device (file, array) further, these classes are classified. The hierarchy of the I/O classes has been illustrated in Figure 2.

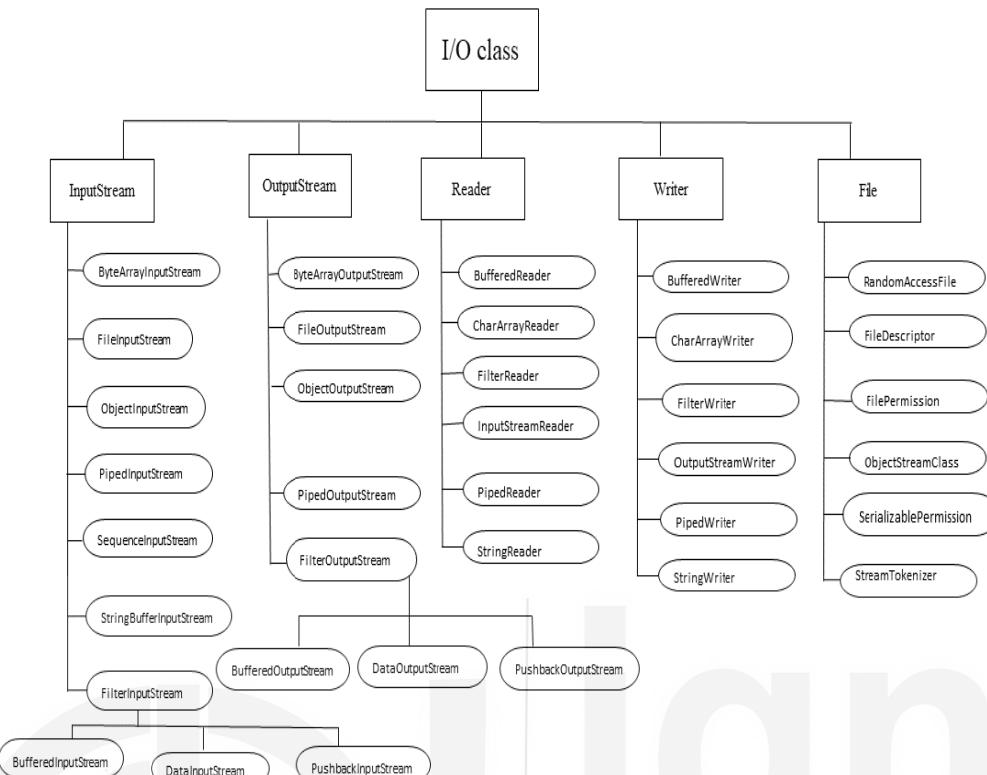


Figure 2: I/O Class hierarchy

Each of these classes defined above serves only one objective but, when combined with other classes can serve complex task objectives. For example, a BufferedReader allows buffering of input while a FileReader allows a method to connect file. When both are used together, the user can perform buffered reading from a connected file.

### 3.2.1 Java Interfaces

The Interface provides a way to perform input and output operations using data streams, file, and serialization. The list of interfaces provided by java.io is summarized in Table 1.

Table 1: List of Interfaces

Sr.No.	Interface & Description
1	<b>Closeable</b> Allows source and destination of data to be closed.
2	<b>DataInput</b> Allows reading of byte data from binary stream and converting these data to any of Java primitive types.

3	<b>DataOutput</b> Allows converting of Java primitive data to byte and write into binary stream.
4	<b>Externalizable</b> Enables instance of class object to save and restore its content.
5	<b>FileFilter</b> Provides filter for abstract pathnames.
6	<b>FilenameFilter</b> Provides filter for filenames.
7	<b>Flushable</b> Allows destination of data to flushed.
8	<b>ObjectInput</b> Extends DataInput interface for reading of objects and allow deserialization of objects
9	<b>ObjectInputValidation</b> Represents callback interface and enable validation of object within the graph.
10	<b>ObjectOutput</b> Extends DataOutput interface for writing of objects and allow serialization of objects
11	<b>ObjectStreamConstants</b> Enables constant to be written in Object Serialization stream.
12	<b>Serializable</b> Allows serialization of objects/class by implementing the java.io.Serializable interface.

### 3.3 FILE CLASS AND ITS METHOD

Most of the java.io package classes operate on stream concept, but the File class defined in java.io does not use stream concept. It directly interacts with the file and file system. The file class only describe the file properties but doesn't provide a view of how information related to file are stored or retrieved. For manipulating or obtaining information associated with a file, an object known as File object is created. The properties of file here represent permission, date and time of file creation and directory path of file. Even though there are some restrictions on using file (within applets) due to security issues, but still file is the primary source of storing and sharing information of many programs. The directory in java is a collection of files, whose filenames (list of files) can be fetched using the list() method.

The following are how an object of the File class can be created using a different form of File constructor.

```
File f1 = new File("c:/JavaPrograms/File_Name.txt");
File f2 = new File("c:/JavaPrograms", "File_Name.txt");
File f3 = new File("JavaPrograms", "File_Name.txt");
```

In the first form, *f1* file object is created by File constructor accepting only one string parameter. The string parameter consists of the path of the file and the file name. For *f1* filepath is c:/JavaPrograms while the name of the file is File\_Name.txt. Similarly, *f2* is created using the File constructor that accepts two parameters. The first parameter represents the path of the directory, while the second parameter represents the file name. Lastly, *f3* is created using File constructor, which accepts directory name and file name as the input parameter.

### 3.3.1 File Methods

There are many methods defined in the file class that can allow users to examine the properties of the file object. Some of them are listed below, along with their addressed function in Table 2.

**Table 2:** Methods of File

Method	Function
getName()	Returns the name of file.
boolean exists()	Returns true if file exists; otherwise, return false.
boolean canWrite()	Returns true if file is writable; otherwise, return false.
boolean canRead()	Returns true if file is readable; otherwise, return false.
boolean isFile()	Returns true if reference is a file and return false if reference is directories.
boolean isDirectory()	Returns true if reference is a directory; otherwise, return false.
String getAbsolutePath()	Returns application absolute path.
boolean renameTo(File newFileName)	Returns true if file name is renamed; otherwise, return false

To properly understand how the file object is created and how the file method can be accessed, we illustrate the concept by using the following program as an example.

```
//program
import java.io.File;
class DemoFile_1
{
    public static void main(String args[])
    {
        File fil_1 = new File ("/testfile_1.txt");
        System.out.println(" Display File name : " + fil_1.getName());
        System.out.println(" Specify the Path : " + fil_1.getPath());
        System.out.println("Absolute Path of file location: " + fil1.getAbsolutePath());
        System.out.println(fil_1.exists() ? "Do Exists" : "Doesn't exist");
        System.out.println(fil_1.canWrite()?"writable" : " not writable");
        System.out.println(fil_1.canRead() ? " file is readable" :" file not readable");
        System.out.println("Size of file : " + fil_1.length() + " bytes");
    }
}
```

**Output: (When testfile\_1.txt does not exist)**

Display File name: testfile\_1.txt  
Specify the Path: \testfile\_1.txt  
Absolute Path of file location : C:\testfile\_1.txt  
Doesn't exist  
Not writable  
File not readable  
Size of file : 0 bytes

**Output: (When testfile\_1.txt exists)**

Display File name : testfile\_1.txt  
Specify Path : \testfile\_1.txt  
Absolute Path of file location : C:\testfile\_1.txt  
Do Exists  
writable  
file is readable  
Size of file : 17 bytes

The following program checks whether the filename FileCreated\_1.txt is created or not. If the creation of the file is successful, then the output of program is “Creation of new File is successful!”. The else part will be generated as output if the file FileCreated\_1.txt is already present in the system.

```
import java.io.*;
public class FileCreation1
{
    public static void main(String[] args)
    {
        try
        {
            File f1_ex1 = new File("FileCreated_1.txt");
            if(f1_ex1.createNewFile())
            {
                System.out.println("Creation of new File is successful!");
            }
            else
            {
                System.out.println("File is already present.");
            }
        }
        catch (IOException fileexp)
        {
            fileexp.printStackTrace();
        }
    }
}
```

**Output of the program:** Creation of new File is successful!

The following program is used to check whether the folder jdk-15.0.1 consists of how many files and how many directories. The program use isDirectory() method for checking whether the inputted file is a directory or file. The list() method in the program is used to fetches all files and directories inside the folder jdk-15.0.1.

```

import java.io.File;
class DirCheckList
{
    public static void main(String args[])
    {
        File dir1 = new File("C:\\Program Files\\Java\\jdk-15.0.1");
        if (dir1.isDirectory())
        {
            System.out.println("Directory of " + "Java JDK");
            String str1[] = dir1.list();
            for (int i=0; i < str1.length; i++)
            {
                File chk_f1 = new File("C:\\Program Files\\Java\\jdk-15.0.1"+ "/" + str1[i]);
                if (chk_f1.isDirectory())
                {
                    System.out.println(str1[i] + " is a directory");
                }
                else
                {
                    System.out.println(str1[i] + " is a file");
                }
            }
        }
        else
        {
            System.out.println("C:\\Program Files\\Java\\jdk-15.0.1" + " is not a directory");
        }
    }
}

```

The output of program displays file in list and directory category.

```

Directory of Java jdk-15.0.1
bin is a directory
conf is a directory
COPYRIGHT is a file
include is a directory
jmods is a directory
legal is a directory
lib is a directory
release is a file

```

## Check Your Progress-1

1) Why do we need Java I/O Package?

.....  
 .....  
 .....  
 .....

2) Write the name of any three interfaces provided in java.io with a description.

.....  
 .....  
 .....

- 3) Write the functions of following methods: boolean exists(), boolean canRead(), boolean isDirectory()

.....  
.....  
.....  
.....

- 4) Draw and explain the class hierarchy of Java I/O?

.....  
.....  
.....  
.....

---

### 3.4 THE CLOSEABLE, FLUSHABLE, AND AUTOCLOSEABLE INTERFACES

---

**Closeable interface:** Closeable interface has only one abstract method i.e. close(). When a call to this method is invoked, system resources held by the stream object is released and these resources can be used further. Generally, stream classes implements Closeable interface and overrides close() method to close the stream. A sub class can use close() method, if its super class implements Closeable interface. E.g. close() method can be used by FileInputStream, because its super class InputStream implements Closeable interface.

**Flushable interface:** Like Closeable interface, Flushable interface also has only one abstract method i.e. flush(). When flush() method is called, data stored in the buffer is flushed out to be written to the destination file. Like Closeable interface, stream classes need to implement Flushable interface in order to override the flush() method.

Both close() and flush() method throws IOException.

**AutoCloseable interface:** Like Closeable interface, AutoCloseable interface also includes close() method, but this method is automatically called. An object that implements AutoCloseable interface, when it is done with the held resource with in a try-with-resources block, close() method is automatically called.

---

### 3.5 I/O EXCEPTIONS

---

Exceptions refer to the interruption that breaks the normal flow of program execution. For example, if you try to access a file that is not present, it will result in a FileNotFoundException exception. The following are different types of I/O Exceptions listed in Table 3.

**Table 3:** Different I/O Exceptions

Sr.No.	Interface & Description
1	<b>CharConversionException</b> Represent the base class responsible for handling character conversion

	exceptions.
2	<b>EOFException</b> Generate exception when (unexpected) end of stream or end of file is encountered during input.
3	<b>FileNotFoundException</b> Thrown while opening a file using a specified file path, but the file path is not present/invalid.
4	<b>InterruptedException</b> Thrown when I/O operation is interrupted.
5	<b>InvalidClassException</b> This is thrown when a problem is detected with a Class during Serialization runtime.
6	<b>InvalidObjectException</b> Thrown when one or more deserializable objects failed validation test.
7	<b>IOException</b> Thrown when some sort of I/O exception occurs.
8	<b>NotActiveException</b> Thrown when serialization and deserialization is inactive.
9	<b>NotSerializableException</b> This is thrown when an instance variable of failed to have Serializable interface.
10	<b>ObjectStreamException</b> Represent superclass of all exceptions occurred in object stream classes.
11	<b>OptionalDataException</b> Throws exception when serialized object of stream performing in read operation result in failure either due to unread primitive data or data end.
12	<b>StreamCorruptedException</b> Thrown when control information read from object stream fails consistency checks.
13	<b>SyncFailedException</b> Thrown when sync operation failed.
14	<b>UnsupportedEncodingException</b> Thrown when the character encoding is not supported.
15	<b>UTFDataFormatException</b> Thrown when unsupported format data( not in UTF-8 format) is read by input stream or by any class that implement data input interface.

The following program demonstrates that an exception will be thrown if we try to read a file that does not exist.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class DemoFileNotFoundException
{
    public void checkFileExistance()
    {
        try
        {
            FileInputStream f = new FileInputStream("abc.txt");
            System.out.println("File Exists");
        }
        catch (FileNotFoundException exceptionFileNotFoundException)
        {
            exceptionFileNotFoundException.printStackTrace();
        }
    }
}
```

```
        }
        public static void main(String[] args)
        {
            DemoFileNotFoundException demo = new DemoFileNotFoundException();
            demo.checkFileExistence();
        }
    }
```

**Output:**

```
java.io.FileNotFoundException: abc.txt <The system cannot find the file specified>
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at DemoFileNotFoundException.checkFileExistence(DemoFileNotFoundException.java:6)
        at DemoFileNotFoundException.main(DemoFileNotFoundException.java:15)
```

---

## 3.6 INTRODUCTION TO STREAM CLASSES

---

As discussed above in the chapter, the java program uses the concept of the stream to perform I/O. A stream can be defined as an abstraction which either consumes or produces information. To perform I/O operations with a physical device in Java, the system makes a link connection between the stream and physical device. Even though they are connected to different physical devices, all streams behaved in the same manner. Thus, it enables the input stream to obtain input from different input sources such as a disk file, keyboard, a network socket, etc.

Similarly, it allows the output stream to transmit output to different data destinations such as console, file disk, network connection. Stream provides an efficient way to perform input/output by the program without worrying about different types of physical devices. All the stream classes are implemented in the `java.io` package.

### Byte Streams and Character Streams

In Java basically, there are two types of streams: Byte Streams and Character Streams. Byte streams are used to deal with input and output of binary or byte data, while Character streams deal with characters. Though all I/O are byte-oriented at the lowest level but character streams provide a convenient method for handling character using Unicode representation and can be easily internationalized making it more useful.

Java I/O classes are mainly built on four abstract classes i.e. `InputStream`, `OutputStream`, `Reader`, and `Writer`. The first two classes are categorized as Byte stream class, while the latter two of them are categorized as Character stream class. Byte stream classes are discussed in section 3.7, and Character stream classes are discussed in detail in section 3.8. In order to implement these classes, one must import `java.io`.

---

## 3.7 BYTE STREAM CLASSES

---

Byte Stream class provides a rich repository for the user that wants to perform byte or binary I/O operations. Any type of object can access these stream classes thus making these stream classes important for many programs. Byte Stream classes are further classified into two abstract class hierarchies i.e `InputStream` and `OutputStream`. Further, with these two top hierarchy abstract classes, several concrete subclasses are

created, which ease the interaction when encountered with different types of devices such as files, network connection, buffer, etc. Fig 2. shows different types of stream classes. Later in the chapter few of these concrete subclasses are discussed.

Several key methods are defined in these two abstract classes that other types of stream classes can implement. For example, the `read()` and `write()` methods defined in `InputStream` and `OutputStream` are used for reading and writing byte data. These methods are abstract in nature and are overridden by any derived stream classes.

As `InputStream` class and `OutputStream` class are the top hierarchized class of byte stream, we start our discussion with `InputStream` class.

### **InputStream**

`InputStream` is an abstract class that models the streaming of byte input. As it is a top-hierarchy abstract class, it defines numerous methods that are used/implemented by various other concrete subclasses. Table 4 describes the methods defined in the `InputStream` class. The methods (all of them) defined in `InputStream` class throws `IOException` in case of occurrence of error.

**Table 4:** Methods of `InputStream`

<b>Method</b>	<b>Descriptions</b>
<code>int available( )</code>	Returns the estimation of available number of bytes present for reading.
<code>void close( )</code>	Results in closing the input source and releases the associated system resources. Throws <code>IOException</code> if further attempts of read is made.
<code>void mark(int num_Bytes)</code>	Marks current point in the input stream, and it remains valid until the size of <code>num Bytes</code> bytes are read.
<code>boolean markSupported( )</code>	Returns <code>true</code> if the invoking stream supports <code>mark()</code> / <code>reset()</code> .
<code>int read( ).</code>	Returns the byte of input stream that is next available in integer form if the end of the file is encountered during reading return -1.
<code>int read(byte buffer[ ])</code>	Returns the total size of byte that was successfully read from buffer when an attempt of reading <code>bufferlength</code> byte (input size) is made. If the end of the file is encountered during reading return -1.
<code>void reset( )</code>	Resets the current input pointer to previously set mark.
<code>long skip(long numBytes)</code>	Skips or Ignores input of size <code>numBytes</code> and return the number of bytes that are skipped.

### **OutputStream**

The next top hierarchy abstract class is the `Output stream` class. It is an abstract class that models the streaming of byte output. The methods(all of it) defined in this abstract class are void in nature and throws `IOException` when an error occurs. The methods defined in this class are summarized in Table 5.

**Table 5:** Methods of OutputStream

Methods	Description
void close()	Results in closing the output stream and releases associated system resources. Throws IOException if further attempts of write are made.
void flush()	Clears or flushes the output buffers.
void write(int b)	Writes b bytes of data to output stream.
void write(byte buffer[ ])	Writes an array of bytes to an output stream.
void write(byte buffer[ ], int offset, int num_Byte)	Writes an array of bytes of size num_Byte to output stream from the array buffer staring from buffer[offset].

Besides the two top hierarchy classes of Byte stream, other concrete subclasses are defined using the methods defined in InputStream and OutputStream class. Some of these subclasses are discussed as follows.

### FileInputStream

The FileInputStream class allows users to read bytes from a file. To create an input stream of FileInputStream, the following constructor is used.

`FileInputStream(String pathFile)`

`FileInputStream(File objtFile)`

Both these constructors throw FileNotFoundException in case any exception occurs (when no file is found or invalid path of the file). Here, *pathFile* represents the full pathname of the referenced file, while *objtFile* represents the object of the referenced file. Let's see how to create FileInputStream using these two constructors with an example. In the following examples, both constructors use the same disk file.

```
FileInputStream fl_1 = new FileInputStream("/FileDemo_1.java")
```

```
File fl_1 = new File("/FileDemo_1.java");
```

```
FileInputStream fl_2 = new FileInputStream(fl_1);
```

As stated in the above example, the first constructor uses file path name while the second uses File object to read byte from file. Commonly the first constructor is used to create FileInputStream, but we can use the second constructor when we want to examine File methods before attaching input streams. Besides mark() and reset() methods of InputStream, this class overrides the other six methods of InputStream class. Any attempt to override mark() and reset() methods, results in the generation of IOException. As soon as FileInputStream is created, it automatically opened.

### FileOutputStream

The FileOutputStream class allows users to write bytes to a file. To create the output stream of File OutputStream, the most common constructor is defined using the following syntax:

`FileOutputStream(String pathFile)`

```
FileOutputStream(File objtFile)
```

```
FileOutputStream(String pathFile, boolean app)
```

These constructors throw either FileNotFoundException or a SecurityException. Here, pathFile represents the full pathname of the referenced file, while objtFile represents the object of the referenced file. If app is true in the third constructor, then the file is opened in append mode. When the user tried to write byte data to a read-only file, IOException will be thrown.

### **ByteArrayInputStream**

ByteArrayInputStream creates an input stream that allows users to read data from byte array(input source). The class uses two types of constructors to create ByteArrayInputStream. In both these constructors, byte array is the source of input data. The syntax of the constructor is of the form:

```
ByteArrayInputStream(byte name_of_array[ ])
```

```
ByteArrayInputStream(byte name_of_array [ ], int start, int noBytes)
```

Here, *name\_of\_array* is the source of input. In the second constructor, input stream is created from a subset of byte array, i.e. *name\_of\_array*, starting from index denoted by *start* parameter till the length of *noBytes*.

The following examples are to illustrate how ByteArrayInputStream is created using the above-discussed constructor.

```
// Demonstrating the creation of byte input stream
import java.io.*;
class ByteArrayInputStream_Demo1
{
    public static void main(String args[]) throws IOException
    {
        String s1 = "howareyouinthishour";
        byte y_1[] = s1.getBytes();
        ByteArrayInputStream input_1 = new ByteArrayInputStream(y_1);
        ByteArrayInputStream input_2 = new ByteArrayInputStream(y_1, 0,3);
    }
}
```

The input\_1 contains howareyouinthishour, while input\_2 contains only how.

### **ByteArrayOutputStream**

ByteArrayOutputStream creates an output stream that allows users to write data to a byte array(output destination). The class uses two types of constructors to create ByteArrayOutputStream. The syntax of the constructor is of the form:

```
ByteArrayOutputStream()
```

```
ByteArrayOutputStream(int noBytes)
```

From the first constructor, a buffer of 32-byte is created, while from the second constructor buffer of size *noBytes* is created. The count field of ByteArrayOutputStream class is used to denote the number of bytes stored in the buffer. The count field is protected in nature.

Following program demonstrates the use of ByteArrayOutputStream class

```
import java.io.*;
class DemoByteArrayOutputStream
{
    public static void main(String args[]) throws IOException
    {
        ByteArrayOutputStream op=new ByteArrayOutputStream();
        byte b[]={“Have a nice day”.getBytes()};
        op.write(b);
        System.out.println(op.toString());
        op.close(); //closing the stream
    }
}
Output: Have a nice day
```

### Filtered Byte Streams

*Filtered streams* are the abstract class that supports basic input or output streams but with additional functionality. These streams are implemented by the class, which requires generic streams. The additional functionality here refers to buffering, character, and raw data translation. **FilterInputStream** and **FilterOutputStream** are types of filtered byte streams.

The constructor of this class is of the form:

**FilterInputStream(InputStream is)**

**FilterOutputStream(OutputStream os)**

**InputStream** and **OutputStream** methods and these classes' methods are similar in nature.

### Buffered Byte Streams

Buffered streams extend the **FilterStream** class and allow buffering of byte I/O stream by attaching a buffer memory. Due to this buffering, it allows java to perform I/O operation on more than one byte simultaneously, thereby increasing the program's performances. Buffer also allows users to perform skipping, marking and resetting of input streams. **BufferedInputStream** and **BufferedOutputStream** are buffered byte stream classes. **PushbackInputStream** also implements the buffered stream.

#### **BufferedInputStream**

One of the common ways of performance optimization is achieved by buffering of I/O. In java, this performance is achieved using **BufferedInputStream** class that enables buffering of input streams. The constructor of this class is of the following form:

**BufferedInputStream(InputStream streamInput)**

**BufferedInputStream(InputStream streamInput, int size\_Buf)**

In the first form of the constructor, a stream is created and buffered using the default size buffer while in the second form of the constructor , the stream is buffered in

buffer of size *size\_Buf*. For significant performance improvement, the buffer is usually taken in terms of a memory page, disk block.

The buffering also enables to trace stream in backward direction due to the storage of input. Hence, it supports mark() and reset() methods defined in InputStream class. For examining this support, `BufferedInputStream.markSupported()` is used that returns true when it supports the aforementioned methods.

### **BufferedOutputStream**

The BufferedOutputStream behaves similarly to other OutputStream but with an additional flush() method. The flush() method defined here confirms/ensures that the data in buffers are physically written to actual output devices. BufferedOutputStream also tries to improve the performance by decreasing the number of times writes data.

Following are the form of its constructor:

`BufferedOutputStream(OutputStream streamOutput)`

`BufferedOutputStream(OutputStream streamOutput, int buf_Size)`

The first one creates a buffer of 512 bytes for a write operation of output streams, while the second one creates a size of the buffer **buf\_Size** for writing output streams.

### **PushbackInputStream**

Buffering allows pushback operations in java. Pushback allows a byte to read from the input stream and then return this byte to the input stream. It allows users to see what is coming from the input stream without disturbing the flow of the input stream. In java, the pushback of byte is implemented via PushbackInputStream class.

Following are the form of constructor defined in PushbackInputStream:

`PushbackInputStream(InputStream input_stream)`

`PushbackInputStream(InputStream input_stream, int num_Bytes)`

The first constructor creates a stream object that allows readied byte to return to the input stream. The second constructor creates a buffer of multiple data of size *num\_Bytes* and returns it to the input stream.

### **SequenceInputStream**

When a user wants to concatenate multiple Input streams, it can use **SequenceInputStream** class. The constructor of this class either uses a pair of InputStream or Enumeration of InputStream class. Following are the form of the constructor:

`SequenceInputStream(InputStream first_Stream, InputStream second_Stream)`

`SequenceInputStream(Enumeration stream_Enum)`

## Multithreading and I/O

In the first case, the class starts reading data from the first InputStream until its end and then switch to the next or second InputStream for reading. While in the second case i.e. Enumeration, it will continue reading data until the last stream end is encountered.

### PrintStream

The PrintStream class offers all types of formatting capabilities for the stream as used from the starting of books, i.e. as in System.out and system filehandle.

Following are its constructor form:

`PrintStream(OutputStream output_Stream)`

`PrintStream(OutputStream output_Stream, boolean flush_On_Newline)`

Here `flush_On_Newline` controls the flushing of the output stream whenever a `\n` is encountered. If `flush_On_Newline` results in false, flush operation is not invoked automatically; else, it is automatically invoked.

These class objects support both `print()` and `println()` methods to accept all types of arguments /parameters, including objects. In both methods, if parameters are not of simple type, a call to `toString()` method is invoked, and then the result is printed.

### RandomAccessFile

When a user wants to randomly access a file, it can use RandomAccessFile class. This class is not derived from either `InputStream` or `OutputStream` class. It implements the `DataInput` and `DataOutput` interfaces defined in I/O methods. The class also supports the feature of positioning the file pointer. The defined constructor is of the following forms:

`RandomAccessFile(File file_Obj1, String mode_Access)` throws  
`FileNotFoundException`

`RandomAccessFile(String name_file, String mode_Access)` throws  
`FileNotFoundException`

In the first constructor, `file_Obj1` represents the name of the file to be open as a `File` object while in the second constructor, `name_file` represents the name of the file to be opened. The `mode_Access` here represents the type of access permitted regarding file access. If '`mode_Access =r`', the file can only be read but cannot be written. If '`mode_Access =rw`' then the file can be used for both reading and writing operations. If '`mode_Access =rwd`', the file is eligible for both read and write operations, and the changes made to files are directly written to the physical device.

To set the file pointer to the current point in the file, `seek()` method is defined.

`void seek(long new_Pos)` throws `IOException`

Here, the file pointer is set to a new position, i.e. `new_Pos` from the starting of the file. After this read or write operation is performed from this new position. Another method `setLength()` is used to increase or decrease the length of the file.

Following program writes a string “new end” at the end of the file, i.e. append a string “new end” using RandomAccessFile class.

```
import Java.io.*;
public class DemoRandomAccessFile
{
    public static void main(String args[])
    {
        try
        {
            RandomAccessFile raf = new RandomAccessFile("xyz.txt", "rw");
            raf.seek(raf.length());
            raf.writeBytes("\n new end \n");
        }
        catch (IOException e)
        {
            System.out.println("Error Opening a file" + e);
        }
    }
}
```

**Ouput:** Above program will append “new end” in xyz.txt file.

### ☛ Check Your Progress-2

1) What is the difference between Byte Streams and Character Streams?

.....  
.....  
.....  
.....

2) What is the difference between closeable and autocloseable interfaces?

.....  
.....  
.....  
.....

3) Which of the following classes is used for input and output in Byte streams?

- (a) FileInputStream (b) BufferedReader (c) BufferedWriter (d) File

.....  
.....  
.....  
.....

4) Which of the following class supports print() and println() methods?

- (a) System (b) BufferedOutputStream (c) PrintStream (d) System.out

- 5) Write a program for reading data from two or more input stream using SequenceInputStream class.

### 3.8 CHARACTER STREAM CLASSES

## Character Streams

Even though the byte stream classes are sufficient for handling any type of I/O operation, but the limitation of these classes is that they never directly work with Unicode characters. So, to support one of the main features, i.e. write once and run anywhere of java, support for direct character-oriented I/O is vital. Thus, in this section discussion of different character I/O classes is carried out. As seen earlier in this chapter, character stream classes are top hierarchized by two abstract classes, i.e. Reader and Writer. Many other stream classes use the methods defined in these classes. For example, `read()` and `write()` defined here are used by other stream classes to perform read and write character operations. So, let's begin with Reader abstract class.

Reader

This is an abstract class that models the streaming of character input in Java. IOExceptions are thrown by every method defined in this class on the occurrence of any error. The methods of this class are described in Table 6.

**Table 6:** Methods of Reader Class

Methods	Description
void close( )	Performs closing of source input. Throws IOException if any further read attempt is carried out.
void mark(int num_Chars)	A pointer (marker) is placed at current point in input stream, and the pointer is valid until num_Chars are read.
boolean markSupported( )	Returns true if stream supported mark()/reset().
int read( )	Reads a character from the input stream and return its integer representation when file end is encountered return -1.
int read(char buffer[ ])	Returns the actual number of characters that are successfully read when reading of characters of buffer.length is attempted in buffer, when file end is encountered return -1.

abstract int read(char <i>buffer</i> [ ], int <i>offset</i> , int <i>num_Chars</i> )	Returns the number of characters that were successfully read when the attempt of reading <i>num_Chars</i> character into buffer is performed starting from the <i>buffer</i> [ <i>offset</i> ]. When file end is encountered during reading return -1.
boolean ready( )	Returns false if the next input request is not ready(waiting) else, return true.
void reset( )	The input pointer is reset to the previously set mark.
long skip(long <i>noChars</i> )	Returns the number of characters that are successfully skipped when skip of <i>noChars</i> input character is carried out.

### Writer Class:

This is an abstract class that models the streaming of character output. The methods defined in this class are void in nature and throws IOException when an error occurred. The methods defined in this class are described in Table 7.

**Table 7:** Methods of Writer Class

Methods	Description
abstract void close( )	Used to perform the closing of the output stream. Throws I/OException if any further write attempt is carried out.
abstract void flush( )	Flushes the output buffers.
void write(int <i>ch</i> )	Used to perform writing of a single character to the invoked output stream.
void write(char <i>buffer</i> [ ])	An array of stream is written to the invoked output stream.
abstract void write(char <i>buffer</i> [ ], int <i>offset</i> , int <i>numChars</i> )	A subarray of the character of length <i>num_Chars</i> from <i>buffer</i> array starting from <i>buffer</i> [ <i>offset</i> ] is written to the invoked output stream.
void write(String <i>str</i> )	<i>str</i> is written to the invoked output stream.
void write(String <i>str</i> , int <i>offset</i> , int <i>num_Chars</i> )	A subrange of <i>num_Chars</i> character from array <i>str</i> starting from <i>str</i> [ <i>offset</i> ] is written to the invoked output stream.

### FileReader

The FileReader class allows users to read the contents (input stream) from the file. Syntax of two of its most used constructors are as follows:

FileReader(String *pathFile*)

FileReader(File *objtFile*)

Here *pathFile* represents the path where the file is located, while *objtFile* represent the File object of the invoked file.

In order to demonstrate how to read from a file is described using the following program. The program demonstrates reading of line from file and printing to the output stream.

```
// Demonstrating the reading of file using FileReader
import java.io.*;
class FileReader_Demo1
{
    public static void main(String args[]) throws Exception
```

```
{  
    FileReader rf_1 = new FileReader("FileReader_Demo1.java");  
    BufferedReader rb_1= new BufferedReader(rf_1);  
    String str;  
    while((str = rb_1.readLine()) != null)  
    {  
        System.out.println(str);  
    }  
    rf_1.close();  
}
```

### FileWriter class

The class allows users to write characters (input stream) to file. The most common constructors defined in this class are of the following syntax:

```
FileWriter(String pathFile)  
FileWriter(String pathFile, boolean app)  
FileWriter(File objtFile)  
FileWriter(File objtFile, boolean app)
```

These constructors defined here throws **IOException**. Here, *pathFile* represents full pathname of the referenced file while *objtFile* represents the object of the referenced file. If *app* is true, then the file is opened with appended output. FileWriter first creates a file by creating an object and then open it for output writing. When a user tries to open read-only file, IOException will be thrown.

### CharArrayReader

The class provides another method for users to read the input stream (character) from character array (Input source). The constructor of this class uses character array as a source of data reading. The form of constructors is as follows:

```
CharArrayReader(char name_Array[ ])  
CharArrayReader(char name_Array[], int begin, int num_Chars)
```

Here *name\_Array* is the data source, while in the second constructor, a subset of character of length *num\_Chars* is read from *name\_Array* starting with index specified by *begin* parameter.

### CharArrayWriter

The class provides another method for users to write the character stream (character)to the character array (data destination). The constructor of this class uses a character array as the destination of data writing. The form of constructor is as follows:

```
CharArrayWriter( )  
CharArrayWriter(int num_Chars)
```

In the first constructor, a default size buffer is created for performing write operations, while in the second constructor, a buffer size equal to *num\_Chars* size is created. CharArrayWriter uses *buf* field to denote/hold buffer information. Automatically the size of the buffer increase when required. A count field defined in the class is used to keep track of the number of characters in the buffer. Both these fields are protected fields.

## BufferedReader

The class enable the java program to increase its performance by attaching a buffer to the input stream. The buffer allows the user to hold character and allow to perform I/O operation on more than one character at a time. Following are the form of this class constructor.

`BufferedReader(Reader in_Stream)`

`BufferedReader(Reader in_Stream, int buf_Size)`

The first constructor creates buffered character stream using default buffer size, while in the second constructor the size of the buffer is set to *buf\_Size* for creating buffered character streams. Since we have seen that buffering enable moving stream in the backward direction by implementing *mark()* and *reset()* methods and return true when checked by calling **BufferedReader.markSupported()**.

The following program demonstrates the reading of data using BufferedReader class.

```
import java.io.*;
public class BufferedReaderDemo
{
    public static void main(String[] args) throws IOException
    {
        InputStreamReader abc = new InputStreamReader(System.in);
        BufferedReader xyz = new BufferedReader(abc);
        String str = "";
        System.out.print("Please enter a string: ");
        str = xyz.readLine();
        System.out.println("Entered string is: " + str);
    }
}
```

### Output:

Please enter a string: abcdef

Entered string is: abcdef

## BufferedWriter

The BufferedReader stream behaves in a similar fashion as other Writer class but with additional *flush()* method. The *flush()* method defined here makes the data in buffers to be physically written to the actual output stream. BufferedWriter also tries to improve the performance by decreasing the number of times actually writes operation is performed.

Following are the form of these constructors:

BufferedWriter(Writer *output\_Stream*)

BufferedWriter(Writer *output\_Stream*, int *buf\_Size*)

The first one creates a buffer of 512 bytes for write operation of output character streams, while the second create a size of the buffer *buf\_Size* for writing output streams.

The following program demonstrates the writing of data to a file using BufferedWriter class.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
public class DemoBufferedWriter
{
    public static void main(String args[])
    {
        String str = "This is the output file";
        try
        {
            FileWriter f = new FileWriter("output.txt");
            BufferedWriter op = new BufferedWriter(f);
            op.write(str);
            op.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

### PushbackReader

Buffering enables pushback operations in java. PushbackReader allows one or more character to read from input stream and then return these characters to input stream. Basically, it allows users to see what is coming from input stream without disturbing the flow of input stream. In java, push back of character is implemented via PushbackReader class. Following are the form of its constructor:

PushbackReader(Reader *inStream*)

PushbackReader(Reader *inStream*, int *bufSize*)

In the first constructor, the first character is read and push backed to *in\_Stream*, while the second constructor allows pushback of stream equal to the size of *bufSize*. One must check whether the pushback buffer is full or not before returning a character. If the pushback buffer is full, it results in IO Exception. If we want to return one or more than one characters to the input stream, we can use unread() method of PushbackReader.

### PrintWriter

PrinterWriter works similar to Print Stream and can be thought of as character oriented version of PrintStream. It provides all types of formatting functionality by defining print() and println() methods. Following are its Constructor forms:

`PrintWriter(OutputStream output_Stream)`

`PrintWriter(OutputStream output_Stream, boolean flushOn_Newline)`

`PrintWriter(Writer output_Stream)`

`PrintWriter(Writer outputStream, boolean flushOn_Newline)`

Where `flushOn_Newline` controls flushing of output whenever `println()` is invoked. Flushing of output is done automatically when `flushOn_Newline` is true else flushing is not automatic. This class objects provide support for both `print()` and `println()` methods to accept all types of arguments /parameters, including object also. In both methods, if parameters passed are not of simple type, a call `toString()` method is invoked, and then the result is printed.

## 3.9 SERIALIZATION

Java not only allows users to read and write text but also provides support to read and write objects of file. With serialization, users can write the state of objects to a byte stream. Serialization is useful to users as they allow users to save the content of their program in any non-volatile storage (file, memory) and later allows the users to restore these objects when required or necessary. The restoring process is known as Deserialization.

In Java, Remote Method Invocation (RMI) is implemented with the help of Serialization. With RMI in Java, the object of one machine can invoke methods of another object located on a different machine. In RMI, the object is passed as an argument to the invoked remote method. Before passing this object as a parameter, serialization is used to serialize the passed object and then transmit it. On receiving the serialized object, the destination machine deserialize it.

One important question that arises is that how serialization in Java is easy as compared to serialization in another language. For instance, suppose an object has a reference to another object, and the referenced object has a reference to any other object. In some cases, they may form circular references among objects creating a lot of confusion or chance of inconsistency. Java efficiently handles this situation by doing the following things. If you serialize one object, all its referenced objects are recursively located and serialized. Similarly, when performing deserialization all its referenced objects get restored correctly in the same order. In java if you want to serialize an object, you have to call the `writeObject()` method defined in  `ObjectOutputStream`, and when you want to deserialize any object, invoke the `readObject()` method defined in  `ObjectInputStream`. Serialization also enables transferring of object state over the network i.e Marshalling of objects. Implementation of a `Serializable` interface is also necessary for the serialization of objects.

Following are the interfaces and class that supports serialization.

### Serializable

The object that implements `Serializable` interfaces can only be saved and restored using the serialization process. No members are defined in the `Serializable` interface. To serialize the object of the class, the class must be declared as public and the class must have no parameter constructor. All subclass of the serializable class is also

serializable. Transient and static variables are not saved by serialization facilities. java.io.Serializable interface is by default implemented by the string and all wrapper class. In the following example, Employee class implements Serializable interface. Now object of Employee class can be converted into stream.

```
import java.io.Serializable;
public class Employee implements Serializable
{
    int e_id;
    String e_name;
    public Employee(int e_id, String e_name)
    {
        this.e_id = e_id;
        this.e_name = e_name;
    }
}
```

### **Externalizable**

In java, serialization and deserialization have automatically made the state of an object to be stored and restored. But in some cases, the programmer desires to control this automatic process. For example, when implementing encryption and compression techniques, the programmer can use Externalizable interface to control these automatic processes.

#### **ObjectOutput:**

The interface extends the DataOutput interface and provides support for object serialization, For making object serializable, call to writeObject() method defined in this interface is invoked. Methods defined in this interface throws IOException in case of error occurrence.

#### **ObjectOutputStream:**

This class implements the ObjectOutput interface along with extending OutputStream class for writing serializable objects into the output stream. Following is the form of this class constructor:

ObjectOutputStream(OutputStream **out\_Stream**) throws IOException

Here the serializable objects are written to the output stream, i.e. out\_Stream.

#### **ObjectInput:**

The interface extends the DataInput interface and provides support for object deserialization. For making object deserializable, call to readObject() method defined in this interface is invoked. The method defined in this interface throws IOException in case of error occurrence.

#### **ObjectInputStream:**

This class implements the ObjectInput interface along with extending OutputStream class for reading serializable objects from the input stream. Following is the form of this class constructor:

ObjectInputStream(InputStream *in\_Stream*) throws IOException,  
StreamCorruptedException

Here from the input stream, i.e. *in\_Stream*, the serializable objects are read.

For an illustration of how object serialization and deserialization take place, we use the following program as an example. First, an object of class MyClass is instantiated. The objects define three instance variables of types int, String, and double. For this example, we tried to save and restore the information of these three-instance variables.

First, we create FileOutputStream with the file name “DemoSerial”. Then ObjectOutputStream is created for the created file stream. Then for object serialization, we call writeObject() method defined in ObjectOutputStream. After serialization, it's flushed and closed.

Now we create FileInputStream by referring to the same file, i.e. “DemoSerial”. Then we create ObjectInputStream for the respective input stream. Then we perform deserialization of object using readObject() method of ObjectOutputStream class. Then this object input stream is closed.

Here we defined MySeriClass that implements Serializable interface. In case MySeriClass does not implement Serializable, it will throw NotSerializableException error.

```
import java.io.*;
public class DemoSerialization1
{
    public static void main(String args[])
    {
        // Object Serialization
        try
        {
            MySeriClass objt_1 = new MySeriClass("Welcome Ram", 85, 99.5);
            System.out.println("First object: " + objt_1);
            FileOutputStream filestr_1 = new FileOutputStream("DemoSerial");
            ObjectOutputStream objtstr_1= new ObjectOutputStream(filestr_1);
            Objtstr_1.writeObject(objt_1);
            Objtstr_1.flush();
            Objtstr_1.close();
        }
        catch(Exception exp)
        {
            System.out.println("Catching Exception: " + exp);
            System.exit(0);
        }

        // Deserialization of object
        try
        {
            MySeriClass objt_2;
            FileInputStream filestr_2= new FileInputStream("DemoSerial");
            ObjectInputStream objtstr_2 = new ObjectInputStream(filestr_2);
            objt_2 = (MySeriClass)objtstr_2.readObject();
            objtstr_2.close();
            System.out.println("Second object: " + objt_2);
        }
    }
}
```

```
        catch(Exception exp)
        {
            System.out.println("Catching Exception : " + exp);
            System.exit(0);
        }
    }
}
class MySeriClass implements Serializable
{
    String str;
    int a1;
    double db;
    public MySeriClass(String str, int a1, double db)
    {
        this.str = str;
        this.a1 = a1;
        this.db = db;
    }
    public String toString()
    {
        return "str=" + str + "; a1=" + a1 + "; db=" + db;
    }
}
```

**Output:**

First object: str=Welcome Ram; a1=85; db=99.5

Second object: str=Welcome Ram; a1=85; db=99.5

The same output of object instance variables here represents successful serialization and deserialization operation.

**☛ Check Your Progress-3**

1) What are Serialization and Deserialization?

.....  
.....  
.....  
.....

2) What is the role of Externalizable?

.....  
.....  
.....  
.....

3) Which of the following classes is used for input and output in Character streams?

- (a) BufferedReader (b) FileInputStream (c) FileOutputStream (d) InputStream

- 4) Which of the following exceptions, read() method throws?

  - (a) InterruptedException (b) IOException (c) SystemInputException
  - (d) SystemException

5) Write a program to read only first five character from character array C\_arr[]={‘E’,’x’,’c’,’e’,’l’,’l’,’e’,’n’,’t’} using CharArrayReader Class. Also write a separate program to the write the content of character array CharArr[] to a file name “CharWriteClass.txt” using CharArrayWriter Class.

### 3.10 SUMMARY

Generally, java-based applications process some inputs and, based on the input generates output. The source of input and the destination of output may be a file, physical devices, network, etc. Java provides an API, JAVA IO, which helps in reading and writing the data, i.e. input and output. This chapter provides a clear view of the Java input/output hierarchy. This chapter also explained how various classes and interfaces in the IO package are organized and their purpose. Further, the File class is discussed that helps in handling files and directories. Various IO exceptions are discussed and summarized. Java supports two types of streams: Byte Streams and Character Streams. Both streams and associated classes are discussed in the chapter. This chapter discusses serialization that allows us to represent an object in a sequence of bytes, i.e. byte stream.

### 3.11 SOLUTION/ANSWERS TO YOUR PROGRESS

## ☛ Check Your Progress-1

- 1) Java I/O Package consists of input streams and output streams. The Input stream is used to read data from input source, while the Output stream helps to write data to destination. Here source and destination can be anything that generates, consumes and stores the data such as program, disk, peripheral device, etc.
  - 2) DataInput: Allow reading of byte data from the binary stream and converting these data to any of Java primitive types.

FileFilter: Provides filter for abstract pathnames.

ObjectOutput: Extends DataOutput interface for the writing of objects and allow serialization of objects

3)

boolean exists(): Returns true if file exists otherwise return false.

boolean canRead(): Returns true if file is readable otherwise return false.

boolean isDirectory(): Returns true if reference is a directory otherwise return false.

4) The diagram of class hierarchy of I/O is shown in figure 2 in section 3.2.

### ☛ Check Your Progress 2

1) Byte streams process the data byte by byte, while Character streams process the data character by character in Unicode.

2) Closeable interface has only one abstract method i.e. close(). When a call to this method is invoked, system resources held by stream object are released, which can be used further in the program. Like Closeable interface, AutoCloseable interface also includes close() method but this method is automatically called unlike Closeable interface.

3) FileInputStream

4) PrintStream

5) In order to show how to perform concatenation of multiple input streams, a program is written. The program concatenates two file input streams named BufferReadFile.txt and ReadFile.txt, respectively. The SequenceInputStream first read the input from BufferReadFile.txt and then read input from ReadFile.txt (sequentially one by one in the order mentioned in the program).

```
import java.io.*;
class SequenceInStreamEx
{
    public static void main(String args[])throws Exception
    {
        FileInputStream fin_str1 =new FileInputStream("D:\\BufferReadFile.txt");
        FileInputStream fin_str2=new FileInputStream("D:\\ReadFile.txt");
        SequenceInputStream con_seq=new SequenceInputStream(fin_str1, fin_str2);
        int re;
        while((re=con_seq.read())!=-1)
        {
            System.out.print((char)re);
        }
        con_seq.close();
        fin_str1.close();
        fin_str2.close();
    }
}
```

### ☛ Check Your Progress 3

1) With serialization, users can write the state of objects to a byte stream. Serialization is useful to users as they allow users to save the content of their program in any non-volatile storage (file, memory) and later allows the users to restore these objects when required or necessary. The restoring process is known as Deserialization.

2) Externalizable is used to customize the serialization. In java, serialization and deserialization have automatically made the state of an object to be stored and

restored. But in some cases, the programmer desires to control this automatic process. For example, when implementing encryption and compression techniques, the programmer can use Externalizable interface to control these automatic processes.

**3) BufferedReader**

**4) IOException**

**5) Program 1: For reading character using CharArrayReader Class**

```
import java.ioCharArrayReader;
public class ProgressCheck3
{
    public static void main(String[] ag) throws Exception
    {
        char[] C_arr = {'E', 'x', 'c', 'e', 'l', 'l', 'e', 'n', 't' };
        CharArrayReader ch_rd1 = new CharArrayReader(C_arr,0,5);
        int k = 0;
        // Read until the end of a file
        while ((k = ch_rd1.read()) != -1)
        {
            char ch_read = (char) k;
            System.out.println(ch_read);
        }
    }
}
```

**Output:-**

```
E
x
c
e
l
```

**Program 2: For writing content of C\_arr to file name “CharWriteClass.txt” using CharArrayWriter Class.**

```
import java.ioCharArrayWriter;
import java.io.FileWriter;
public class CharProgramWrite3
{
    public static void main(String args[])throws Exception
    {
        char[] C_arr = {'E', 'x', 'c', 'e', 'l', 'l', 'e', 'n', 't' };
        String conv_str1 = new String(C_arr);
        CharArrayWriter arr_wr1 =new CharArrayWriter();
        arr_wr1.write(conv_str1);
        FileWriter new_f1=new FileWriter("D:\\CharWriteClass.txt");
        arr_wr1.writeTo(new_f1);
        new_f1.close();
        System.out.println("Successfully Written");
    }
}
```

**Output:** Successfully Written

---

## 3.12 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.
- ... <https://docs.oracle.com/javase/tutorial/essential/io/>



---

## **UNIT 4    JAVA API**

---

<b>Structure</b>	<b>Page No.</b>
4.0 Introduction	
4.1 Objectives	
4.2 Date and Time	
4.3 Set	
4.3.1 Classes that implement Set interface	
4.3.2 Interface that extends Set interface	
4.4 Map	
4.5 HashMap	
4.6 List	
4.7 Vector	
4.8 Stack	
4.9 Summary	
4.10 Solutions/ Answer to Check Your Progress	
4.11 References/Further Reading	

---

### **4.0 INTRODUCTION**

---

In this unit, you are going to learn about Java Application Programming Interface (API) for Date, Time and the Collections. The Java API is a group of pre-written packages, classes, interfaces along with their methods and attributes. You can import a particular package or a class as per your requirement and use its pre-written code in your program to perform basic and advanced programming tasks. This, in turn, reduces the lines of code of your program. Java API is a component of Java Development Kit. Java 8 introduced a completely new API in the `java.time` package that comprises a large number of classes to perform various Date and Time operations.

Collections API in Java is used for storing and manipulating a group of objects. The Collection framework is contained in `java.util` package. Students are advised to refer to Figure 1 in order to understand this section on the Introduction of Java API. The `Iterable` interface is the root interface of all collection classes and is extended by the `Collection` interface. All derived classes of collection interface implement the `Iterable` interface.

The Collection framework comprises interfaces like `Collection` (Figure 1) and `Map` (Figure 2). `List`, `Set` and `Queue` are the subinterfaces of the `Collection` interface, as shown in Figure 1. For the implementation of the `List` subinterface, there are Standard classes such as `ArrayList`, `LinkedList` and `Vector`. Further, the `Vector` class is inherited by `Stack` class. Next, for implementation of the `Set` subinterface, there are Standard classes such as, `HashSet`, `LinkedHashSet` and `TreeSet`. For the implementation of `Queue` subinterface, `PriorityQueue` class and `ArrayDeque` class are used.

There are few classes that provide an implementation of the `Map` interface. Student should refer Figure 2, which shows hierarchy of `Map` interface. `AbstractMap` class provides an implementation of the `Map` interface, and further `HashMap` class extends `AbstractMap` class in order to use a hash table. As shown in Figure 2, the `SortedMap` interface extends the `Map` interface and is inherited by the `TreeMap` class.

Finally, as discussed earlier Collection interface extends the Iterable interface, which is the root interface of all collection classes. The iterator() method of the Iterable interface facilitates the traversal of the elements in a collection. You can perform searching, sorting, insertion, and deletion easily and efficiently with Java Collections.

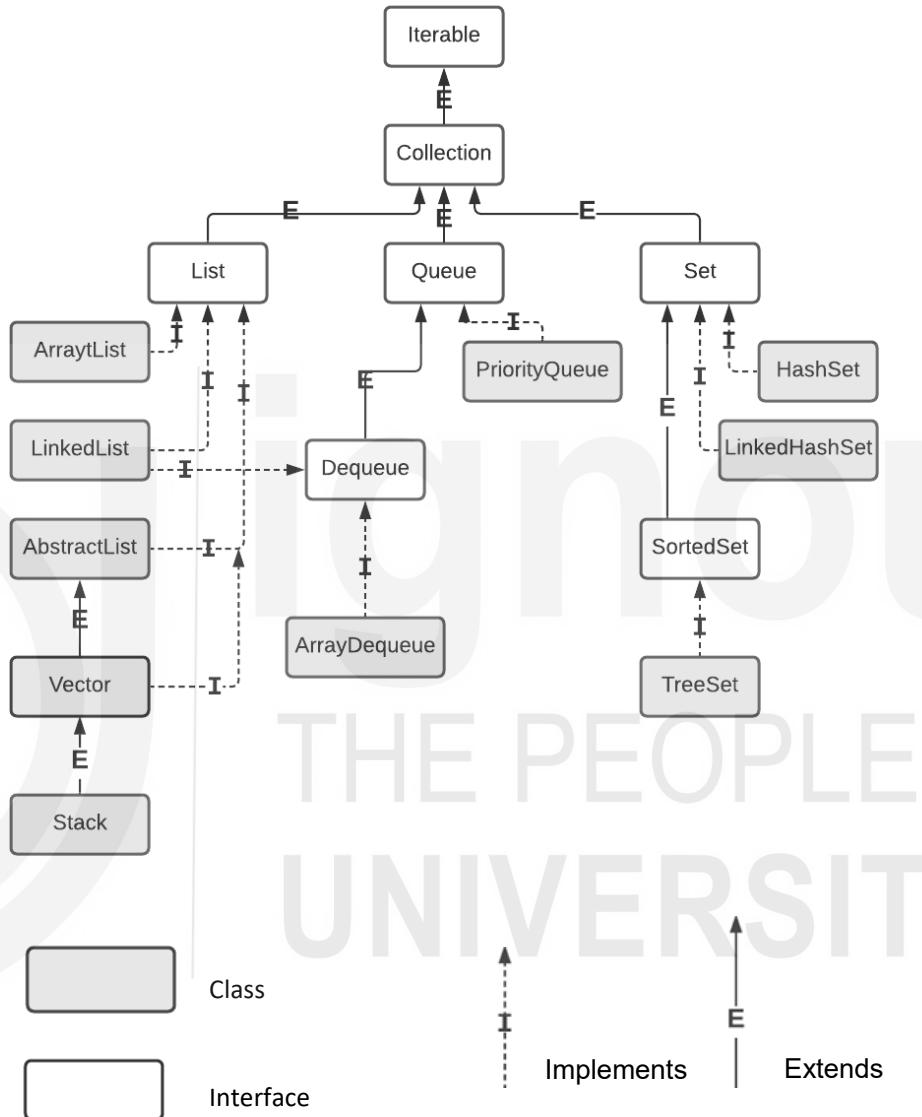


Figure 1. Collection Hierarchy in Java

## 4.1 OBJECTIVES

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

- ... explain the benefits of Java APIs,
- ... write programs that compare different date and time values,
- ... describe the use of various interfaces and classes of collection framework,
- ... use various interfaces, namely Set, Map and List, in order to write programs in a simpler way,

- ... use Traversal of Set using Iterator, and
- ... write programs by using the methods to perform various operations on maps.

## 4.2 DATE AND TIME

The older Date class before Java 8, was provided by java.util package, and it offered various functions related to date and time, as described in Table 1. There are two constructors for the Date class as given below: -

- ... Date( ) : It constructs Date object and initializes it with the current date and time.
- ... Date(long millisec) : It receives an argument corresponding to the quantity of milliseconds that have passed after twelve midnight, 1<sup>st</sup> January, 1970.

Table 1: Date class and its methods

S.No.	Method	Description
1	boolean after(Date dt)	It gives true when the Date object that invoked <b>after</b> the method has a value of date which is later than the one stated by object dt, else gives false.
2	boolean before(Date dt)	Gives true when the Date object that invoked <b>before</b> the method has a value of date which is before the one stated by object dt, else gives false.
3	Object clone( )	Returns duplicate object of the Date object that invoked clone method.
4	int compareTo (Date dt)	Compares the value date of invoking object to the value of date of dt object. If both are equal, this method returns 0. It returns a negative value if the value of date of invoking object is earlier than that of dt object. It returns a positive value if the value of date of invoking object is after the date of the dt object.
5	int compareTo (Object ob)	This function is identical to compareTo(Date) provided ob is an object of class Date. If not, an exception named ClassCastException is thrown.
6	boolean equals (Object dt)	Gives true when the Date object that invoked <b>equals</b> method has the same time and date value as mentioned in dt object, else the method gives false.
7	long getTime( )	It gives value of time in terms of number of milliseconds passed since 1 <sup>st</sup> January 1970 (midnight).

8	int hashCode()	Gives the hash code of the object that invoked this method
9	void setTime(long t)	Sets the current time and date as stated by t. The time in long corresponds to the time in milliseconds that have passed since midnight, 1 <sup>st</sup> January, 1970.
10	String toString()	Changes the Date object that invoked this method into a string, and the same is returned.

```

import java.util.Date;
class DateEx
{
    public static void main(String args[])
    {
        // Create an object of Date class
        Date date = new Date();
        // Print time and date
        System.out.println(date);
        // Retrieve the number of milliseconds since midnight, January 1, 1970 GMT
        long msec = date.getTime();
        System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
        // Create a Date object with msec value (as long datatype) as passed to constructor
        Date date1 = new Date(1599117817589l);
        // Retrieve the number of milliseconds in date1 object
        long msec1 = date1.getTime();
        // Compare current date with date1 object and print message accordingly
        int result = date.compareTo(date1);
        if(result==0)
            System.out.println("Both dates are same");
        else if(result==1)
            System.out.println(msec+" is later than "+msec1);
        else
            System.out.println(msec+" is earlier than "+msec1);
    }
}

```

### Output

```

Tue Nov 03 12:04:32 IST 2020
Milliseconds since Jan. 1, 1970 GMT = 1604385272113
1604385272113 is later than 1599117817589

```

### Date Formatting Using SimpleDateFormat

In order to give an understandable format for date, there is a class called `SimpleDateFormat`. This class allows us to choose any pattern for displaying date-time. You can also change a string to date by means of an appropriate method of this class.

```

import java.util.*;
import java.text.*;
public class Date_form
{
    public static void main(String args[])
    {
        Date date = new Date();
        SimpleDateFormat date_format = new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
    }
}

```

```

        System.out.println("Current Date: " + date_format.format(date));
    }
}

```

**Output**

```
Current Date: Tue 2020.11.03 at 12:16:55 PM IST
```

There are various codes for SimpleDateFormat. These are given below in Table 2

Table 2: SimpleDateFormat codes

Character	Description	Example
G	Era designator	AD
y	Year in four digits	2001
M	Month in year	July or 07
d	Day in month	10
h	Hour in A.M./P.M. (1~12)	12
H	Hour in day (0~23)	22
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)
w	Week in year	40
W	Week in month	1
a	A.M./P.M. marker	PM
k	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	10
z	Time zone	Eastern Standard Time
'	Escape for text	Delimiter
"	Single quote	'

This Date-Time API was not thread-safe and offered very few operations related to date. With the advent of Java 8, these drawbacks were removed as a new Date-Time API was introduced, which is immutable, doesn't have setter methods and includes many date operations. Two important classes present in this API are mentioned below: -

- ... LocalDate: It is a simplified date-time API, and there is not any complexity of timezone handling.
- ... ZonedDateTime: It is a special date-time API that makes you deal with various timezones in an easy manner.

### ☛ Check Your Progress-1

- 1) What do you understand by the Collection framework in Java?

.....  
.....  
.....  
.....

- 2) Which of these classes is not part of Java's collection framework?

- a) Array
- b) Maps
- c) Stack
- d) Queue

- 3) How do you know if two date objects are equal?

.....  
.....  
.....

---

## 4.3 SET INTERFACE

---

We can define a set as a group of dissimilar objects. We all know that sets are one of the most fundamental concepts in mathematics. Java provides Set as an interface that extends the Collection interface, and hence it has access to all the methods inherited from the Collection interface, as explained in Table 3.

Table 3: Description of methods of Collection interface

No.	Method	Description
1	public boolean add(object o)	It is written to insert an object in the collection. Returns true if object is added.
2	public boolean addAll(Collection c)	The purpose is to insert the elements of the mentioned collection c, in the collection that invoked the addAll method. The addAll() method returns true if the collection got changed because of the method call otherwise, false is returned.
3	public boolean remove(Object o)	The purpose of this method is to remove an object from a particular collection that invoked this method. It returns true if object is removed.
4	public boolean removeAll(Collection c)	The purpose of this method is to remove all the elements of the mentioned collection c, from the collection that invoked this method. The removeAll() method returns true only if the invoking collection was changed otherwise, false is returned.
5	default boolean removeIf(Predicate filter)	The purpose of this method is to remove all the elements of the collection that satisfy the given predicate. The parameter 'filter' represents a predicate which returns true for the elements to be removed. This method returns true if we are able to remove elements.
6	public boolean retainAll(Collection c)	The purpose of this method is to remove all the elements of invoking collection except the mentioned collection c. It returns true if it is successful.
7	public int size()	The purpose of this method is to tell the count of elements of the invoking collection.
8	public void clear()	This method deletes all elements of the mentioned collection.
9	public boolean contains(Object o)	With this method, we can search an object in the collection. The method will give true when the object is found.
10	public boolean containsAll(Collection c)	With this method, we can search the stated collection in the invoking collection. Again, the method will give true when the collection c is found.
11	public Iterator iterator()	It gives an iterator over elements of the invoking collection.
12	public Object[] toArray()	It changes the invoking collection to an array and returns the same.

13	public <T> T[] toArray(T[] a)	It changes the invoking collection to an array and returns the same. In this method, the returned array's runtime type is same as that of the mentioned array.
14	public boolean isEmpty()	This method would test whether the invoking collection is empty and returns true if it is so.
15	public boolean equals(Object element)	This method tests the two collections for similarity and returns true if they match.
16	public int hashCode()	This method gives the hash code value of the invoking collection.

Set represents an unordered group of members, and the members must be unique. So, a false is returned by the add() method when you try to add duplicate elements in it. You are allowed to have just one null value in a Set. There are no extra methods defined by Set; only the methods of collection interface are available for use with Set.

#### 4.3.1 Classes that implement Set interface

The Set interface is implemented by HashSet and LinkedHashSet class. There is an interface named SortedSet that extends Set interface, and the class TreeSet implements the Sortedset interface.

**HashSet class:** Its purpose is to produce a collection and keep it in a hash table. Each element in the Set has a unique value of its own, called hash code which is utilized as an index at which element related to that hash code is present. Hashing ensures that the execution time of operations like add(), remove(), remain constant irrespective of the size of the HashSet. The members are inserted in HashSet irrespective of the order of insertion, and the order depends on the hashcode of elements. Constructors for HashSet class are given below: -

- ... HashSet( ) : Constructs a default HashSet which is empty.
- ... HashSet(Collection coll) : Initializes the HashSet with the members of collection coll.
- ... HashSet(int cap) : Initializes the HashSet with the size of the integer value of cap.
- ... HashSet(int cap, float fill\_Ratio) : This constructor is used to initialize the data members - capacity and the fill ratio, of HashSet. The valid value of fill ratio has to be between 0.0 and 1.0, and it governs how full the HashSet is allowed to be before it is resized upwards. If the number of elements in HashSet is more than the capacity (cap) of the Hashset multiplied by its fill ratio, then the hash set is lengthened.

Corresponding to the first three constructors, the value of fill ratio is 0.75.

For instantiating a object of the HashSet class, we have to mention the data type of elements of set. For example:-

```
HashSet<datatype> hset = new HashSet<datatype>();
```

Here, datatype is the type of elements that can be stored in hset.

```
import java.util.*;
class HashSetProgram
{
    public static void main(String args[])
    {
        // create a HashSet
        HashSet<String> hset1 = new HashSet<String>();
        // add elements to the hash set
        hset1.add("Java");
        hset1.add("C++");
        hset1.add("Python");
        hset1.add("R");
        hset1.add("Matlab");
        hset1.add("Scala");
        System.out.println(hset1);
    }
}
```

## Output

[Java, C++, R, Scala, Python, Matlab]

We can see that the set elements are not printed as per the order of insertion of elements.

## Traversal of Set using Iterator

If we want to pass over the elements of any collection, we can make use of Iterable interface, which is the most important interface of the collection framework and is at the root of the hierarchy. As we know that the Iterable interface is extended by collection interface; hence all the interfaces and classes of collection framework implement this interface. The most important feature of the Iterable interface is to deliver an iterator for the collections. Iterable interface has just one method which is abstract and whose name is iterator(). It returns an object of type Iterator.

Iterator iterator();

The methods declared by Iterator are described in Table 4.

Table 4: Description of methods of Iterator interface

S.No.	Method	Description
1.	boolean hasNext( )	Gives true only if there are more elements in the collection. Else, this method returns false.
2.	Object next( )	Gives the next element. This method throws NoSuchElementException if there does not exist any next element.
3.	void remove( )	Eliminates the current element. It throws IllegalStateException if an attempt is made to invoke remove( ) method that is not preceded by a call to next( ) method.

In order to use an iterator to pass over the elements of a collection, you should follow the following steps.

1. Get an iterator to the start of the collection by invoking the iterator( ) method of the collection.
2. Write a while loop that invokes hasNext( ). The loop would iterate till the time hasNext( ) returns true.
3. Inside the loop, get and display each element by invoking next( ) method.

**Same program is written here again by using iterator.**

```
import java.util.*;
class HashSetwithIterator
{
    public static void main(String args[])
    {
        // create a HashSet
        HashSet<String> hset1 = new HashSet<String>();
        // add elements to the hash set
        hset1.add("Java");
        hset1.add("C++");
        hset1.add("Python");
        hset1.add("R");
        hset1.add("Matlab");
        hset1.add("Scala");
        // traverse the set using Iterator and print each element one by one
        Iterator<String> itr = hset1.iterator();
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
    }
}
```

#### **Output**

```
Java
C++
R
Scala
Python
Matlab
```

#### **LinkedHashSet class**

A LinkedHashSet is an ordered form of HashSet that keeps a doubly-linked list of the elements in the Set. The sequence of insertion is preserved in LinkedHashSet and while iterating through LinkedHashSet, the elements are returned in the order of insertion. This is an advantage over HashSet. The hash code is used for indexing in the same way as in HashSet.

Constructors for LinkedHashSet class are given below,

Given below are the list of constructors supported by the LinkedHashSet:

- ... LinkedHashSet(): Constructs a default LinkedHashSet which is empty.
- ... LinkedHashSet(Collection Coll): This constructor would Initialize the LinkedHashSet with the elements of collection Coll.
- ... LinkedHashSet(int cap): This constructor would initialize the HashSet with capacity of the integer value of cap parameter.

- ... `LinkedHashSet(int capacity, float fillRatio)`: This particular constructor would initialize both the capacity and the fill ratio of the `LinkedHashSet`. The permissible value of fill ratio is between 0.0 and 1.0, and it governs how full the `HashSet` can be before it is resized upwards. If the number of elements in `LinkedHashSet` is greater than the capacity of the `LinkedHashSet` multiplied by its fill ratio, then it is expanded.

For the first three constructor functions, the value of the fill ratio is 0.75.

All methods of Collection interface and Iterable interface are available to `LinkedHashSet` class.

In order to instantiate an object of `LinkedHashSet` class, we have to mention the data type of its elements. For example:-

```
LinkedHashSet<datatype> Lhset = new LinkedHashSet<datatype>();
```

Here, `datatype` is the type of elements that can be stored in `Lhset`.

Let's see a program based on `LinkedHashSet` class.

```
import java.util.*;
class LinkedHSet
{
    public static void main(String args[])
    {
        // create a LinkedHashSet
        LinkedHashSet<String> Lhset = new LinkedHashSet<String>();
        // add elements to the LinkedHashSet
        Lhset.add("Operating System");
        Lhset.add("Java Programming");
        Lhset.add("Data Structures");
        Lhset.add("Database Management System");
        Lhset.add("Computer Networks");
        Lhset.add("Computer Architecture");
        // traverse the set using Iterator and print each element one by one
        Iterator<String> itr = Lhset.iterator();
        System.out.println("Original contents of LinkedHashset");
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
        //Remove an element from the LinkedHashSet
        Lhset.remove("Data Structures");
        // traverse the set again
        System.out.println("Elements of LinkedHashset after removing 'Data Structures' ");
        System.out.println(Lhset);
    }
}
```

## Output

```

Original contents of LinkedHashSet
Operating System
Java Programming
Data Structures
Database Management System
Computer Networks
Computer Architecture
Elements of LinkedHashSet after removing 'Data Structures'
[Operating System, Java Programming, Database Management System, Computer Neti
Computer Architecture]

```

Students can notice here that the elements iterate in the order of their insertion.

#### 4.3.2 Interface that extends Set interface

##### Sorted Set Interface implemented by TreeSet class

This interface extends Set interface and affirms that the elements of set are sorted in increasing order. It defines first() and last() methods to quickly return the first and last element respectively. This interface is implemented by **TreeSet class**. The important features of TreeSet class are given below: -

- ... It uses a tree for storage of the elements.
- ... Elements can be accessed and retrieved very fast.
- ... TreeSet class does not allow null element to be stored.
- ... As TreeSet class extends Sorted Set interface, automatically it maintains ascending order.
- ... Although elements are stored in increasing order, we can pass through in decreasing sequence also by using method TreeSet.descendingIterator().

The constructors supported by the TreeSet class are given below: -

- ... TreeSet(): Constructs a default TreeSet which is empty.
- ... TreeSet(Collection C): Initializes the TreeSet with the elements of collection c.
- ... TreeSet(Comparator cmp): Creates an empty TreeSet that will be sorted as per the value of comparator specified by cmp..
- ... TreeSet(SortedSet sset): Creates a treeset that comprises of members of SortedSet sset.

You can instantiate an object of TreeSet class in one of the following ways.

```
TreeSet<datatype> tset = new TreeSet<datatype>();
```

Here, datatype is the type of elements that are allowed to be stored in tset.

Another way to instantiate an object of TreeSet is by using SortedSet interface, as given below: -

```
SortedSet<datatype> tset = new TreeSet<datatype>();
```

Let's see a program based on TreeSet class.

```

import java.util.*;
class TreeSetProg
{
    public static void main(String args[])
    {
        // create a TreeSet
        TreeSet<String> tset1 = new TreeSet<String>();
        // add elements to the TreeSet
        tset1.add("India");
    }
}

```

```

tset1.add("Australia");
tset1.add("USA");
tset1.add("Canada");
tset1.add("Japan");
// traverse the set using Iterator and print each element one by one
Iterator<String> itr = tset1.iterator();
System.out.println("Original contents of TreeSet, by default in ascending order");
while(itr.hasNext())
{
    System.out.println(itr.next());
}
//Print the last element
System.out.print("The last element is ");
System.out.println(tset1.last());
// traverse the treeset again but now in descending order
System.out.println("Contents of TreeSet in descending order are...");
Iterator<String> desc_itr= tset1.descendingIterator();
while(desc_itr.hasNext())
{
    System.out.println(desc_itr.next());
}
}
}
}

```

### Output

Original contents of TreeSet, by default in ascending order  
 Australia  
 Canada  
 India  
 Japan  
 USA  
 The last element is USA  
 Contents of TreeSet in descending order are...  
 USA  
 Japan  
 India  
 Canada  
 Australia

### ➤ Check Your Progress-2

- 1) What do you understand by Iterator in the Java Collection Framework?

.....  
 .....  
 .....  
 .....

- 2) What is the HashSet class in Java, and how does it store elements?

.....  
 .....  
 .....  
 .....

- 3) Differentiate between TreeSet and HashSet?

.....  
.....  
.....  
.....

- 4) What is the initial fill ratio of HashSet?

- a) 1.5
- b) 0.5
- c) 0.75
- d) 1.0

- 5) What is the output of the following program

```
import java.util.*;  
class test1  
{  
    public static void main(String[] args)  
    {  
        HashSet<String> set=new HashSet<String>();  
        set.add(null);  
        set.add("One");  
        for (String s: set)  
            System.out.println(s);  
    }  
}
```

---

## 4.4 MAP INTERFACE

---

Map interface is a part of `java.util` package but map is not a collection as it does not implement the collection interface. Map interface represents a mapping between a key and a value. A map contains unique keys though the values for two or more keys can be same. Each key can map to at the most one value only. A Map is suitable whenever we want to search, update or delete items on the basis of a key. An example of map is zip code (key) and city (value). The hierarchy of classes and interfaces of Map interface is shown in Figure 2. The methods to perform various operations on maps are given in Table 5.

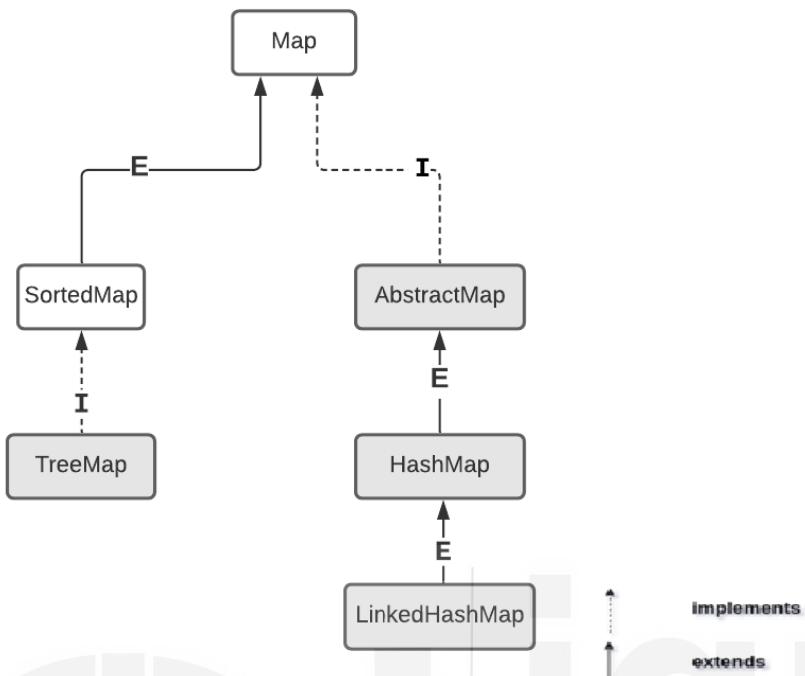


Figure 2 Map interface of Collections Framework

Table 5: Description of methods of Map interface

S.No.	Method	Description
1	void clear()	Deletes all key/value pairs from the invoking map.
2	boolean containsKey(Object k)	Gives true if the map object that invoked this method has k as one of the keys, else it gives false.
3	Boolean containsValue(Object obj)	It gives true if the map object that invoked this method has obj as one of the values, else its gives false.
4	Set entrySet( )	This method gives a Set that comprises of all the elements of the map. The set has in it all objects of type Map.Entry. We can use this method to provide a Set-view of the invoking map.
5	boolean equals(Object ob)	It gives true if ob is a Map and has the same entries as invoking Map. Else, it gives false.
6	Object get(Object obj)	It gives the value corresponding to the key obj.

7	int hashCode( )	It gives the hash code of the map that invoked this method.
8	boolean isEmpty( )	If the object that invoked this method does not have any element in it, then it gives true, else false.
9	Set keySet( )	This method produces a Set that has all the keys of the invoking map. It is used to provide a set-view of only the keys present in the invoking map.
10	Object put(Object k, Object v)	It adds an element in the invoking map. If there is already an element in the map having key same as k, then the value of that key is overwritten. Here, k corresponds to the key and v corresponds to the value. This method gives null if the key was not present already in the map. If a key was present, then the previous value linked to that key is returned.
11	void putAll(Map mp)	Adds all the elements of map mp into the invoking map.
12	Object remove(Object k)	Deletes the entry of map whose key is equal to k.
13	int size( )	It gives the number of key/value pairs that are present in the invoking map.
14	Collection values()	This method gives a collection-view of only the values of the map.

## 4.5 HASHMAP CLASS

HashMap class implements the Map interface. A hash table is used by HashMap in order to implement Map interface. The advantage of hash table is that the time for basic operations like get() and put() remains constant irrespective of size of Map. Given below are the constructors for instantiating an object of HashMap.

- ... HashMap( ) : It instantiates a default hash map.
- ... HashMap(Map mp) : It initializes the hash map with the elements of m.
- ... HashMap(int cap) : It constructs a hash map with a capacity equal to argument cap.
- ... HashMap(int cap, float fillR): This constructor uses the arguments cap and fillR to initialize both the capacity and fill ratio. The gist of capacity and fill ratio remains the same as for HashSet, discussed earlier.

You should remember that there is no guarantee of the order of elements in the case of HashMap. So, the order in which elements are inserted into a hash map may not match the order of their retrieval by the iterator.

You can instantiate an object of HashMap class by using one of the given below techniques.

1. Non-Generic way (Old Style)

```
Map mp=new HashMap(); or HashMap mp=new HashMap();
```

Anything involving HashMap or Map without a datatype argument (the angle brackets < and > and the part between them) is a raw type that is an older approach and shouldn't be used. A raw type is not generic and lets you do hazardous things.

2. Generic way (New Style)

```
Map<T1,T2> mp1=new HashMap<T1,T2>();
```

T1, T2 – The generic type parameter passed to the generic interface Map and its implementation class HashMap, during map declaration.

Generics declaration allows compiler to check the 'mp1' usages during compile-time and it is very safe.

We will be using the Generic way in the example given below: -

```
import java.util.*;
class HashMapEx
{
    public static void main(String args[])
    {
        Map<Integer,String> map1=new HashMap<Integer,String>();
        map1.put(1,"Anita");
        map1.put(2,"Charu");
        map1.put(3,"Geeta");
        System.out.println();
        System.out.print("Map Elements :- " +map1);
        System.out.println();
        //For traversing Map, convert it into Set
        Set set=map1.entrySet(); //Converting to Set so that we can traverse
        Iterator itr1=set.iterator();
        System.out.println("Map elements by traversing the map after being converted to set");
        while(itr1.hasNext())
        {
            //Converting to Map.Entry so that we can get key and value separately
            Map.Entry entry=(Map.Entry)itr1.next();
            System.out.println(entry.getKey()+" "+entry.getValue());
        }
        System.out.println("Number of elements in map is "+map1.size()+" elements.");
        boolean b=map1.containsKey(1);
        System.out.println("It is "+b+" that the map contains an element with key 1");
        //Remove the entry whose key is 2
        map1.remove(2);
        System.out.println("Contents of map after deleting element with key 2 are "+map1);
        if (map1.isEmpty())
        System.out.println("Map is empty");
        else
        System.out.println("Map is not empty");
```

```
//Delete all elements of map  
map1.clear();  
System.out.println("All elements cleared");  
if (map1.isEmpty())  
    System.out.println("Now Map is empty");  
else  
    System.out.println("Now Map is not empty");  
}  
}
```

## Output

```
Map Elements :-{1=Anita, 2=Charu, 3=Geeta}  
Map elements by traversing the map after being converted to set  
1 Anita  
2 Charu  
3 Geeta  
Number of elements in map is 3 elements.  
It is true that the map contains an element with key 1  
Contents of map after deleting element with key 2 are {1=Anita, 3=Geeta}  
Map is not empty  
All elements cleared  
Now Map is empty
```

Given below is a program that uses HashMap for storing employee ids with their names. This program is menu-driven and takes input from the user, and accordingly uses an appropriate method to display the result.

```
import java.util.*;  
class HashMapEx1  
{  
    public static void main(String args[])  
    {  
        Scanner sc = new Scanner(System.in);  
        Map<Integer, String> map1 = new HashMap<Integer, String>();  
        map1.put(401, "Ramesh");  
        map1.put(205, "Balram");  
        map1.put(156, "Vinod");  
        map1.put(555, "Geeta");  
        System.out.println();  
        int choice = 1;  
        do  
        {  
            System.out.println("Main Menu");  
            System.out.println("1: Size of Map");  
            System.out.println("2: Remove all elements of Map");  
            System.out.println("3: Check if hashmap is empty?");  
            System.out.println("4: Search for an employee");  
            System.out.println("5: Input Employee ID and display name of that employee");  
            System.out.println("6: Display names of all employees");  
            System.out.println("7: Display employee id with their names");  
            System.out.println("8: Exit");  
            System.out.println("Enter choice: ");  
            choice = sc.nextInt();  
            switch (choice)  
            {  
                case 1: System.out.println("Size of the hash map: " + map1.size() + " records");  
                break;
```

```

case 2: map1.clear();
        System.out.println("The New map: " + map1);
        break;
case 3: boolean result = map1.isEmpty();
        System.out.println("It is " + result + " that hash map is empty");
        break;
case 4: System.out.print("Enter employee id to be searched.. ");
        int eid=sc.nextInt();
        if (map1.containsKey(eid))
            System.out.println("yes! " + eid + " is present.");
        else
            System.out.println("no! " + eid + " is not there.");
        break;
case 5: System.out.print("Enter employee id whose name is required.. ");
        eid=sc.nextInt();
        if (map1.containsKey(eid))
        {
            System.out.println("yes! " + eid + " is present.");
            String val=(String)map1.get(eid);
            System.out.println("Name for " + eid + " is " + val);
        }
        else
            System.out.println("no! " + eid + " is not there.");
        break;
case 6: System.out.println("Names of all employees: "+ map1.values());
        break;
case 7: Set set = map1.entrySet();
        System.out.println("Employee ID with names: " + set);
        break;
case 8: System.out.println("Byeeee....");
        break;
default: System.out.println("Invalid choice");
        break;
    }
}
while(choice!=8);
}
}

```

## Output

```

Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee
6: Display names of all employees
7: Display employee id with their names
8: Exit
Enter choice:
1
Size of the hash map: 4 records
Main Menu
1: Size of Map
2: Remove all elements of Map
3: Check if hashmap is empty?
4: Search for an employee
5: Input Employee ID and display name of that employee
6: Display names of all employees
7: Display employee id with their names
8: Exit

```

```
Enter choice:  
3  
It is false that hash map is empty  
Main Menu  
1: Size of Map  
2: Remove all elements of Map  
3: Check if hashmap is empty?  
4: Search for an employee  
5: Input Employee ID and display name of that employee  
6: Display names of all employees  
7: Display employee id with their names  
8: Exit  
Enter choice:  
5  
Enter employee id whose name is required.. 6  
no! 6 is not there.  
Main Menu  
1: Size of Map  
2: Remove all elements of Map  
3: Check if hashmap is empty?  
4: Search for an employee  
5: Input Employee ID and display name of that employee
```

```
Enter choice:  
6  
Names of all employees: [Ramesh, Geeta, Vinod, Balram]  
Main Menu  
1: Size of Map  
2: Remove all elements of Map  
3: Check if hashmap is empty?  
4: Search for an employee  
5: Input Employee ID and display name of that employee  
6: Display names of all employees  
7: Display employee id with their names  
8: Exit  
Enter choice:  
8  
Byeeee....
```

### ➤ Check Your Progress-3

- 1) Differentiate between Set and Map.

.....  
.....  
.....  
.....

- 2) Why do we use Map interface? What are the main classes implementing Map interface?

.....  
.....  
.....  
.....

- 3) What is the difference between HashSet and HashMap?
- .....  
.....  
.....  
.....

- 4) Is hashmap an ordered collection.

- a) True
- b) False

- 5) Write down the output produced by the following program

```
import java.util.*;
public class test2
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Ram", 20);
        map.put("Shyam", 25);
        map.put("Harish", 22);
        System.out.println("Size of map is:- " + map.size());
        System.out.println(map);
        if (map.containsKey("Shyam"))
        {
            Integer a = map.get("Shyam");
            System.out.println("value for key" + " \"Shyam\" is:- " + a);
        }
    }
}
```

## 4.6 LIST INTERFACE

List interface extends the Collection interface, and hence it comprises all the methods available in Collection interface. It is used to manage the ordered collection of objects. Duplicate data is also allowed in list. Elements can be accessed by referring to their position in the list, with zero-based index. Various methods defined by list interface are mentioned in Table 6. The list interface is implemented by Vector, ArrayList and LinkedList classes. The object of list can be instantiated by any of the following approaches: -

```
List <T> aList = new ArrayList<>();
List <T> lList = new LinkedList<>();
List <T> v = new Vector<>();
```

Where T is a generic type parameter passed to the generic interface List.

Table 6: Description of methods of List Interface

Sr.No.	Method	Description
1	void add (int idx, Object ob)	Adds ob into the invoking list at the index mentioned in the idx. None of the elements is lost by overwriting because any pre-existing element at or after the position of insertion are shifted up.
2	boolean addAll (int idx, Collection col)	Adds all elements of col in the list at the index passed in the idx. None of the elements is lost by overwriting because any pre-existing element at or after the point of insertion are moved up. This method gives true if the invoking list modifies otherwise, it gives false.
3	Object get(int idx)	It gives the object available at the mentioned idx of the invoking list.
4	int indexOf(Object ob)	It gives the index of the first occurrence of ob in the invoking list. If ob is not present in the list, it returns -1.
5	int lastIndexOf(Object ob)	It gives the index of the last occurrence of ob in the invoking list. If ob is not found in the list, then it returns -1
6	ListIterator iterator( )	It gives an iterator to the beginning of the invoking list.
7	ListIterator iterator (int index)	It gives an iterator that begins at the specified index of the invoking list.
8	Object remove(int idx)	It deletes the element at position idx from the invoking list and gives that deleted element. This makes the resulting list compact, and hence the total number of elements is decreased by one.
9	Object set (int idx, Object ob)	Allocates ob to the location specified by idx within the list that invoked this method.
10	List subList (int begin, int end)	It gives a list that includes elements from begin to (end-1) from the invoking list.

## 4.7 VECTOR CLASS

The Vector class extends AbstractList class. The List interface is implemented by this class. Vector implements a dynamic array that can grow or shrink as per the requirement. The elements of vector can be retrieved with an integer index. Vector is quite alike ArrayList, but it is synchronized i.e. it is thread-safe. The Vector class should be used in the thread-safe implementation only, and if you don't require synchronous access then ArrayList should be used as it will perform better.

Given below are the Constructors for Vector class:

`Vector()`: This constructs a default vector with the original capacity to store ten elements.

`Vector(int s)`: A vector is constructed that has initial capacity equal to argument s.

`Vector(int s, int increm)`: This constructor initializes both the size and increment of vector by using its arguments. The meaning of increm is the number of elements to assign each time when a vector is resized upward. (If the argument increm is not mentioned then vector's capacity will be doubled in each allocation round).

`Vector(Collection col)`: This particular constructor initializes the vector with the elements of collection col.

Vector defines the following data members that have protected visibility mode:

`int capacityIncrement`: Stores the value of the increment.

`int elementCount`: Contains the count of elements presently in vector.

`Object elementData[]`: An array that stores the elements of the vector.

Vector class has all the methods inherited from its super classes, and it also defines additional methods explained in Table 7.

Table 7: Description of methods of Vector class

S.No.	Method	Description
1	<code>void add (int idx, Object obj)</code>	Adds the specified object at the mentioned idx in the Vector. This method is not synchronous.
2	<code>boolean add (Object obj)</code>	Returns true after appending the mentioned object to the end of the Vector.
3	<code>boolean addAll(Collection col)</code>	Appends all elements of the mentioned Collection to the end of the Vector, in the same sequence as they are returned by the iterator of the mentioned Collection. Returns true after appending.
4	<code>boolean addAll (int idx, Collection c)</code>	Inserts each and every element of the mentioned Collection into the Vector at the mentioned index. Returns true after inserting element.
5	<code>void addElement (Object ob)</code>	Adds the mentioned object to the end of the invoking vector, and increases its size by one.

		addElement() is synchronized.
6	int capacity()	The current capacity of the invoking vector is returned.
7	void clear()	All elements from the vector are removed.
8	Object clone()	A clone of the vector is returned.
9	boolean contains(Object obj)	Returns true if the mentioned object is a member in the invoking vector.
10	boolean containsAll(Collection c)	If the invoking vector contains all the elements of the mentioned Collection, then this method returns true.
11	void copyInto (Object[] anArray)	All the components of the invoking vector are copied into the mentioned array.
12	Object elementAt(int idx)	The component at the mentioned index is returned.
13	Enumeration elements()	An enumeration of all components of the invoking vector is returned.
14	void ensureCapacity (int minCapacity)	This method increases the capacity of the invoking vector, if required, so that it can store no less than the number of components mentioned by minCapacity.
15	boolean equals (Object ob)	If the invoking vector is equal to Object, then true is returned.
16	Object firstElement()	It gives the first element (the object at index 0) of the invoking vector.
17	Object get(int idx)	It gives the element at the mentioned index of the vector.
18	int hashCode()	The hash code for the invoking vector is returned.
19	int indexOf(Object obj)	The position of first occurrence of passed object is searched and returned
20	int indexOf (Object obj, int idx)	The first occurrence of passed object is searched but the search begins at idx.
21	void insertElementAt (Object ob, int idx)	The passed object is inserted (as a component) in the vector at the mentioned index.
22	boolean isEmpty()	Returns true if the vector does not have any components.
23	Object lastElement()	The last element of the invoking vector is

		returned.
24	int lastIndexOf (Object obj)	The index of the last occurrence of object obj, in the vector is returned.
25	int lastIndexOf (Object obj, int index)	Searches towards the back for passed object, beginning at the mentioned index, and gives the position of it.
26	Object remove(int idx)	Deletes and returns the element at the mentioned index of the vector.
27	boolean remove (Object obj)	Deletes the first occurrence of the mentioned object in the invoking vector and returns true. If the element is not found in the vector, it returns false.
28	boolean removeAll (Collection c)	Eliminates all elements of the specified collection from the vector and returns true. The method returns false, if the collection is not found.
29	void removeAllElements()	All elements of the vector are removed and its size is set to zero.
30	boolean removeElement (Object obj)	The first occurrence of the passed object is removed from the vector.
31	void removeElementAt (int idx)	The element at the mentioned index is removed from the vector.
32	protected void removeRange (int fmIndex, int toIndex)	All elements whose index lies between fmIndex, inclusive and toIndex, exclusive, are removed from the vector.
33	boolean retainAll(Collection c)	Only the elements that are contained in the mentioned Collection are retained in the invoking vector.
34	Object set(int index, Object element)	Element at the specified index in the invoking vector is replaced with the mentioned element.
35	void setElementAt(Object obj, int idx)	The object at the specified index (idx) of the vector is set to the object obj.
36	void setSize(int newSize)	The size of the vector is set to the newSize.
37	int size()	It gives the number of elements in the invoking vector.
38	List subList (int fromIndex, int toIndex)	A view of the slice of the List between fromIndex, inclusive, and toIndex, exclusive is returned.

39	Object[] toArray()	An array containing all the components of the vector in the exact order is returned.
40	Object[] toArray(Object[] a)	An array comprising all of the elements in this vector in the correct order is returned. The runtime type of the returned array is same as that of the specified array.
41	String toString()	A string representation of the vector is returned.
42	void trimToSize()	The capacity of the vector is trimmed to its current size.

You can instantiate an object of HashMap class in one of the following ways.

1. Non-Generic way (Old Style)

Vector v=new Vector(); or List v=new Vector();

Anything involving HashMap or Map without a datatype argument (the angle brackets < and > and the part between them) is a raw type that is an older approach and shouldn't be used. A raw type is not generic and lets you do hazardous things.

2. Generic way (New Style)

Vector<T> v=new Vector<T>();

T – The generic type parameter passed to the generic interface List and its implementation class Vector.

Generic declaration allows compiler to check the 'v' usages during compile-time and it is very safe.

We will be using the Generic way in the example given below: -

We will be using the Generic way in the example given below: -

```
import java.util.*;
public class VectorEx
{
    public static void main(String args[])
    {
        //Create an empty vector
        Vector<String> vec1 = new Vector<String>();
        //Add elements to the vector
        vec1.add("Television");
        vec1.add("Refrigerator");
        vec1.add("Washing Machine");
        //display size and capacity
        System.out.println("Size is: "+vec1.size());
        System.out.println("Default capacity is: "+vec1.capacity());
        //Display Vector elements
        System.out.println("Vector element is: "+vec1);
        //Add two more elements
        vec1.add("Microwave Oven");
        vec1.add("Mobile Phone");
        //Again display size and capacity after two insertions
        System.out.println("Size after addition: "+vec1.size());
        System.out.println("Capacity after addition is: "+vec1.capacity());
        //Display Vector elements again
        System.out.println("Elements are: "+vec1);
        //Checking if Television is present or not in this vector
    }
}
```

```

if(vec1.contains("Television"))
{
    System.out.println("Television is present at the index " +vec1.indexOf("Television"));
}
else
{
    System.out.println("Television is not present in the list.");
}
//Print the first element
System.out.println("The first element of the vector is = "+vec1.firstElement());
//Print the last element
System.out.println("The last element of the vector is = "+vec1.lastElement());
//Add Refrigerator again
vec1.add("Refrigerator");
//Display Vector elements again
System.out.println("Elements are: "+vec1);
//use remove() method to delete the first occurrence of an element
System.out.println("Remove first occurrence of element Refrigerator:
"+vec1.remove("Refrigerator"));
//Display the vector elements after remove() method
System.out.println("Values in vector: " +vec1);
//Remove the element at index 4
System.out.println("Remove element at index 3: " +vec1.remove(3));
System.out.println("New Values in vector: " +vec1);
//Get the hashCode for this vector
System.out.println("Hash code of this vector = "+vec1.hashCode());
//Get the element at specified index
System.out.println("Element at index 1 is = "+vec1.get(1));
}
}

```

### Output

```

Size is: 3
Default capacity is: 10
Vector element is: [Television, Refrigerator, Washing Machine]
Size after addition: 5
Capacity after addition is: 10
Elements are: [Television, Refrigerator, Washing Machine, Microwave Oven, Mobile Phone]
Television is present at the index 0
The first element of the vector is = Television
The last element of the vector is = Mobile Phone
Elements are: [Television, Refrigerator, Washing Machine, Microwave Oven, Mobile Phone, Refrigerator]

Remove first occurrence of element Refrigerator: true
Values in vector: [Television, Washing Machine, Microwave Oven, Mobile Phone, Refrigerator]
Remove element at index 3: Mobile Phone
New Values in vector: [Television, Washing Machine, Microwave Oven, Refrigerator]
Hash code of this vector = 1730521510
Element at index 1 is = Washing Machine

```

---

## 4.8 STACK CLASS

---

The stack is a linear data structure that extends the Vector class. It is based on Last-In-First-Out (LIFO). Only one constructor is defined by the Stack class, which creates an empty stack.

`Stack<T> stk = new Stack<T>();`

T – The generic type parameter passed to the Stack.

All methods defined by Vector class are available in Stack in addition to few of its own methods. These added methods are given in Table 8.

Table 8: Description of methods of Stack class

S.No.	Method	Description
1	boolean empty()	If stack is empty, true is returned otherwise false is returned.
2	Object peek()	The element at the top of the stack is returned without removing it from stack.
3	Object pop()	The element at the top of the stack is returned as well as removed from stack.
4	Object push(Object element)	The element is inserted onto the stack.
5	int search(Object element)	The mentioned element is searched in the stack. If found, its offset from the top of the stack is returned otherwise, -1 is returned.

Given program demonstrates some of the operations on stack.

```

import java.util.Stack;
import java.util.Scanner;
public class StackExample
{
    public static void main(String[] args)
    {
        int j;
        Scanner in = new Scanner(System.in);
        //creating an instance of Stack class
        Stack<Integer> stk= new Stack<>();
        // checking stack is empty or not
        boolean result = stk.empty();
        System.out.println("It is "+result+" stack is empty");
        // pushing five elements into stack
        for (int i=1;i<6;i++)
        {
            System.out.println("Enter element number "+i+" to be pushed");
            j = in.nextInt();
            stk.push(j);
        }
        System.out.println("Elements in Stack: " + stk);
        System.out.println("Enter the element to be searched...");
        j = in.nextInt();
        Integer pos = (Integer) stk.search(j);
        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position: " + pos);
    }
}

```

### Output

```

It is true stack is empty
Enter element number 1 to be pushed
56
Enter element number 2 to be pushed
43
Enter element number 3 to be pushed
98
Enter element number 4 to be pushed
73
Enter element number 5 to be pushed
122
Elements in Stack: [56, 43, 98, 73, 122]
Enter the element to be searched...
73
Element is found at position: 2

```

#### ☛ Check Your Progress-4

- 1) Write down the differences and similarities between List and Set?

.....  
 .....  
 .....

- 2) Why do we use the List interface? Name the classes that implement the List interface.

.....  
 .....  
 .....

- 3) What is the output of the following program?

```

import java.util.*;
public class test3
{
    public static void main(String[] args)
    {
        List<Integer> list1 = new ArrayList<Integer>();
        list1.add(0, 1);
        list1.add(1, 2);
        System.out.println(list1);
        List<Integer> list2 = new ArrayList<Integer>();
        list2.add(1);
        list2.add(2);
        list2.add(3);
        list1.addAll(1, list2);
        System.out.println(list1);
        list1.remove(1);
        System.out.println(list1);
        System.out.println(list1.get(3));
        list1.set(0, 5);
    }
}

```

```
        System.out.println(list1);
    }
}
```

- 4) Explain the Vector class and its importance.

.....  
.....  
.....  
.....

- 5) What do you mean by the Stack class, and what are the various methods provided by it?

.....  
.....  
.....  
.....

---

## 4.9 SUMMARY

---

This unit explains various classes and interfaces that are part of java.util package. We have deliberated the functions of date and time classes along with suitable program codes. This unit deals with List and Set that are important interfaces of collection framework. Different classes that implement these interfaces have been elaborated with various programs. We have discussed Map interface along with HashMap class that implements it. Overall, the facilities offered by Collection framework have been explained in detail.

---

## 4.10 SOLUTIONS/ANSWERS

---

### ➤ Check Your Progress-1

- 1) Collection Framework is a group of classes and interfaces to store and manipulate the group of objects. Various operations like searching, sorting, insertion and deletion can be performed on these objects. Various classes such as ArrayList, Vector, Stack, and HashSet, etc. are provided by the collection framework. Also, there are many interfaces such as List, Queue, Set, etc. available in collection framework.
- 2) Maps is not a part of the collection framework.
- 3) In order to check whether two date objects are equal, the following method is used.

int compareTo (Date date) : The value of the invoking object is compared with that of date object. If both are equal, the function returns 0. A negative

value is returned, if the invoking object is earlier than date. Or a positive value is returned, if the invoking object is later than the date.

### ☛ Check Your Progress-2

- 1) In order to pass over the elements of any collection, we can make use of Iterable interface, which is the root interface of the whole collection framework. The Iterable interface is extended by the collection interface. Hence, all the interfaces and classes of collection framework implement this interface. The most important feature of Iterable interface is to offer an iterator for the collections. Only one method, which is abstract and whose name is iterator(), is contained in this interface. It returns an object of type Iterator.

```
Iterator iterator();
```

- 2) java.util.HashSet class is a member of the Java collections framework, which implements the Set interface. Its purpose is to create a collection and store it in a hash table. Each element in Set has a unique value of its own, called hash code which is used as an index at which element associated with that hash code is stored. Hash code is obtained by converting the informational content into a unique value (known as hash code). Hashing ensures that the execution time of operations like add(), remove(), remain constant even for set of bigger size. The elements are inserted in HashSet irrespective of the order of insertion, and the order depends on the hash code of elements.
- 3) The HashSet and TreeSet, both classes, implement Set interface. The differences between the both are listed below: -

HashSet	TreeSet
It does not maintain any sequence of elements added to it.	It maintains ascending order of elements.
It is implemented by hash table.	It is implemented by a Tree structure.
Performance is better than TreeSet.	TreeSet performance is not as good as HashSet.
HashSet is backed by HashMap.	TreeSet is backed by TreeMap.

- 4) 0.75
- 5) null
- One

### ☛ Check Your Progress-3

- 1) The differences between Set and Map are given below: -
  - ... Set contains only the values, whereas Map contains key and values both.
  - ... Set has unique values, whereas Map can contain unique Keys with duplicate values (Keys are unique but two or more keys can have the same value.)

... Set can include just one null, whereas Map can include a single null key with n number of null values.

- 2) Map interface is a part of java.util package, but map is not a collection as it does not implement the collection interface. Map interface represents a mapping between a key and a value. A map contains unique keys though the values for two or more keys can be the same. At the most, one value can be mapped by each key. A Map is suitable if we have to search, update or delete elements on the basis of a key. The classes that implement Map interface are: HashMap, Hashtable, LinkedHashMap.
- 3) The differences between the HashSet and HashMap are listed below.

HashSet	HashMap
Set interface is implemented by Hashset.	Map interface is implemented by the HashMap.
It contains only values.	It contains a mapping between a key and its value.
It can be traversed.	HashMap needs to be converted into Set to be iterated.
It cannot have any duplicate value.	It can have duplicate values with unique keys.
It can hold just one null value.	It can contain a single null key with n number of null values.

- 4) False. Hashmap outputs in the order of hashCode of the keys. Though actually it is unordered but will always have the same result for the same set of keys.
- 5)

```
Size of map is:- 3
{Shyam=25, Harish=22, Ram=20}
value for key "Shyam" is:- 25
```

#### ☛ Check Your Progress-4

- 1) The List and Set both extend the collection interface. However, there are some differences between the both, as mentioned below :-

List	Set
Duplicate elements are allowed in List.	Only unique elements are allowed in Set.
It is an ordered collection that maintains the order of insertion of elements.	It is an unordered collection that does not preserve the order of insertion.
The List interface can allow any number of null values.	The Set interface allows just a single null value.

- 2) The List interface is an ordered collection of elements. List preserves the insertion order and allows duplicate values to be stored. This interface comprises different methods that enable easy manipulation of elements based on the element index. The main classes that implement List interface are ArrayList, LinkedList, Stack, and Vector.

3)

```
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
[5, 2, 3, 2]
```

- 3) An index is used to access the elements of a vector. Vector is just like a dynamic array. There is no specific size of a vector; it can shrink or grow automatically whenever required. It is quite similar to ArrayList, but there are two differences as given below: -

- ... Vector is synchronized.
- ... Many legacy methods are available in Vector class that are not part of the collections framework.

- 5) Last-in-first-out is the basic principle of Stack class. The elements are added as well as removed from the rear end (that is where the latest item was inserted or removed from). The action of adding an element to a stack is called push, while removing an element is referred to as pop. In order to retrieve or fetch the top element of the Stack, we can use peek() method. The element retrieved is only accessed, not deleted or removed from the Stack.

## 4.11 REFERENCE/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.
- ... E Balagurusamy , “Programming with Java”, McGraw-Hill Education,2019.
- ... Benjamin J. Evans, David Flanagan , “Java in a Nutshell: A Desktop Quick Reference”, O'Reilly,2019.
- ... <https://www.javatpoint.com/collections-in-java>
- ... [https://www.tutorialspoint.com/java/java\\_collections.htm](https://www.tutorialspoint.com/java/java_collections.htm)
- ... <https://www.geeksforgeeks.org/collections-in-java-2/>
- ... <https://www.programiz.com/java-programming/collections>
- ... [https://en.wikipedia.org/wiki/Java\\_collections\\_framework](https://en.wikipedia.org/wiki/Java_collections_framework)
- ... <https://dzone.com/articles/an-introduction-to-the-java-collections-framework>

---

# **UNIT 1 INTRODUCTION TO GUI IN JAVA**

---

## **Structure**

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Introduction to AWT, Swing and JavaFX
  - 1.2.1 Abstract Window Toolkit (AWT)
  - 1.2.2 Swing
  - 1.2.3. JavaFX
- 1.3 Features of JavaFX
- 1.4 User Interface Components of JavaFX
- 1.5 Work with Layouts
- 1.6 Add HTML Content
- 1.7 Add Text and Text Effects in JavaFX
- 1.8 Summary
- 1.9 Solutions/ Answer to Check Your Progress
- 1.10 References/Further Reading

---

## **1.0 INTRODUCTION**

---

Graphical User Interface (GUI) is a kind of interface through which users can interact with computer applications. Be it web applications, mobile apps, or other commonly used applications like online banking or the online railway reservation system, we look for ease of interaction by using proper buttons, menus, and other controls. These interfaces use proper GUI components and controls. Earlier engineers or programmers used to interact with the computer through the DOS prompt/interface using command line. This interface is also used to be called as the character user interface(CUI), which was very inconvenient for non-programming users. With the development of the Graphical user interface, the use of computing devices by non-technical users or ordinary users has been more convenient because GUI provides an interface that allows users to interact with electronic devices through graphical objects and controls to convey information and represent action taken by the users.

Java provides a very rich set of Graphical User interface (GUI) API for developing GUI programs. In fact, GUI is one of the most helpful features provided by Java Development Kit (JDK). Java provides various essential classes and libraries for the implementation of graphic classes to the programmers for constructing their own Graphical User Interface applications. The Graphic classes developed by the JDK developer team are highly complex and uses many advanced design patterns. However, for the programmers, the reuse of these classes and methods are very easy to use. Some commonly used APIS for Graphic User interface (GUI) programming are : Applet, AWT, Swing, JavaFX. In this unit, you will learn about the basics of AWT, Swing and JavaFX API and its implementation with examples.

JavaFX is an open-source programming platform for developing next-generation client applications to implement mobile apps, desktop systems and embedded systems built using Java . It is designed using the Model View Controller (MVC) design pattern to keep the code that handles an application's data, separate from the User Interface Code. To develop enterprise applications, generally, MVC design pattern is used, because it does not mix the UI code with the application logic code. The controller is the mediator between UI and Business logic (the data). Working with JavaFX, the model corresponds to an application's data model, the view is FXML. The FXML is XML-based language designed to create the user interface for JavaFX applications. The controller is the code that determines the action when the user

interacts with the UI. Mainly the controller handles all events in the application. The First LTS (long term support) version of JavaFX is the JavaFX 11 which was released by Gluon. It is recommended to use the current LTS version for applications development. In this unit JavaFX 11 is used for the demonstration of the examples.

## 1.1 OBJECTIVES

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

- explain GUI and different types of GUI API available in Java,
- apply the concept of JavaFX in programming,
- differentiate between JavaFX other GUI's APIs,
- describe important features of JavaFX,
- use components & layouts in JavaFX, and
- write a program using JavaFX.

## 1.2 INTRODUCTION TO AWT, SWING and JAVAFX

Java provides three main sets of Java APIs for Graphic User interface (GUI) programming languages

- AWT (Abstract Windowing Toolkit)
- Swing
- JavaFX

Let us see these Java APIs one by one, which are used for Graphical User Interface (GUI) programming in the Java family.

### 1.2.1 Abstract Window Toolkit (AWT)

AWT is an API that is used for developing window-based applications in Java. AWT was introduced with JDK 1.0, and it is used to create GUI objects like textboxes, labels, checkbox, buttons, scroll bars, and windows etc. AWT is part of the Java Foundation Classes (JFC) from Sun Microsystems, creator of Java programming language. The JFC contains a set of graphs libraries and classes used to develop the user interface of the Windows-based application program. AWT is one of Java's largest packages (Java.awt). Its hierarchy is logically organized in a top-down fashion. The hierarchy of the Java AWT package/class is shown in figure 1.

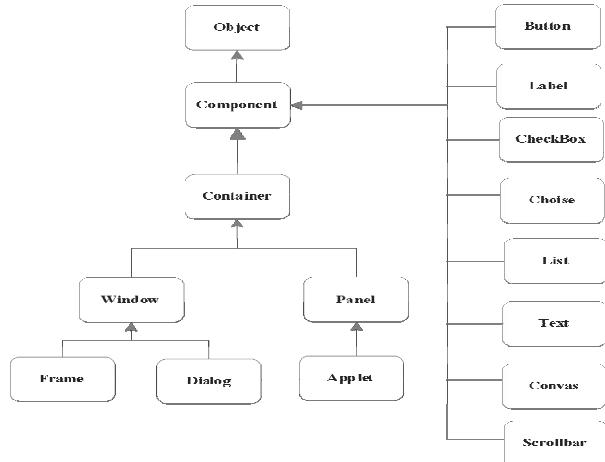


Figure 1: Hierarchy of Java AWT classes

## Component Class

As we can see in figure 1, the Component class is on the top of the AWT hierarchy. It is an abstract class that encapsulates all of the attributes of a visual component. It is platform-dependent i.e., its components are displayed according to the view of the operating system on which it runs. It is also heavyweight because its components uses the OS's resources during the execution of the program. User interface elements (i.e., textbox, label, list, checkbox, etc.) which are displayed on the screen and used to interact with the user are subclasses (such as Label, TextField, TextArea, RadioButton, CheckBox, List, Choice etc.) of Component class.

## Container Class

The container is a subclass of Component class of AWT. It provides the additional methods that allow other Component objects to call within the container object. Other Container objects can be stored inside a container because they are themselves instances of the Component class. This makes it a multilevelled containment system. A container is responsible for laying out or positioning the component's object using the various layout managers.

## Panel Class

The panel class is the subclass of the container class that implements the container class only. It does not provide any special new methods. It provides space to assemble all components, including other panels. Other components can be added within a Panel object by calling its add() method (which is inherited from Container). Once these components are assembled or added within the panel, you can set their position and resize them manually using the setLocation(), setSize(), setPreferredSize(), or setBounds() methods which are defined by the Component class.

## Windows Class

The Window is an area that is displayed on the screen when you execute an AWT program. It creates a top-level window that is not contained with any other object; it directly sits on the desktop screen. It provides a multitasking environment for users to interact with the system through the component shown on the screen. It must have a frame, dialog or another window defined by the programmer/owner when it is constructed in the program because window objects can not be created directly.

## Frame Class

The Frame is the subclass of Window. It is the top-level of windows that provides title bar, menu bar, borders and resizing option for window. It can also have other components like button, text field, scrollbar etc. It encapsulates the window and uses BorderLayout as the default layout manager. Frame is one of the most widely used containers for AWT applications.

Now let us see use of different awt components in a program.

### Example 1: Write a First programme using AWT in Java

```
package org.ignou.gui;

import Java.awt.*;
import Java.awt.event.WindowAdapter;
import Java.awt.event.WindowEvent;

public class AWTFirstExample extends WindowAdapter
{
    Frame myFrame;

    AWTFirstExample ()
    {
```

```
//Creating a frame
myFrame= new Frame();

//Add Windows Lister
myFrame.addWindowListener(this);
myFrame.setTitle("AWT Example");

//Creating a label
Label myLabel = new Label ("Welcome to First AWT Programme.");

//adding label to the frame
myFrame.add(myLabel);

//setting frame size.
myFrame.setSize(400, 400);

//set frame visibility true
myFrame.setVisible(true);
}

//setting Window close operation.
public void windowClosing(WindowEvent e)
{
    myFrame.dispose();
}

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

**Output:**

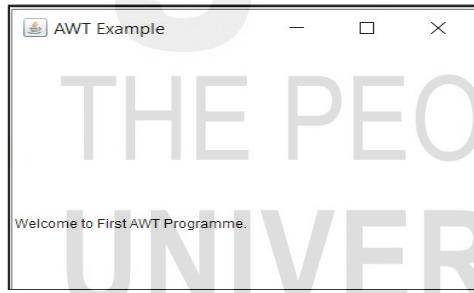


Figure 1 : Output screen of Example 1

### 1.2.2 Swing

In section 1.2.1 we learned about AWT in Java, the first GUI API provided by Java programming language. Swing is another API for developing Windows based applications in Java. Due to the limitation of AWT (components use native code resources, heavyweight, platform-dependent, etc.), Swing was introduced in the year of 1997 which is a platform-independent “model–view–controller” GUI framework for Java. The components of swing are written in Java which is a part of Java Foundation Classes (JFC). Swing API in Java not only provides platform independence but also are lightweight components. JFC consists of Swing, Java2D, Accessibility, Internationalization and pluggable look and feel support API. The JFC has been integrated into code Java since JDK1.2.

As mentioned above, Swing is developed on top of the AWT, but it has many differences. Some significant differences are given in table 2.

**Table 2:** Major Differences between AWT and Swing

S.No.	Java Swing	Java AWT
1	It is platform-independent API.	It is platform-dependent API
2	It has lightweight GUI components.	It has heavyweight GUI components.
3	It also supports pluggable look and feel of GUI.	It does not support pluggable look and feel.
4	It provides more advanced components than AWT e.g. tables, lists, scrollpanes, colorchooser, tabbedpane etc.	It provides fewer components than Swing
5	It supports the MVC design pattern	It does not follow the MVC design pattern.
6.	Execution is faster	Execution is slower
7.	The components of Swing do not require much memory space	The components of AWT require more memory space

The hierarchy of the Swing package/class in Java is given in figure 3.

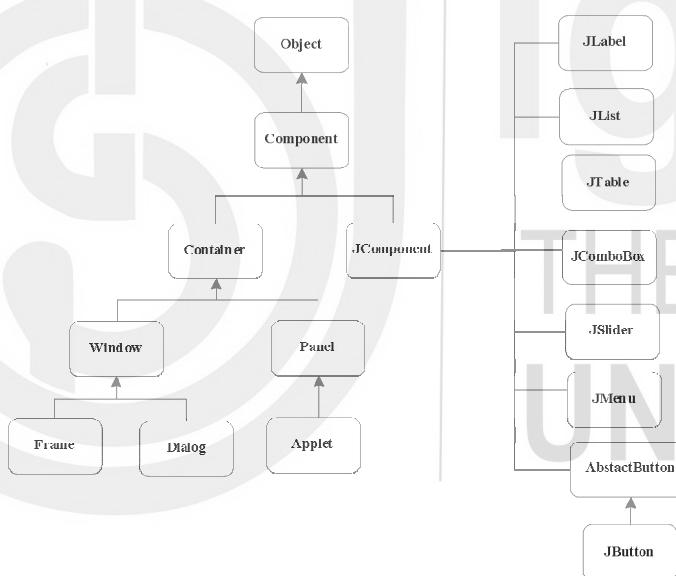


Figure 2: Hierarchy of Swing Package/Class

As explained above, Swing is developed on top of the AWT which overcomes the limitations of AWT. Two major features of the Swing Components are:

- Lightweight
- Swing Supports a Pluggable Look and Feel

These features provides an effective and easy-to-use solution to the problems of GUI development in Java. Both the features of Swing are explained below:

**Swing Components Are Lightweight:** As we know the Swing has been written in Java which make it platform-independent and does not map directly to platform-specific peers. Another reason is that it is lightweight because its components are rendered using graphics primitives; components can be transparent, which enables

non rectangular shapes. That is why the lightweight components are flexible and more efficient. It determines the look and feel of each component; therefore, lightweight components do not translate into native peers. This means that each component of the Swing will work in a consistent manner across all the platforms.

The look and feel of a component is controlled by Swing. It supports Pluggable Look and Feel because each of the component is rendered by Java code in place of native peers. It means that it is possible to separate the look and feel of a component from the logic of the component in Swing. By separating the look and feel, it provides a significant advantage. It makes it possible to “plug in” a new look and feel for any given component without having any adverse effects on the code that uses it. Hence, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, programmer need to simply apply “plug-in” in the design. Once this is done, all components are automatically rendered using that style. For example, if you are sure that an application is going to run only in a Windows environment, it is possible to specify the Windows look and feel. Also, through proper programming, the look and feel can be changed dynamically at run time. But we are not going to discuss that in this course. Now let us see the example program developed using Swing in Java

### Example 2: Use of Swing components

```
package org.ignou.gui;
import javax.swing.*;
import java.awt.*;
public class MySwingExample extends JFrame
{
    public MySwingExample ()
    {
        //setting title of frame as My Window
        setTitle("Swing Example");

        //Creating a label named Welcome to My Second Window
        JLabel jLabel1 = new JLabel("Welcome to First Swing Programme.");

        //adding label to frame.
        add(jLabel1);

        //setting layout using Flow Layout object.
        setLayout(new FlowLayout());

        //setting Window close operation.
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //setting size
        setSize(400, 400);

        //setting frame visibility
        setVisible(true);
    }
    public static void main (String[] args)
    {
        new MySwingExample ();
    }
}
```

### Output:

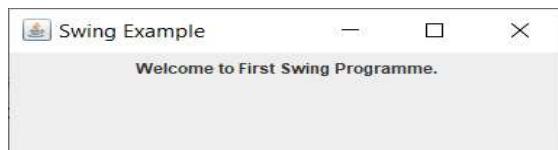


Figure 3: Output screen of above example 2

Graphical User Interface  
and Java Database  
Connectivity

### 1.2.3 JavaFX

JavaFX is the most popular set of APIs for GUI development using Java, which is used to develop Windows-based applications and Rich Internet Applications (RIA) that can run through a wide variety of devices. JavaFX is the next-generation open-source client application platform for implementing mobile, desktop, and embedded systems, which is built on the Java platform. Firstly, it was introduced into JDK1.8, which intended to replace Swing in Java. An application built on JavaFX can run on multiple platforms, including Web, Mobile and Desktops. In this unit we will deep dive into all essential aspects of JavaFX.

Java provides its GUI APIs (e.g., AWT/Swing/JavaFX graphic) with JDK. Also, apart from this, there are some other GUI tools that work with Java easily, such as Google Web Toolkit (GWT), which Google provides, and Standard Widget Toolkit (SWT) provided by Eclipse.

This unit mainly focuses on a practical base session on the latest GUIs APIs called JavaFX. Prerequisite software/tools/libraries to a setup development environment for JavaFX:

Minimum prerequisite software/tools/libraries	Version of software/tools/libraries used during this unit writing(for running JavaFX programs)
JDK 8 and above	JDK 11
Any IDE (Eclipse/NetBeans/IntelliJ Idea, etc.)	IntelliJ Idea
Latest JavaFX Libraries	JavaFX 11 LTS

#### Now let us create the first Programme using JavaFX

1. Create a new JavaFX Project using IntelliJ IDEA
  1. Open IntelliJ IDEA which is preinstalled in your PC, if not installed then install it first.
  2. There are two way to create project, first Click on + New Project from welcome screen as shown in figure 5. Second Click to File menu → New → Project...
  3. Popup window of New Project will display where you have to select project type and SDK as shown in figure no 6.
  4. Enter the Project Name(JavaFXFirstProgramme), Location, Language, select SDK, Change Group and Artifacts as per your requirement then click Next button to .



Figure 4: Create New Project in IntelliJ IDEA

## Introduction to GUI in Java

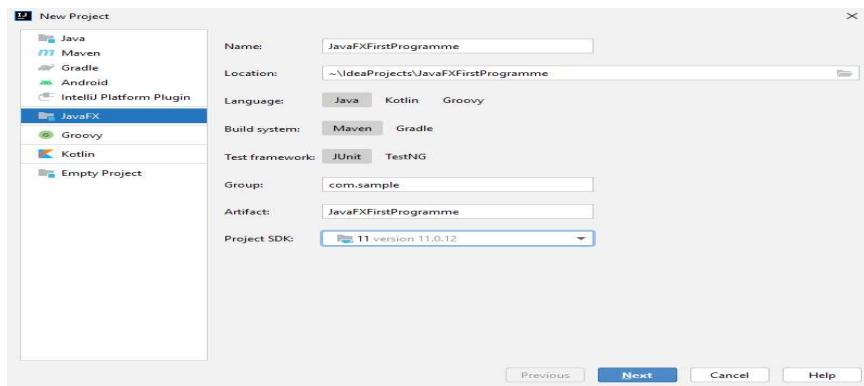


Figure 5: Creating Java FX Application in IntelliJ IDEA

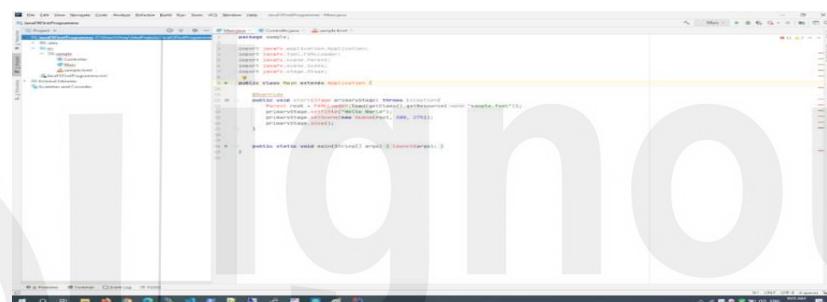


Figure 6: First look of default JavaFX Project with Error

5. Default project has been created with error because of running JDK 11 and missing the JavaFX SDK. This error will not generate if you use JDK8, because JDK 8 provides an inbuilt library of JavaFX. Then you have to configure the global library. Then you may create a new project of JavaFX running on JDK 11.
6. For fixing the above error and setting up JavaFX Library in the Project, first we click right-click on the project and select open module settings as shown in figure no 8 .

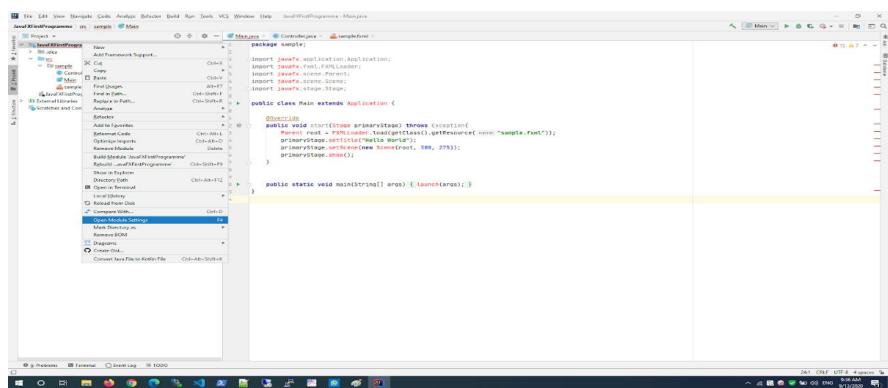


Figure 7: Open Project Setting

7. Firstly, check if the project SDK is correctly configured or not. Here the project language level needs to be configured the same as the project SDK selected during the Project creation. In the case of JDK 11, please follow instructions as specified in figure no 9:

- In project Tab:
  - Project SDK 11
  - Project Language Level: 11- Local Variable Syntax for lambda Parameters
- In Module Tab:
  - Sources Language Level: 11- Local Variable Syntax for lambda Parameters
  - Dependency: Module SDK 11

if you are using JDK-8, you must select **Project SDK 8** and **Project Language Level: 8 – Lambdas, Type annotations etc.** If in your case, you have to set it as per your JDK version, you are having on your PC/laptop.

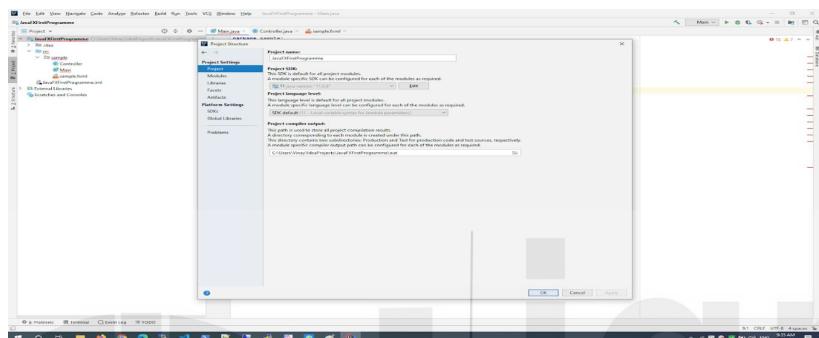


Figure 8: Project settings in IntelliJ IDEA

8. Next step is to configure global libraries of JavaFX-SDK-11. For this you have to go to the global libraries in Platform Setting option in Project Structure as shown in figure no. 10. Here we need to click on Plus(+) icon to add new Global Library. Doing this we are configuring IntelliJ for the specific libraries that we need to use in this particular Project.

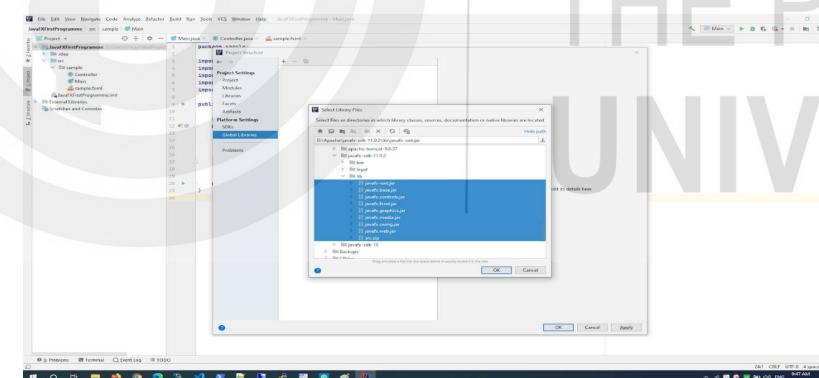


Figure10: Setting Global Libraries in Project Structure

Now you will no longer get any errors as you observed in the figure no 7, but still you will find that if you try to run this application, you will get an **Error: “JavaFX runtime components are missing, and are required to run this application”**. This error is only in case of JDK 11, but JDK 8 you will not have this issue.



Figure 11(a): Default Project without error while using JDK 11

To fix this error, you need to add a module - info.java file in the source code directory, which define the JavaFX control so that the code will work with JDK 11.

Right click on src folder → New-->Module-Info.java and add the following line of codes:

```
module JavaFXFirstProgramme
{
    requires JavaFx.fxml;
    requires JavaFx.controls;
    opens sample;
```

No test run the application; you will get the following output:

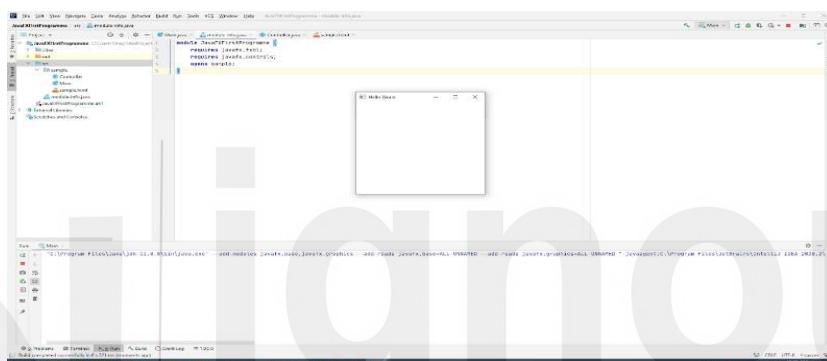


Figure 9(b): Output screen of First Run JavaFX Project

Finally, we have a directory structure of the First JavaFX Project as shown in figure no 12.

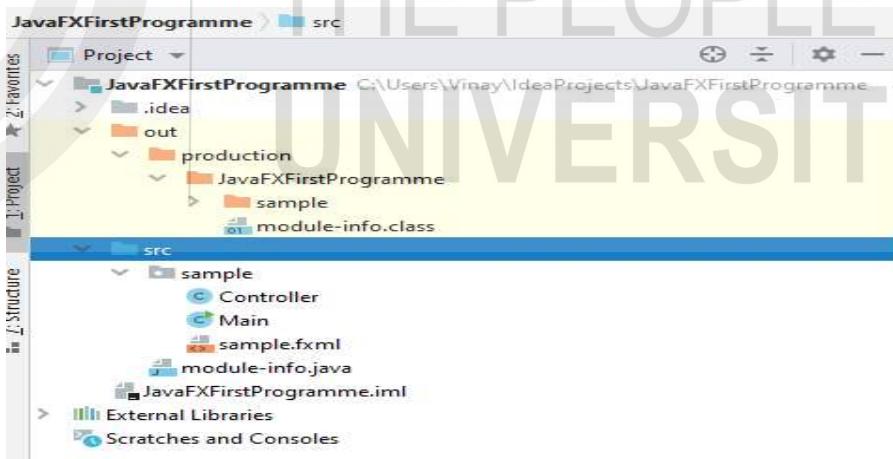


Figure 10: JavaFX Project Directory Structure

## 1.3 Features of JavaFX

In this section, we will discuss the features of JavaFX along with its components. As per the official JavaFX documentation, the following features have been incorporated

since the release of JavaFX 8 and later versions. The main items which were announced since the release of JavaFX 8 are given below:

Graphical User Interface  
and Java Database  
Connectivity

- JavaFX is completely written in Java which contains classes and interfaces. Its API is designed for alternative use of Java Virtual Machines (JVM) such as JRuby and Scala.
- JavaFX supports two types of coding styles FXML and Scene builder. FXML is an XML based interface; however, scene builder is a pure Java interface. Scene builder in JavaFX is used for interactively designing the graphical user interface (GUI). Scene builder in JavaFX creates FXML, which can be ported to an IDE where a developer can also add the application's business logic.
- JavaFX supports WebView using WebKitHTML technology to embed web pages. JavaScript running in WebView can be called by Java APIs. Since the release of JavaFX 8, it has also supported HTML5 containing Web Sockets, Web Workers, Web Fonts and printing capabilities features.
- Some JavaFX features can be implemented in existing Swing applications, such as enabled web content and rich graphics media playback. This feature has been incorporated since the release of JavaFX 8. All the key controls are required to develop full-featured application, which is available in JavaFX 8 release, including standard Web Technology such as CSS (e.g., DatePicker and TableView controls are also available in JavaFX including a public API for CSS Styleable class which allows objects to be styled by CSS.)
- The 3D Graphics libraries are introduced in JavaFX 8 release. API for Shape3D (Box, Cylinder, MeshView and Sphere subclasses), SubScene, PickResult, Material, SceneAntialiasing and LightBase (AmbientLight, PointLight subclasses) are added in 3D Graphics libraries. It also comprises of the Camera API class.
- Canvas API: This API allows drawing directly within an area of the JavaFX scene that contains one graphical element which is also known as node.
- Printing API: The JavaFx.print package is included in Java SE 8 release provides the public classes for the JavaFX Printing API.
- Rich Text Support: The JavaFX 8 brings enriched text support to JavaFX comprising bi-directional text and complex text scripts such as Thai and Hindu in controls and multi-line, multi-style text in text nodes.
- Multi-touch Support: The JavaFX provides support for multi-touch operations based on the capabilities of the underlying platform.
- Hi-DPI support: The JavaFX 8 also supports Hi-DPI displays(Hi-DPI displays have increased pixel density).
- Hardware-accelerated graphics pipeline: The JavaFX graphics are based on the graphics rendering pipeline (Prism). JavaFX allows smooth graphics that render quickly through Prism when it is used with a supported graphics card or graphics processing unit (GPU). If a system does not feature one of the recommended GPUs supported by JavaFX, then the Prism defaults that to the software rendering stack.
- High-performance media engine: The media pipeline supports the playback of web multimedia content. It provides a stable, low-latency media framework that is based on the ‘GStreamer’ multimedia framework.
- Self-contained application deployment model: Self-contained application packages have all of the application resources and a private copy of the Java and JavaFX runtimes. They are distributed as native installable packages and

provide the same installation and launch experience as native applications for that operating system.

The above items related to JavaFX are given here to make you aware of the various features and supports provided by JavaFX8. Detailed coverage of the above items/points are beyond scope of this course.

#### 1.4 USER INTERFACE COMPONENTS OF JAVAFX

Your familiarity with the form components available in HTML will help you learn JavaFX components. In JavaFX many components are available for developing GUI. In this section, you will learn components in JavaFX . JavaFX controls are the components that provide control functionalities in application development using JavaFX. In GUI Programming, the UI element is the core graphical element that users see and interact with. JavaFX also provides a wide list of common elements/controls (e.g., label, checkbox , textbox, menu, a button, radio button, table, tree view, date picker etc.) from basic to advanced level of uses in JavaFX programming. The package “**Javafx.controls**” in JavaFX defines various classes to create the GUI components (controls). The package “**Javafx.scene.chart**” define various types of charts and package. The “**Javafx.scene.Scene**” provides the scene and its components that are available for the JavaFX UI toolkit. JavaFX supports several controls like Table view, Treeview, FileChooser, date picker, button text field etc. Controls are mainly nested inside a layout component, and it manages the layout of controls. The following list of important controls available in JavaFX:

- Accordion
- Button
- CheckBox
- ChoiceBox
- ColorPicker
- ComboBox
- DatePicker
- Label
- ListView
- Menu
- MenuBar
- PasswordFile
- ProgressBar
- RadioButton
- Slider
- Spinner
- SplitMenu
- Button
- SplitPane
- TableView
- TabPane
- TextArea
- TextField
- TitledPane
- ToggleButton
- ToolBar
- TreeTable
- View
- TreeView

Every component needs to be initialized with a new key before it is used. Let us see some important components one by one:

- **Accordion:** Accordion is the graphical control that looks like a collapsible content panel to display textual or graphical information in limited space along with a scrollbar if needed. It is very similar to the accordion component in HTML and bootstrap. It's used as a container that contains multiple controls internally (such as label, textbox, button, image, etc), each of which may have its own contents. The Accordion control is implemented by the class “**Javafx.scene.control.Accordion**”.

##### Example 3: JavaFX Accordion implementation

```
import Javafx.application.Application;
import Javafx.fxml.FXMLLoader;
import Javafx.scene.Parent;
import Javafx.scene.Scene;
import Javafx.scene.control.Accordion;
import Javafx.scene.control.Label;
import Javafx.scene.control.TitledPane;
import Javafx.scene.layout.VBox;
```

```

import JavaFx.stage.Stage;
public class MyAccordionControlExample extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        Parent root = FXMLLoader.load(getClass().getResource("mySample.fxml"));
        primaryStage.setTitle("Accordion Control Example");
        primaryStage.setScene(new Scene(root, 500, 500));

        Accordion accordion1 = new Accordion();

        TitledPane pane1 = new TitledPane("Student" , new Label("Show all menu available in Student"));
        TitledPane pane2 = new TitledPane("Faculty" , new Label("Show all menu available in Faculty"));
        TitledPane pane3 = new TitledPane("Books", new Label("Show all menu available in Student Books"));

        accordion1.getPanels().add(pane1);
        accordion1.getPanels().add(pane2);
        accordion1.getPanels().add(pane3);

        VBox vBox = new VBox(accordion);
        Scene scene = new Scene(vBox);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

**Output:**



Figure 11: Output screen of Example no 3

- **Label:** It is a JavaFX component that is used to display textual or graphical content in GUI. It's also called non-editable text control. It can be created using the class “**JavaFx.scene.control.Label**”.
- **Button:** The button control is used for enabling some action executed when the application user clicks the button. The Button control is created using the class “**JavaFx.scene.control.Button**”.

- **ColorPicker:** It is a component used to choose or manipulate color in a dialog box. The ColorPicker control is implemented by the class “**JavaFx.scene.control.ColorPicker**”.
- **CheckBox:** This component enables users to check/tick a component that can be either on (true) or off (false). The CheckBox control is implemented by the class “**JavaFx.scene.control.CheckBox**”.
- **RadioButton:** This component is used in a situation where one needs to have either an ON (true) or OFF (false) state in a group. This control is implemented by the class “**JavaFx.scene.control.RadioButton**”.
- **ListView:** A ListView component of JavaFX is used to present the user with a scrolling list of text items. This control is represented by the class “**JavaFx.scene.control.ListView**”.
- **TextField:** A TextField component in JavaFX allows users to edit a single line text. It is implemented by the call “**JavaFx.scene.control.TextField**”.
- **PasswordField:** A PasswordField is a special text component that allows the user to enter a secure or sensitive text in a text field. It differs from TextField because the entered text does not show after typing. It is implemented by the class “**JavaFx.scene.control.PasswordField**”.
- **Scrollbar:** A Scrollbar component is used to allow the user to select from a range of values.
- **FileChooser:** A FileChooser control provides a dialog window from which the user can select a file.
- **ProgressBar:** Using this component, as the task progresses towards completion, the task's percentage of completion can be shown.
- **Slider:** A Slider lets the user graphically select a value by sliding a knob within a bounded interval.

---

## 1.5 WORK WITH LAYOUTS

---

Layout in GUI programming is used for organizing the UI elements or components on the screen and providing a final look and feel to the GUI (Graphical User Interface). This section will discuss the Layouts and what JavaFX Layout allows us to do. The JavaFX provide various predefined layouts such as **Grid Pane**, **Anchor Pane**, **Stack Pane**, **H Box**, **V Box**, **Flow Pane**, **Tile Pane**, **Border Pane**, **Text Flow**, etc. There is no need to memorize all the layouts because each layout is denoted by a class and all these classes belong to the package **JavaFx.layout**. The base class of all layouts in JavaFX is **Pane** class. The default layout size in JavaFX is 300x275.

Before you get into the layouts, let us discuss what is meant by the preferred size for JavaFX controls because preferred sizes are an essential concept. Every control computes its preferred size based on its contents, so what is the need of talking about preferred size? What is meant by the preferred width and height of the control? When it is displayed?

JavaFX has a button control. As you are familiar with buttons, especially ones for ok and cancel, which you have seen in various applications. You might have observed that by default the button control will size itself. In other words, if we are working with an ok button, the button control will size itself so that its border fits around that text ok. So, it would not stretch itself across the width of the entire window. For example, it will only be just wide enough to accept that text, so layouts often use a

preferred size of the controls they are laying out to determine how much space it controls. When the controller is placed into a layout it becomes a child of that layout, so some layouts will ensure that their children display at the preferred widths or heights and sometimes it would depend on where controllers are placed within the layout.

## Layout Pane Classes

Let us discuss Layout Panes Classes; JavaFX contains several container classes. Figure 3 shows the Hierarchy of Layout Classes provided by JavaFX. Layout Panes is the subclass of the package `javafx.scene.layout`. Layout can be created using Java files as well as FXML files.

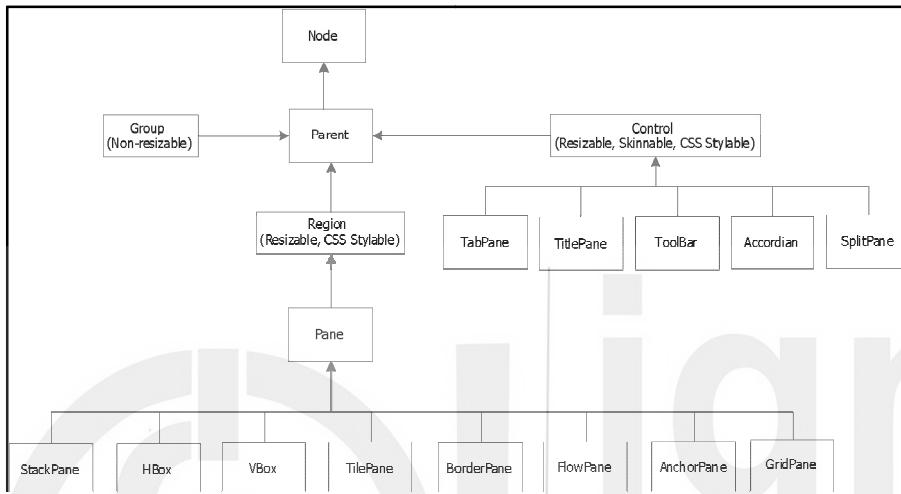


Figure 12: JavaFX 2.0 Layout Hierarchy

**GridPane** is the default layout of JavaFX Project setup as shown in figure 4.

**Main.java**

```

package ignou;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("ignou.fxml"));
        primaryStage.setTitle("Welcome to Layout in JavaFx");
        primaryStage.setScene(new Scene(root, 500, 275));
        primaryStage.show();
    }

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

**FXML file Entry**

```

<?xml version="1.0" encoding="UTF-8"?>
<fx:root fx:controller="ignou.Controller" xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
</fx:root>
  
```

Figure 13: Default Pane coding in JavaFX Project

- **HBox:** The HBox layout organizes all the nodes in an application in a single horizontal row. The class named `HBox` of the package `Javafx.scene.layout` signifies the text horizontal box layout.

### Syntax in FXML:

<?import Javafx.scene.layout.HBox?>

```
<HBox fx:controller="ignou.Controller" xmlns:fx="http://JavaFx.com/fxml"
      alignment="center">
    <Label text="Welcome to HBox Layout using FXML"/>
    <Button text="Button 1"/>
    <Button text="Button 2"/>
</HBox>
```

#### Syntax in Java:

```
Label label1 =new Label("Welcome to HBox Layout with Pure Java code");
Button button1 = new Button("Button Number 1");
Button button2 = new Button("Button Number 2");
HBox hbox = new HBox(label1, button1, button2);
primaryStage.setScene(new Scene(hbox, 500, 275));
```

- **VBox:** The VBox layout organizes all the nodes in our application in a single vertical column. The class named VBox of the package JavaFx.scene.layout denotes the text Vertical box layout.

#### Syntax in FXML:

```
<?import JavaFx.scene.layout.VBox?>
<VBox fx:controller="ignou.Controller" xmlns:fx="http://JavaFx.com/fxml"
      alignment="center">
    <padding>
      <Insets bottom="10" right="10"/>
    </padding>
    <Label text="Welcome to VBox Layout"/>
    <Button text="Button 1"/>
    <Button text="Button 2"/>
</VBox>
```

#### Syntax in Java:

```
Label label1 = new Label("Welcome to VBox Layout using Java Code");
Button button1 = new Button("Button 1");
Button button2 = new Button("Button 2");
VBox vbox = new VBox(label1, button1, button2);
primaryStage.setScene(new Scene(vbox, 500, 275));
```

- **BorderPane:** The Border Pane layout organizes the nodes in our applications in top, left, right, bottom and center positions. The class BorderPane of the package JavaFx.scene.layout is used for the border pane layout.

#### Syntax in FXML:

```
<?import JavaFx.scene.layout.BorderPane?>
<BorderPane fx:controller="ignou.Controller"
            xmlns:fx="http://JavaFx.com/fxml">
    <bottom>
        <HBox xmlns:fx="http://JavaFx.com/fxml">
            <padding>
                <Insets bottom="10" right="10"/>
            </padding>
            <Label text="Welcome to BorderPane Layout"/>
            <Button text="Button 1" prefWidth="90"/>
            <Button text="Button 2" prefWidth="90"/>
        </HBox>
    </bottom>
</BorderPane>
```

#### Syntax in Java:

```
Label label1 = new Label("Welcome to BorderPane Layout using Java  
Code");  
Button button1 = new Button("Button 1");  
Button button2 = new Button("Button 2");  
BorderPane borderPane = new BorderPane();  
borderPane.setTop(label1);  
borderPane.setRight(button1);  
borderPane.setLeft(button2);  
primaryStage.setScene(new Scene(borderPane, 500, 275));
```

- **StackPane:** The stack pane layout organizes the nodes in an application on top of another component like in a stack. In this pane, the node added first is placed at the bottom of the stack and the next node added is placed on top of it. The class StackPane is in the package Javafx.scene.layout denotes the stack pane layout. In the following example Button 1 is top of the Label, and Button 2 is top of the Button 1.

#### Syntax in FXML:

```
<?import Javafx.scene.layout.StackPane?>  
<StackPane fx:controller="ignou.Controller"  
xmlns:fx="http://Javafx.com/fxml">  
    <padding>  
        <Insets bottom="10" right="10"/>  
    </padding>  
    <Label text="Welcome to StackPane Layout"/>  
    <Button text="Button 1" />  
    <Button text="Button 2"/>  
</StackPane>
```

#### Syntax in Java:

```
Label label1 = new Label("Welcome to StackPane Layout using Java Code");  
Button button1 = new Button("Button 1");  
Button button2 = new Button("Button 2");  
StackPane stackPane = new StackPane();  
stackPane.setMargin(label1, new Insets(50, 50, 50, 50));  
ObservableList observableList = stackPane.getChildren();  
observableList.addAll(label1, button1, button2);  
primaryStage.setScene(new Scene(stackPane, 700, 275));
```

- **TextFlow:** The Text Flow layout organizes multiple text nodes in a single flow. The class TextFlow in the package Javafx.scene.layout denotes the text flow layout.

#### Syntax in FXML:

```
<?import Javafx.scene.text.TextFlow?>  
<TextFlow fx:controller="ignou.Controller"  
xmlns:fx="http://Javafx.com/fxml" >  
    <padding>  
        <Insets bottom="10" right="10"/>
```

```
</padding>
<Label text="Welcome to TextFlow Layout"/>
<Button text="Button 1"/>
<Button text="Button 2"/>
</TextFlow>
```

### Syntax in Java:

```
Text text1 = new Text("Welcome to TextFlow Layout");
text1.setFont(new Font(15));
text1.setFill(Color.DARKSLATEBLUE);

Button btn1 = new Button("Button 1");
Btn1.setFont(Font.font("Times new Roman", FontWeight.BOLD, 12));

TextFlow textFlowPane = new TextFlow();

textFlowPane.setAlignment(TextAlignment.JUSTIFY);
textFlowPane.setPrefSize(10, 10);
textFlowPane.setLineSpacing(5.0);
ObservableList list1 = textFlowPane.getChildren();
list1.addAll(text1, btn1);
Scene scene = new Scene(textFlowPane);
stage.setTitle("text Flow Pane Example");
stage.setScene(scene);

stage.show();
```

- **AnchorPane:** The Anchor pane layout used to anchor the nodes in an applications at a particular distance from the pane. The class AnchorPane in the package JavaFx.scene.layout denotes the Anchor Pane layout.

### Syntax in FXML:

```
<?import JavaFx.scene.layout.AnchorPane?>
<AnchorPane fx:controller="ignou.Controller"
xmlns:fx="http://JavaFx.com/fxml">
<padding>
<Insets bottom="10" right="10"/>
</padding>
<Label text="Welcome to AnchorPane Layout"/>
<Button text="Button 1" />
<Button text="Button 2"/>
</AnchorPane>
```

### Syntax in Java:

```
Text text1 = new Text("Welcome to AnchorPane Layout");
text1.setFont(new Font(15));
text1.setFill(Color.DARKSLATEBLUE);
Button btn1 = new Button("Button 1");
Btn1.setFont(Font.font("Times new Roman", FontWeight.BOLD, 12));
```

```
AnchorPane anchorPane1 = new AnchorPane();
anchorPane1.setTopAnchor(text1, 50.0);
anchorPane1.setTopAnchor(btn1, 50.0);
VBox vBox=new VBox(anchorPane1);
```

- **TilePane:** Using Tile Pane layout the nodes in an application are added in the form of uniformly sized tiles. The class TilePane is in the package JavaFx.scene.layout denotes the TilePane layout.

**Syntax in FXML:**

```
<?import JavaFx.scene.layout.TilePane?>
<TilePane fx:controller="ignou.Controller"
xmlns:fx="http://JavaFx.com/fxml">
    <padding>
        <Insets bottom="10" right="10"/>
    </padding>
    <Label text="Welcome to TilePane Layout"/>
    <Button text="Button 1" />
    <Button text="Button 2"/>
</TilePane>
```

**Syntax in Java:**

```
Text text1 = new Text("Welcome to TilePane Layout");
text1.setFont(new Font(15));
text1.setFill(Color.DARKSLATEBLUE);
Button btn1 = new Button("Button 1");
Btn1.setFont(Font.font("Times new Roman", FontWeight.BOLD, 12));
TilePane tilePane1 = new TilePane();

tilePane1.getChildren().add(text1);
tilePane1.getChildren().add(btn1);
Scene scene = new Scene(tilePane1, 10, 10);
```

- **GridPane:** The Grid Pane layout is used to organize the nodes in an application as a grid of rows and columns. This layout comes handy while creating forms using JavaFX. The class GridPane in the package JavaFx.scene.layout denotes the GridPane layout.

**Syntax in FXML:**

```
<?import JavaFx.scene.layout.GridPane?>
<GridPane fx:controller="ignou.Controller"
xmlns:fx="http://JavaFx.com/fxml" alignment="center" hgap="10"
vgap="10">
    <Label text="Welcome to Gridpane Layout" GridPane.rowIndex="0"
GridPane.columnIndex="1"/>
    <Button text="Button 1" GridPane.columnIndex="0"
GridPane.rowIndex="1"/>
    <Button text="Button 2" GridPane.columnIndex="0"
GridPane.rowIndex="2"/>
</GridPane>
```

**Syntax in Java:**

```
Text text1 = new Text("Welcome to Gridpane Layout");
text1.setFont(new Font(15));
text1.setFill(Color.DARKSLATEBLUE);
Button btn1 = new Button("Button 1");
Btn1.setFont(Font.font("Times new Roman", FontWeight.BOLD, 12));
```

```
GridPane gridPane1= new GridPane ();
gridPane1.add(text1, 0, 0, 1, 1);
gridPane1.add(btn1, 1, 0, 1, 1);
Scene scene = new Scene(gridPane1, 40, 20);
```

- **FlowPane:** The flow pane layout wraps all the nodes in a flow. A horizontal flow pane wraps the elements of the pane at its height, while a vertical flow pane wraps the elements at its width. The class `FlowPane` in the package `Javafx.scene.layout` denotes the Flow Pane layout.

#### Syntax in FXML:

```
<?import Javafx.scene.layout.FlowPane?>
<FlowPane fx:controller="ignou.Controller"
xmlns:fx="http://Javafx.com/fxml" orientation="HORIZONTAL">
<padding>
    <Insets bottom="10" right="10"/>
</padding>
<Label text="Welcome to FlowPane Layout"/>
<Button text="Button 1" />
<Button text="Button 2"/>
</FlowPane>
```

#### Syntax in Java:

```
Text text1 = new Text("Welcome to FlowPane Layout");
text1.setFont(new Font(15));
text1.setFill(Color.DARKSLATEBLUE);
Button btn1 = new Button("Button 1");
Btn1.setFont(Font.font("Times new Roman", FontWeight.BOLD, 12));

FlowPane flowpane1 = new FlowPane();

flowpane1.getChildren().add(text1);
flowpane1.getChildren().add(btn1);

Scene scene = new Scene(flowpane1, 40, 20);
```

#### Steps to create layout in JavaFX :

- **Create node:** First of all, create the required nodes of the JavaFX application by instantiating their respective classes.
- **Instantiate the respective class of the required layout:** After creating the nodes (and completing all the operations on them), instantiate the class of the required layout.
- **Set the properties of the layout:** After instantiating the class, you need to set the layout's properties using their respective setter methods.
- **Add all the created nodes to the layout:** Finally, you need to add the object of the shape to the group by passing it as a parameter of the constructor.

#### Example 4: Creating a Layout using GridPane in pure Java Code

```

package org.ignou.gui;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;
public class MyGridPaneExample extends Application
{
@Override
public void start(Stage primaryStage) throws Exception
{
primaryStage.setTitle("Welcome to GridPane Layout in JavaFx
using Pure Java code");

GridPane grid1 = new GridPane();
grid1.setAlignment(Pos.CENTER);
grid1.setHgap(10);
grid1.setVgap(10);
grid1.setPadding(new Insets(25, 25, 25, 25));

Text sceneTitle = new Text("Welcome to First Layout in JavaFx
using Pure Java code");
sceneTitle.setFont(Font.font("Times new Roman",
FontWeight.NORMAL, 20));
grid1.add(sceneTitle, 0, 0, 2, 1);

Label userName = new Label("User Name:");
grid1.add(userName, 0, 1);

TextField userTextField = new TextField();
grid1.add(userTextField, 1, 1);

Label pw = new Label("Password:");
grid1.add(pw, 0, 2);

PasswordField pwBox = new PasswordField();
grid1.add(pwBox, 1, 2);

Button btn = new Button("Login");
btn.setFont(Font.font("Times new Roman", FontWeight.BOLD, 20));
grid1.add(btn, 1, 4);

primaryStage.setScene(new Scene(grid1, 700, 275));
primaryStage.show();
}
public static void main(String[] args)
{
launch(args);
}
}

```

#### Output:



### ☛ Check Your Progress -1

1. What is the difference between AWT and Swing?

.....  
.....  
.....  
.....

2. Define user interface component in JavaFX?

.....  
.....  
.....  
.....

3. Explain the Layout Hierarchy in JavaFX?

.....  
.....  
.....  
.....

---

## 1.6 ADD HTML CONTENT

Before going to HTML content in JavaFX, let us briefly describe CSS in HTML. What is a Cascading Style Sheet? A cascading style sheet (CSS) is a language used to describe the presentation (the look or the style) of UI elements in a GUI application. CSS was primarily developed for use in web pages for styling HTML elements. It allows for the separation of the presentation from the content and behaviour. In a typical web page, the content and presentation are defined using HTML and CSS, respectively. JavaFX allows us to define the look and feel (or the style) of JavaFX applications using CSS. It can define UI elements using JavaFX class libraries or FXML and use CSS to define their look and feel. The CSS provides the syntax to write rules to set the visual properties. Following are the steps to write the CSS classes:

- A rule consists of a selector and a set of property-value pairs.
- A selector is a string that identifies the UI elements to which the rules will be applied.
- A property-value pair will have a property name and its corresponding value separated by a colon (:).
- Two property-value pairs are separated by a semicolon (;).
- A set of property-value pairs is enclosed within curly braces ({} ) preceded by the selector.

An example of a rule in CSS is given below:

```
.button { -fx-background-color: red;  
          -fx-text-fill: white;}
```

### What are Styles, Skins, and Themes?

- Styles, skins, and themes are three related concepts. A CSS rule is also known as a style. A collection of CSS rules is known as a style sheet. Styles provide a mechanism to separate the presentation and content of UI elements. They also facilitate grouping of visual properties and their values, so that they can be shared by multiple UI elements. JavaFX provides styles using JavaFX CSS.
- Skins are collections of application-specific styles which define the appearance of an application. Skinning is the process of changing the appearance of an application (or the skin) on the fly. JavaFX does not provide a specific mechanism for skinning.
- Themes are the visual characteristics of an operating system that are reflected in the appearance of UI elements of all applications. For example, changing the theme on the Windows operating system changes the appearance of UI elements in all applications that are running. The skins are application specific, whereas themes are operating system specific.

To implement CSS Style in JavaFX through style file following steps need to be followed:

- Create a resources/css folder in the project folder
- Add CSS file(Style Sheet) in css folder
- Write rules in Stylesheet as explained above
- Set scene property using getStylesheets() method  

```
getStylesheets().add("file:resources/css/style.css");
```
- Set style/CSS ID in JavaFX's component using getStyleClass() method  

```
getStyleClass().add("button1"); (where button1 is the  
same ID ruled in CSS.)
```

#### Example 5: CSS Style implementation in Java FX

##### 1. MyHtmlCSSStyleExample.Java

```
package org.ignou.gui;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MyHtmlCSSStyleExample extends Application
{
    public static void main(String[] args)
    {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage)
    {
        Text text1 = new Text();
        text1.setText("Welcome to use of CSS in JavaFX");

        /* Call Style ID from CSS file */
        text1.getStyleClass().add("text1");
    }
}
```



```
Button yesBtn = new Button("Yes");
/* Call Style ID from CSS file */
yesBtn.getStyleClass().add("button");

Button noBtn = new Button("No");
/* Call Style ID from CSS file */
noBtn.getStyleClass().add("button1");

Button cancelBtn = new Button("Cancel");
/* Call Style ID from CSS file */
cancelBtn.getStyleClass().add("button2");

HBox root = new HBox();
root.getChildren().addAll(text1, yesBtn, noBtn, cancelBtn);
Scene scenel = new Scene(root, 700, 70);

/* Add a stylesheet to the scene*/
scenel.getStylesheets().add("file:resources/css/style.css");
stage.setScene(scenel);
stage.setTitle("HTML CSS Styling Example");
stage.show();
}

}
2. Create style sheet file in the location:
resources/css/style.css

.button
{
    -fx-background-color: blue;
    -fx-text-fill: white;
    -fx-font-size: 10px;
    -fx-font: Arial;
    -fx-alignment: left;
}

.button1
{
    -fx-background-color: red;
    -fx-text-fill: white;
    -fx-font-size: 15px;
    -fx-font: Times new roman;
    -fx-alignment: center;
}

.button2
{
    -fx-background-color: green;
    -fx-text-fill: white;
    -fx-font-size: 20px;
    -fx-font: Arial;
    -fx-alignment: right;
}

.text1
{
    -fx-font: Arial;
    -fx-font-size: 20px;
}
```

### Output:

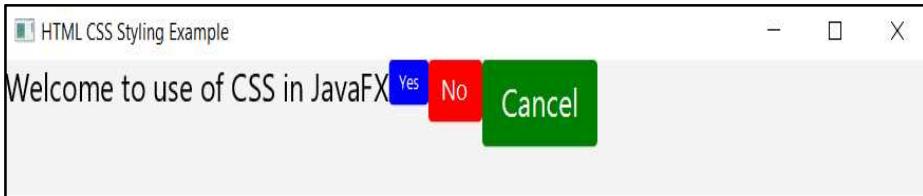


Figure 15: Output screen of Example no 5

## 1.7 ADD TEXT AND TEXT EFFECTS IN JAVAFX

As explained above, a single element in the scene is called a node which handles many types of content, which also include text. Text elements can be added as a textbox, label and using other components. The package of Text node is “*Javafx.scene.text*” and represented by the class named Text which is inherited from the Node class in JavaFX that used to display text. As per requirement, it can apply effects, transformations and animation to text nodes. All node types permit us to provide cultured text contents that fulfil the demands of modern rich Internet applications using all such features.

To add a text object to an application, first, instantiate a class as follow:

```
Text text1=new Text();
```

The data type of a class Text is string type which means it will store String text; to do that, we have to set the value using setText() method after instantiating a Text node:

```
String textString= "Welcome to Text Node";  
text1.setText(textString);
```

### Setting Text Position

A position of the text can be defined or set by specifying the value of the properties x and y using their respective setter methods namely setX() and setY():

```
text1.setX(50);  
text1.setY(50);
```

### Setting Text Font and Color

Text Font and color can be defined or set using its properties. You can use an instance of the “*Javafx.scene.text.Font*” class. The Font.font() method allows to specify the font family name and size. setFill() method can be used for setting a text color. Syntax of setting Text font and color as follow:

```
text1.setText("Setting Text Font and color");  
text1.setFont(Font.font ("Times New Roman", 20));  
text1.setFill(Color.RED);
```

Alternatively, we can also use a system font, which is platform dependent, means text font varies as per platform in which the application is running. To use this functionality, you have to call the Font.getDefault() method.

```
text1.setFont(Font.getDefault());
```

Alternatively, we can also use custom fonts in JavaFX. For using a custom font, you can embrace a TrueType font (.ttf) or an OpenType (.otf) in the application. To comprise font as a custom font, follow the following procedure:

- Create a resources/fonts folder in the project folder.
- Copy your font files to the fonts subfolder in the project created.
- In the source code, load the custom font as follow

```
text1.setFont(Font.loadFont("file:resources/fonts/AlexBrush-Regular.ttf", 120));
```

### Setting Text Bold or Italic

Use the FontWeight constant of the setFont() method to make Bold Text using follow syntax:

```
text1.setFont(Font.font ("Times New Roman", FontWeight.BOLD, 20));
```

The property named FontWeight accepts following 9 values:

- FontWeight.BLACK
- FontWeight.BOLD
- FontWeight.EXTRA\_BOLD
- FontWeight.EXTRA\_LIGHT
- FontWeight.LIGHT
- FontWeight.MEDIUM
- FontWeight.NORMAL
- FontWeight.SEMI\_BOLD
- FontWeight.THIN

Use the FontPosture constant of the setFont() method to make italic Text using follow syntax:

```
text1.setFont(Font.font ("Times New Roman", FontPosture.ITALIC, 20));
```

The property named FontPosture accepts 2 values FontPosture.REGULAR and FontPosture.ITALIC.

### Example 6: Setting a Font, Color and Font Size

```
package org.ignou.gui;

import JavaFX.application.Application;
import JavaFX.geometry.Pos;
import JavaFX.scene.Group;
import JavaFX.scene.Scene;
import JavaFX.scene.layout.GridPane;
import JavaFX.scene.paint.Color;
import JavaFX.scene.text.Font;
import JavaFX.scene.text.FontPosture;
import JavaFX.scene.text.FontWeight;
import JavaFX.scene.text.Text;
import JavaFX.stage.Stage;

public class MyTextMain extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("Welcome to Add Text and Text Effects in
JavaFX");
        GridPane grid1 = new GridPane();
        grid1.setAlignment(Pos.CENTER);
        grid1.setHgap(10);
        grid1.setVgap(10);

        /* Initialize Text and add string to display along with positioning
in layout. */
        Text text1 = new Text();
        String textString = "Welcome to Text 1 Node";
        text1.setText(textString);
        text1.setX(10);
        text1.setY(10);
    }
}
```

```
/* Setting of Font with font size and font color. */
Text text2 = new Text();
text2.setText("Setting Text 2 Font and color");
text2.setFont(Font.font("Times New Roman", 20));
text2.setFill(Color.RED);
text2.setX(10);
text2.setY(35);

/* Make Font Weight Bold */
Text text3 = new Text();
text3.setText("Setting Text 3 Font Bold");
text3.setFont(Font.font("Arial", FontWeight.BOLD, 20));
text3.setX(10);
text3.setY(65);

/* Set Font Posture Italic */
Text text4 = new Text();
text4.setText("Setting Text 4 Italic Font");
text4.setFont(Font.font("Times New Roman", FontPosture.ITALIC, 20));
text4.setX(10);
text4.setY(90);

/* Use of Custom Font Example */
Text text5 = new Text();
text5.setText("Use of Custom font in JavaFX");
text5.setFont(Font.loadFont("file:resources/fonts/AlexBrush-Regular.ttf", 40));
text5.setX(10);
text5.setY(130);

/* Grouping all text variables together */
Group group1 = new Group(text1, text2, text3, text4, text5);
grid1.add(group1, 0, 0);

/* putting text group on the scene */
primaryStage.setScene(new Scene(grid1, 700, 275));
primaryStage.show();
}

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

#### Output:



Figure 16: Output screen of Example no 6

### ☛ Check Your Progress-2

1. Define the terms Styles, Skins and Themes.

.....  
.....  
.....  
.....

2. Write a programme in JavaFX using the VBox pane?

.....  
.....  
.....  
.....

3. Explain the setFont() method and its property with example.

.....  
.....  
.....  
.....

---

## 1.8 SUMMARY

This unit gives an overview of the graphical user interface(GUI) in Java. In this unit AWT/Swing/JavaFX and User Interface Components of JavaFX are explained. Also GUI programming examples are given using coding style in AWT, Swing and JavaFX. Also, it is explained how to setup a project for JavaFX using intelliJ Idea and create the First Hello World program. The unit described how to work with layout in JavaFX along with coding style using FXML and pure Java code. This unit also explained concept of HTML Contents uses in JavaFX, text and various text effects in JavaFX with the help of programming examples.

---

## 1.9 SOLUTION/ANSWERS TO CHECK YOUR PROGRESS

---

### ☛ Check Your Progress-1

1. The difference between AWT and Swing are as follows:

Java Swing	Java AWT
It is platform-independent API.	It is platform-dependent API

It has lightweight GUI components.	It has heavyweight GUI components.	Graphical User Interface and Java Database Connectivity
It supports pluggable look and feel.	It doesn't support pluggable look and feel.	
It provides more advanced components than AWT like tables, lists, scrollpanes, colorchooser, tabbedpane etc.	it provides less components than Swing	
It supports MVC design pattern	It does not follow the MVC design pattern.	
Execution is faster	Execution is slower	
The components of swing do not require much memory space	The components of AWT require more memory space	

2. JavaFX controls are the components that provide control functionalities in JavaFX Applications. In GUI Programming, the user interface element are the core graphical elements, with that users see and interact with. JavaFX gives a wide list of common elements/controls such as label, textbox, checkbox, button, radio button, table, tree view, date picker, menu. The package “JavaFx.controls” defines various classes to create the GUI components, charts, and skins available for the JavaFX UI toolkit.
3. A layout in GUI is required to organize the User Interface elements on the screen and provide a final look and feel to the Graphical User Interface(GUI). In JavaFX, Layout defines the way in which the components are to be seen on the stage. It basically organizes the scene-graph nodes. We have several built-in layout panes in JavaFX: HBox, VBox, StackPane, FlowBox, AnchorPane, etc. Each Built-in layout is denoted by a separate class that needs to be instantiated to implement that particular layout pane. All these classes are a member of JavaFx.scene.layout package. The JavaFx.scene.layout.Pane class is the base class for all the built-in layout classes in JavaFX.

## Check Your Progress-2

1. Styles provide a mechanism to separate the presentation and content of UI elements. They also facilitate the grouping of visual properties and their values to be shared by multiple UI elements. JavaFX provides styles using JavaFX CSS.

Skins are collections of application-specific styles which define the appearance of an application. Skinning is the process of changing the appearance of an application (or the skin) on the fly. JavaFX does not provide a specific mechanism for skinning. However, the JavaFX CSS and JavaFX API, available for the Scene class and other UI-related classes, can easily provide skinning for JavaFX applications.

Themes are visual characteristics of an operating system that are reflected in the appearance of UI elements of all applications. For example, changing the theme on the Windows operating system changes the appearance of UI elements in all applications that are running.

```
2 . package org.ignou.gui;
import JavaFx.application.Application;
import JavaFx.scene.Scene;
import JavaFx.scene.control.Button;
import JavaFx.scene.layout.VBox;
import JavaFx.stage.Stage;
public class MyVBoxExample extends Application
{
```

```
public void start (Stage stage) throws Exception
{
    stage.setTitle("VBox Example");

    Button b1 = new Button ("Play ");
    Button b2 = new Button("Stop");

    VBox vbox1 = new VBox(b1, b2);
    Scene scene = new Scene (vbox1, 200, 100);
    stage.setScene(scene);
    stage.show();
}
public static void main (String[] args)
{
    Application.launch(args);
}
```

3. The `setFont()` method is used to change the font of the text. This method takes an object of the `Font` class. To set the font, you can use an instance of the `JavaFx.scene.text.Font` class. The `Font.font()` method permits you to specify the font family name and size.

The `setFont()` method takes four parameters: family, weight, posture, and size. The **family** parameter signifies the family of the font that you want to apply to the text. The **weight** property denotes the weight of the font, and it comprises values such as `BOLD`, `EXTRA_BOLD`, `EXTRA_LIGHT`, `LIGHT`, `MEDIUM`, `NORMAL`. The **posture** property represents the font posture, such as regular or italic. The **size** property represents the size of the font, for example:

```
Text t = new Text ("Graphical User Interface");
t.setFont(Font.font ("Arial", 18));
t.setFont(Font.font("Tahoma", FontWeight.BOLD, FontPosture.ITALIC,
18));
```

## 1.10 REFERENCE/FURTHER READING

- Carl Dea, Gerrit Grunwald, José Pereda, Sean Phillips and Mark Heckler “JavaFX 9 by Example”, Apress,2017.
- Sharan, Kishori, “ Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs”, Apress, 2014.
- Package and classes in AWT  
(<https://docs.oracle.com/javase/7/docs/api/Java/awt/package-summary.html>)
- JavaFXGetting Started with JavaFX  
Release 8(<https://docs.oracle.com/javase/8/javafx/get-started-tutorial/index.html>)
- <https://www.oracle.com/Java/technologies/Javase/Javafx-docs.html>
- <https://docs.oracle.com/javase/7/docs/api/Java/awt/package-summary.html>
- <https://openjfx.io/openjfx-docs/>
- <https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#BABEDDGH>
- <https://gluonhq.com/products/javafx/>
- <https://openjfx.io/Javadoc/15/javafx.controls/javafx.scene/control/package-summary.html>

---

## **UNIT 2 WORKING WITH UI CONTROLS**

---

### **Structure**

- 2.0 Introduction
  - 2.1 Objective
  - 2.2 Skin Applications with CSS
  - 2.3 Build UI with FXML
  - 2.4 Event Handling in JavaFX
  - 2.5 Effects, Animation and Media
  - 2.6 Summary
  - 2.7 Solutions/Answers to Check Your Progress
  - 2.8 References/Further Readings
- 

## **2.0 INTRODUCTION**

---

The previous unit of this block gave you an introduction to Graphical User Interface(GUI) in Java and also introduced UI controls. The JavaFX provides a wide list of UI control elements such as label, textbox, checkbox, button and radio button. The UI (User Interface) elements are those elements that are actually shown to the user for information exchange. The package `javafx.scene.control` have all the necessary classes required for the UI elements. This unit explains you how to JavaFX UI controls work with CSS, FXML and event handling.

Cascading Style Sheets (CSS) are used for adding styles such as text alignments, font-size, colors, including background(s) images and page settings in HTML documents. Basically, CSS is used to control the presentation of one or more web pages in your application. This unit designates the usage of cascading style sheets with JavaFX application. You are already aware of the XML; one similar language such as FXML will be covered in this unit. You will learn how to use FXML with JavaFX application. If you talk about the events which are object and it describes a state change in a source and, when you interact with elements or nodes of an application. For example, whenever you click on the button, select an item from drop-down menu, an event generates. For handling these events in JavaFx application, you will learn four stages such as Target selection, Route construction, Event capturing and Event bubbling in this unit. In event handling mechanism, a source generates an event and sends to one or more listeners. Once the listeners received this event, processes the event and then returns.

This unit continues the study of effects, animations and media in JavaFX. This Unit explains you how to apply effects on UI elements such as Blend, DropShadow and Glow. You have seen many animated movies or images. In this unit you will use animation with user interface controls. You will also include media like audio or video in your application. For incorporating media in an application, JavaFX API provides necessary classes such as Media, MediaPlayer and MediaView. The main use of this package is media playback. This unit utmost describes concepts with examples. The previous unit already explained to you how to run JavaFX application with IDE. So, this unit is not describing this running procedure again. You can run examples of this unit whether with IDE (Eclipse/NetBeans/IntelliJ Idea, etc.) or through the command prompt.

## 2.1 OBJECTIVES

At the end of this unit, you will be able to:

- ... describe how to create CSS and how to apply this upon UI control elements in JavaFX application,
- ... build User Interface with FXML,
- ... handle events in JavaFX application,
- ... apply effects, animations on the UI controls, and
- ... incorporate audio or video media in JavaFX application.

## 2.2 SKIN APPLICATIONS WITH CSS

You are already aware of the Cascading Style Sheets (CSS), which are used for adding style such as fonts, colors, backgrounds and spacing between the paragraphs in HTML documents. It is used to control the presentation of one or more web pages. This section defines how to use cascading style sheets with JavaFX applications. You can use cascading style sheets to create a custom look and feel for your JavaFX application.

Before going in detail, you may recall the JavaFX application structure, which contains three keys components, namely Stage, Scene and Nodes as shown in the following figure-1. A **stage** (or a window) embraces all the objects of a JavaFX application and it is represented by the Stage class of the package **javafx.stage**. The show() method can be used to display the contents of a stage. A **scene** graph is a data structure that is a collection of nodes, and it is represented by package **javafx.scene** in a JavaFX application. The scene class holds all the contents of a scene graph. A **node** is a basic graphical object of a JavaFX application such as UI control objects (button, radio checkbox, etc.), 2D and 3D geometric objects (circle, rectangle, etc.),

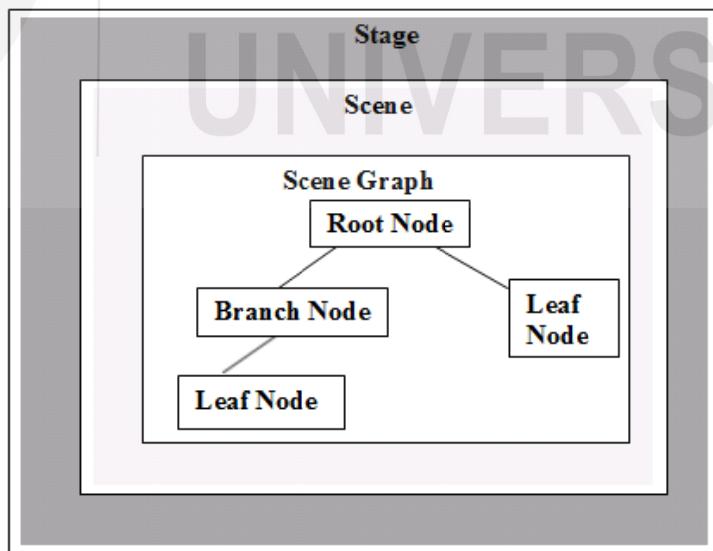


Figure 1: JavaFX application structure

Container/layout objects (Border Pane, Grid Pane, etc.), media element objects (audio, video etc.).

You can use CSS in JavaFX applications similar to use CSS in HTML web documents. JavaFX CSS are created on basis of W3C CSS specification (accessible at <https://www.w3.org/Style/CSS/Overview.en.html>) with some additional JavaFX features. You can refer JavaFX CSS reference guide at the link of Oracle web page (<https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>).

## JavaFX CSS

JavaFX provides a package `javafx.css` which contains the classes that are used to apply CSS in JavaFX applications. You can use CSS to enrich the look of the JavaFX application. A style sheet comprises rules that specify how formatting should be applied to the particular elements in your application. Each style rule contains two parts which are selector and declaration. The selector indicates which element(s) the declaration applies, and the declaration specifies the formatting properties of the element.

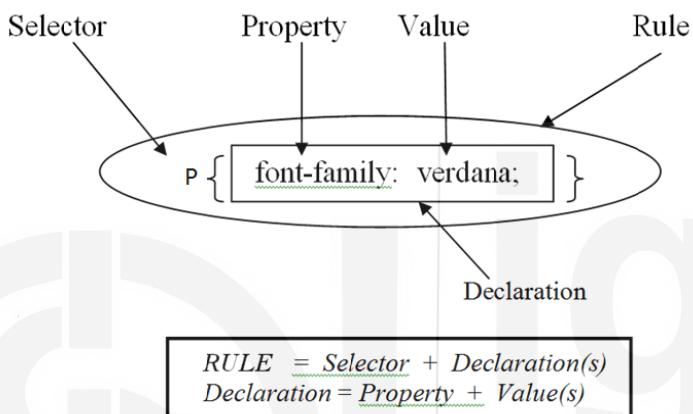


Figure 2: Format of CSS rules

JavaFX contains the new generation UI library, which is used to configure the theme of the application.

The JavaFX 8 application holds `caspian.css` as the default style sheet found in the JavaFX runtime JAR file (`jfxrt.jar`). The latest version of JavaFX application uses `Medona.css` as a style sheet that defines styles for the root node and the UI controls. There are several ways to apply a CSS style in a JavaFX UI controls such as JavaFX default CSS style sheet, Scene specific, Parent specific and Element specific style property. If you do not create your own style sheet, then JavaFX provides a default CSS style sheet that may be applicable to all JavaFX elements.

## Creating Style Sheets

You can generate your own cascading style sheets which override the styles in the default style sheet. You can create style sheet(s) with `.css` extension, and it is placed in the same directory of the main class for your JavaFX application. You can create your external Style Sheet or inline Style Sheet for the JavaFX applications.

When you design either your external Style Sheet or inline Style Sheet, you may follow the rules for the selection of property name as :

- ... You must define "-fx-" as a prefix with all JavaFX property names.
- ... If more than one word exists in the property name, then use the hyphen (-) to separate them.
- ... Property name and their value are separated by colon (:) .

- ... If more rules are defined for the Property name, then they are separated by a semicolon (;).

### CSS class Selector

You can write a dot (.) character followed by the Style class selector name. It is used to identify more than one element in an application. In the following example, the font property name is preceded by prefix -fx- and Property name and their value are separated by colon, and rules are separated by semicolon for the JavaFX label element.

```
.label
{ -fx-font : bold 12pt "arial"; -fx-padding:10 }
```

For accessing the above-mentioned style class which are appropriate to label node, you can use the `getStyleClass().add()` method.

### ID Styles

JavaFX provides us facility to create the style for the individual node. The style name can be given as the ID name preceded by the hash(#) symbol to select one unique element.

```
#submit
{
    -fx-font : bold 14pt "arial";
    -fx-background-color : #87ceeb;
}
```

You can use the above-mentioned style for the individual Submit button like the following way:

```
Button Submit=new Button ("Submit"); //submit button
Submit.setId("submit"); //set ID for the submit button
```

### Create your own Style Sheet

You can add your own external style sheet to a scene in JavaFX application like the following:

```
Scene sc = new Scene(root, 400, 200);
sc.getStylesheets().add("path/name_of_ownstylesheet.css");
```

If you want to set a CSS style sheet on a parent layout element then style sheet will be applied to all elements inside that layout element or child element. It is similar to setting style sheet on a Scene object. You can add your own external style sheet to a Parent element in JavaFX application like the following:

```
Button b1 = new Button("Submit");
Button b2 = new Button("Reset");
VBox vbox = new VBox(b1, b2);
vbox.getStylesheets().add("button_specific_styles.css");
```

Following example illustrate you how to create an own CSS for JavaFX application. This example displays Grade Card page which consists of controls such as label, text field and submit button. This example contains two files: OwnCSSEExample.java and Examplestyle.css. The source code for both files is listed below:

### **Example-1: Source code for OwnCSSEExample.java**

Graphical User Interface  
and Java Database  
Connectivity

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.text.*;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
public class OwnCSSEExample extends Application
{
    public void start(Stage stage) throws Exception
    {
        Label enroll=new Label("Enroll No."); //label creation for Enroll
        Label prg=new Label("Programme"); //label creation for programme

        //Creating Text Field for Enroll and programme
        TextField txtf1=new TextField();
        TextField txtf2=new TextField();

        Button Submit=new Button ("Submit"); //submit button
        Submit.setId("submit"); //set ID for the submit button

        GridPane gpane=new GridPane(); //creating grid pane

        //setting horizontal and vertical gaps between the rows
        gpane.setHgap(5);
        gpane.setVgap(5);

        Scene scene = new Scene(gpane,400,200);
        //adding the the nodes to the GridPane's rows
        gpane.addRow(0, enroll,txtf1);
        gpane.addRow(1, prg,txtf2);
        gpane.addRow(2, Submit);

        gpane.getStylesheets().add("Examplestyle.css"); //add CSS file to the grid pane
        stage.setTitle("Own CSS Example"); //set title to the Stage
        stage.setScene(scene); //set scene to the stage
        stage.show(); //showing contents of the stage
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

### **Source code for Examplestyle.css**

```
.label
{
-fx-font : bold 12pt "arial";
-fx-padding:10
}

#submit
{
-fx-font : bold 14pt "arial";
-fx-background-color : #87ceeb;
```

```
-fx-text-fill:red;  
-fx-padding: 8px 8px  
}
```

Now, you can compile and execute the OwnCSSExample.java file from the command prompt by using the following commands:

```
javac OwnCSSExample.java  
java OwnCSSExample
```

The output of the above example is shown in figure-3 like the following:

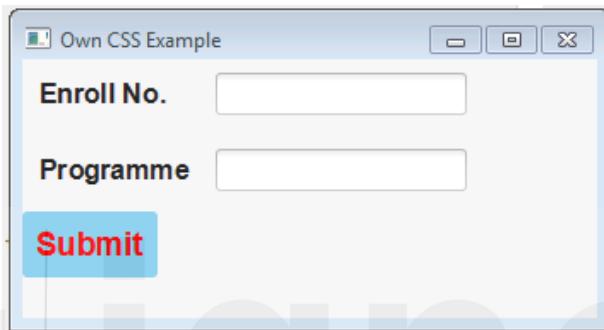


Figure 3: Output Screen for CSS creation example

### Create Inline Style Sheets

JavaFX allows us to describe the style rules in the JavaFX application code itself. You can insert inline styles in JavaFX applications by using setStyle() method. The inline style provides the finest level of control. These styles are applicable only for those nodes on which they are set. The following example illustrates you how to create an inline style sheet to a button.

```
Button b1 = new Button("Submit");  
b1.setStyle("-fx-background-color:yellow; -fx-text-fill: white;");
```

The following example illustrates inline style sheet rules in the source code file for some UI controls in JavaFX such as text field, password field and button. This example contains only one java file namely CssInStyleExample.java. The source code is listed below:

#### **Example-2: Source code for CssInStyleExample.java**

```
import javafx.application.Application;  
import static javafx.application.Application.launch;  
import javafx.geometry.*;  
import javafx.scene.text.*;  
import javafx.scene.*;  
import javafx.scene.control.*;  
import javafx.scene.layout.*;  
import javafx.stage.Stage;  
  
public class CssInStyleExample extends Application  
{  
    public static void main(String args[])  
    {  
        launch(args);  
    }  
}
```

```
}

public void start(Stage stage)
{
    Text txt1 = new Text("Name"); // label for Student Name
    Text txt2 = new Text("Password"); // label for Password
    Text txt3 = new Text("Programme"); // label for Programme

    // Text Field for Student
    TextField txtField1 = new TextField();

    // Text Field for password
    PasswordField txtField2 = new PasswordField();

    // Text Field for programme
    TextField txtField3 = new TextField();

    //Creating Buttons
    Button bt1 = new Button("Submit");
    Button bt2 = new Button("Reset");

    //Creating a Grid Pane
    GridPane gPane = new GridPane();

    //Set size for the pane
    gPane.setMinSize(400, 200);

    //Set the padding
    gPane.setPadding(new Insets(10, 10, 10, 10));

    //Set the vertical and horizontal gaps between the columns
    gPane.setVgap(5);
    gPane.setHgap(5);

    //Set the Grid alignment
    gPane.setAlignment(Pos.CENTER);

    //arrange all the nodes in the grid
    gPane.add(txt1, 0, 0);
    gPane.add(txtField1, 1, 0);
    gPane.add(txt2, 0, 1);
    gPane.add(txtField2, 1, 1);
    gPane.add(txt3, 0, 2);
    gPane.add(txtField3, 1, 2);
    gPane.add(bt1, 0, 3);
    gPane.add(bt2, 1, 3);

    //inline style for nodes
    bt1.setStyle("-fx-background-color:blue; -fx-text-fill: white;");
    bt2.setStyle("-fx-background-color: blue; -fx-text-fill: white;");
    txt1.setStyle("-fx-font: normal 15px 'arial' ");
    txt2.setStyle("-fx-font: normal 15px 'arial' ");
    txt3.setStyle("-fx-font: normal 15px 'arial' ");
    gPane.setStyle("-fx-background-color: lightblue;");

    // creating a scene object
    Scene scene = new Scene(gPane);
```

```

// Set Stage title
stage.setTitle("CSS InStyle Example");

// add scene to the stage
stage.setScene(scene);

//showing the contents of the stage
stage.show();
}
}

```

Now, you can compile and execute the CssInStyleExample.java file from the command prompt by using the following commands.

```

javac CssInStyleExample.java
java CssInStyleExample

```

The output of the above example is shown in following figure-4:



Figure 4: Output Screen for inline CSS creation example

## 2.3 BUILD UI WITH FXML

In the previous section, you have learned about the creation of cascading style sheets (CSS) and linking with JavaFX UI controls. This section describes to you how to create JavaFX user interfaces using FXML.

FXML is a XML-based user interface markup language for defining the user interface of a JavaFX application. It is created by Oracle Corporation and is another form of designing user interfaces by means of procedural code. Using FXML, you can write the UI controls code separate from the application logic.

### JavaFX FXML Example

You can write very simplest way to JavaFX FXML. Here is a FXML example that comprises a simple JavaFX Graphical User Interface.

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox>

```

```

<children>
    <Label text="Welcome Students in the FXML world "/>
</children>
</VBox>

```

The first line in the FXML document is similar to standard first line of XML documents. The next two lines are import statements which you want to import the classes to use in FXML. After the import statements you shall write the actual structure of the GUI. This example defines a VBox JavaFX layout component which contains a single Label as child element. The Label is used to display a text in the Graphical User Interface.

For the understanding of the concept of FXML through the below example which comprises three files:

- ... ApplicationFXML.java - It is the main JavaFX Application class which loads the FXML document using FXMLLoader.
- ... ControllerFXML.java – This file contains event handler code.
- ... FXMLfile.fxml - This FXML Document contains user interface for the application

#### **Example - 3: Source code for ApplicationFXML.java**

```

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ApplicationFXML extends Application
{
    public void start(Stage stage) throws Exception
    {
        Parent root = FXMLLoader.load(getClass().getResource("FXMLfile.fxml"));
        Scene sc = new Scene(root);
        stage.setScene(sc);

        // Set the Stage Title
        stage.setTitle("FXML Example");
        stage.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}

```

#### **Source code for FXMLfile.fxml**

```

<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

```

## Working with UI Controls

```
<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="ControllerFXML">

<children>
    <Button layoutX="126" layoutY="90" text="Click here!"
        onAction="#ButtonEvent" fx:id="button" />
    <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
        fx:id="label" />
</children>
</AnchorPane>
```

### Source code for ControllerFXML.java

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Label;

public class ControllerFXML
{
    @FXML
    private Label label;
    public void initialize() { }
    @FXML
    private void ButtonEvent(ActionEvent event)
    {
        label.setText("Welcome in SOCIS!");
    }
}
```

Now, you can compile both java files. When you run this project, it shall show a small window with a “Click here” button. After clicking on the button, it will show a welcome message.

When your main application class named as ApplicationFXML will run, the FXMLLoader will load FXMLfile.fxml. During loading FXMLfile.fxml, loader will find the name of the controller class which is specified by fx:controller="ControllerFXML" in the FXML file. Then it will call the controller's initialize method, and the code that is registered as an action handler with the button will be executed.

### ☛ Check Your Progress-1

1. What is JavaFX CSS? What are different ways to apply styles to a javaFX application? Explain with examples.
- 
- 
- 
-

2. Explain the differences between CSS class Selector and ID styles.
- 
- 
- 

Graphical User Interface  
and Java Database  
Connectivity

3. In JavaFX application, a key concept is the scene graph. Explain it with diagram.
- 
- 
- 

4. What is FXML? How to use FXML in JavaFX application, explain it with example.
- 
- 
- 

## 2.4 EVENT HANDLING IN JAVAFX

---

In the previous section 2.2, you have used UI control elements with CSS in JavaFX applications. This section describes events and the handling of events for these UI elements.

JavaFX API facilitates us to develop GUI based applications. Whenever you interact with such applications (nodes), an event is generated. For example, when a user clicks on a button, selects an item from a menu, then an event generates. You can say that events are notifications. In JavaFX, events are principally used to notify the application about the actions taken by the user. When you write GUI applications, then the program requires event handling mechanism for controlling the user's interaction with the GUI.

JavaFX provides a `javafx.event` package, which makes available for the basic framework of FX events. The `Event` class works as the base class for JavaFX events. Each event is associated with source, event target or listener and event type. In event handling mechanism, a source generates an event and sends it to one or more listeners. Once the listeners received this event, they process the event and then returns.

**Event source** is an object (e.g., a Button object) that generates an event (e.g., MouseEvent). Event is generated when the changes occurred in the internal state of that object. The source may create various types of events. The origin of the event implements the EventTarget interface.

**Listener** or targets object that listens for a particular type of event which is generated by the source. For the communication between listener and source, the listener must have been connected with the source and must implement methods to receive and process an interface like EventHandler.

There are various **events types** in JavaFX such as ActionEvent (e.g. button is pressed), MouseEvent (e.g. mouse activity), KeyEvent (e.g. keystroke has occurred), ScrollEvent (e.g. scrolling by mouse wheel) etc. You can define your own event by inheriting the class javafx.event.Event. These events are mainly categorised into two groups as Foreground Events and Background Events. **Foreground Events** are based on a user's direct interaction (e.g. button is pressed), while **Background Events** require end-user interaction such as hardware or software failure, timer expiry, etc. For more details, you can refer to the <https://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm>

## **Stages of Event Handling**

When a source generates an event, the JavaFX application goes through the four stages (Target selection, Route construction, Event capturing, and Event bubbling) for event handling. Before knowing the stages of event handling, you should know two important terms; Event Filters and Event Handlers.

### **Event Filter**

Event filter covers application logic to process an event. It can be used to process the events such as mouse actions, keyboard actions, scroll actions and other user interactions which are generated by your application. Event filters allow you to handle an event during the event capturing stage of event handling process. During the event capturing stage, a node must be registered with an event filter to process an event.

### **Event Handler**

An event handler is a function or method executed in response to an event, such as keystrokes and button clicks. It is used during the bubbling stage of event handling process. Node which is participated in the dispatch chain path can register to more than one filter/handler. You can define a common filter/handler to the parent, which is treated as a default for all the child nodes.

Following are stages of event handling:

#### **Target selection**

When any user interacts with UI controls, then an action occurs. At that moment, the system decides which node gets focus. For example, you click somewhere on the screen for the mouse event, the target node is found at the location of the cursor. If more than one node is found at the cursor point, then the uppermost node will be considered as the target.

#### **Route construction**

Whenever an event is generated, an event dispatch route is created in order to handle the events.

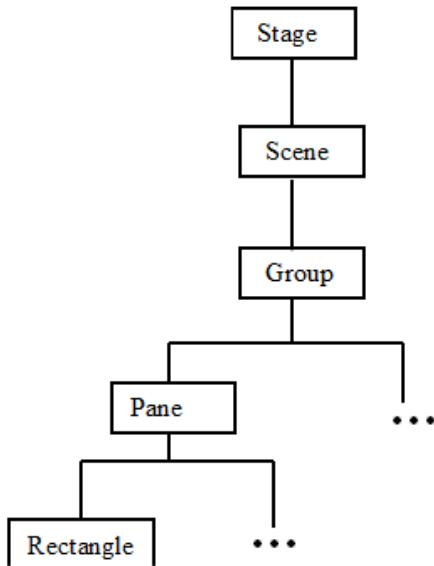


Figure 5: Event Dispatch Route

This dispatch route comprises the path from the stage to the node on which the event is generated. The Event route is regulated by the event dispatch chain which was formed in the operation of the buildEventDispatchChain() method of the selected event target.

### Event capturing

Once the event dispatch route is created, the event is transmitted from the source node. The generated event is traversed all the nodes in the route from top to bottom. If the event filter is registered with any of these nodes which are in the dispatch route, then event will be executed otherwise, it will be transferred to the target node. The target node processes the event in that case. You can see in event dispatch route, which are shown in Figure 5, the event moves from the Stage node to the Rectangle node during the event capturing phase.

### Event bubbling

When the event is captured and processed by the target node or registered filter, the event starts navigation all the nodes again from the bottom to the root node. If any of the node in the despatch chain has handler which is registered for the type of event happened then that handler is called, and it will get executed otherwise the process is returned to the next node up in the route and repeat the process. If a handler does not consume the event then the root node ultimately receives such event, and processing is finished.

You will find the example on event handling with animation feature in the next section.

---

## 2.5 EFFECTS, ANIMATION AND MEDIA

In the previous section, you have known about the event handling in javaFX application. This section describes you how to give effects and animations on the UI controls as well as about the integration of media in your application.

## Effects in JavaFX

You may be aware of the creation of the effects on graphics which are any action or changes that heightens the presence of the graphics in an application. The meaning of effects is functionally similar to the JavaFX application, wherein it is an algorithm that applies on nodes to visually enrich their appearance. The JavaFX API provides effect property of the **Node** class to specify the effect which is used to set various effects such as **Blend**, **Bloom**, **BoxBlur**, **ColorAdjust**, **ColorInput**, **DropShadow**, **GaussianBlur**, **Glow**, **ImageInput**, **InnerShadow**, **Lighting**, **MotionBlur**, **PerspectiveTransform**, **Reflection**, **SepiaTone**, and **Shadow** to a node. Each of these effects is denoted by a class, and all these classes are accessible by a package named **javafx.scene.effect.Effect**. This section describes you the effects of using two DropShadow and Reflection with examples. Here is not possible to define each effect with examples.

### Apply Effect to JavaFX Node

You can apply effects on any JavaFX UI control using **setEffect()** method, which needs to be called through a node object. The following example is showing DropShadow effect on Rectangle shape. Here, color code (FFFF00) representing the yellow color. JavaFX provides **javafx.scene.effect.DropShadow** class is used for providing dropshadow effect on UI controls. The parameters of DropShadow class are Radius, X Offset, Y Offset and Color like the following. These parameters are optional.

```
Rectangle.setEffect(new DropShadow(1, 20, 30, Color.web("#FFFF00")));
```

### DropShadow Effects

The following example-4 demonstrates you how to apply DropShadow effects to a JavaFX UI control. For this you can create an application which has a Rectangle shape and put DropShadow effect on the shape. The source code is listed below:

#### Example- 4 : Source code for EffectsJavaFX.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
public class EffectsJavaFX extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    public void start(Stage stage)
    {
        Rectangle r = new Rectangle(100,100, Color.RED);
        r.setEffect(new DropShadow(1, 20, 30, Color.web("#FFFF00")));
        Scene scene = new Scene(new Pane(r), 300, 250);
        stage.setScene(scene);
        stage.setTitle("Effects Example");
        stage.centerOnScreen();
        stage.show();
    }
}
```

Now, you can compile and execute the above program from the command prompt. The output of the program is display as shown in figure-6:

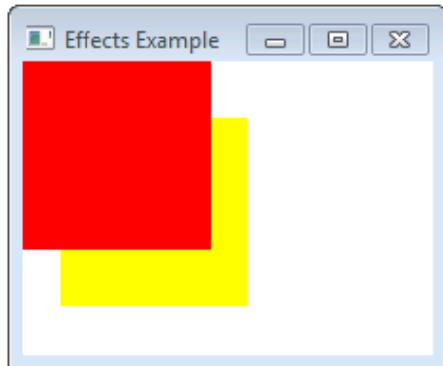


Figure 6: Example for DropShadow Effects

## Reflection Effects

The JavaFX provides mirror-like reflection effect using `javafx.scene.effect.Reflection` class. It takes the parameters such as `topOffset`, `topOpacity`, `bottomOpacity`, `fraction`, which affect resulting reflection to a JavaFX Node.

The following example-5 demonstrates you how to apply Reflection effects to a JavaFX UI control.

### Example-5: [Source code for ReflectionEffects.java](#)

```
import javafx.application.Application;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.effect.Reflection;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;
public class ReflectionEffects extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    public void start(Stage stage)
    {
        Text txt = new Text("Working with UI Controls");
        txt.setLayoutX(30);
        txt.setLayoutY(20);
        txt.setTextOrigin(VPos.TOP);
        txt.setFont(Font.font("Arial", FontWeight.BOLD, 40));
        Reflection ref = new Reflection();
        ref.setTopOffset(0);
        ref.setTopOpacity(0.75);
        ref.setBottomOpacity(0.10);
        ref.setFraction(0.7);
        txt.setEffect(ref);
        Scene scene = new Scene(new Pane(txt), 425, 175);
```

```
stage.setScene(scene);
stage.setTitle("Reflection Effects Example");
stage.show();
}
```

Now, you can compile and execute the above program from the command prompt. The output of the program is displayed as shown in figure-7:



Figure 7: Example for Reflection Effects

## Animation in JavaFX

You have seen many animated graphics which have shown the illusion of its motion using the rapid display. You can create an animated node in JavaFX application by changing its property over time. You can apply animations or transitions such as **Fade Transition**, **Fill Transition**, **Rotate Transition**, **Scale Transition**, **Stroke Transition**, **Translate Transition**, **Pause Transition**, **Parallel Transition**, **Path Transition**, **Sequential Transition** etc. All these transitions are represented by individual classes in the package `javafx.animation`.

**Translate transition** is used to create movement/transition from one point to another point within a defined duration. It is done by keeping changing the `translateX` and `translateY` properties of the node at regular intervals. In JavaFX, this animation is represented by the class `javafx.animation.TranslateTransition`. The example-6 is showing Translate transition feature of animation. Using **Rotate transition** feature, you can rotate an object. You can set rotation using `'toAngle'` and `'byAngle'` method. The `'toAngle'` indicates what angle the node should rotate, and `'byAngle'` indicates how much it should rotate from the current angle of rotation. For more details and example, you can refer to the Oracle JavaFX document at the link (<https://docs.oracle.com/javafx/2/animations/basics.htm>).

To incorporate animations in JavaFX application, you may follow the below steps

- ... Create a node using their class.
- ... Instantiate the animation class that is to be used
- ... Set the properties of the animation
- ... To complete this application, play the animation using the `play()` method of the Animation class.

The following example illustrates you how to apply '**Translate Transition**' animation on a Rectangle node with event handling. For this you can create an application which has a Rectangle shape and two Play and Pause buttons for change the running position of the shape. The source code is listed below:

### **Example-6: Source code for AnimationNEventHandler.java**

```
import javafx.animation.TranslateTransition;
```

```

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.*;
import javafx.scene.control.Button;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;
public class AnimationNEventHandler extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    public void start(Stage stage) throws Exception
    {

        Rectangle r = new Rectangle(100,100); //Create Rectangle
        r.setFill(Color.BLUE); //set the color of Rectangle

        //create play button and set coordinates
        Button btn1 = new Button("Play");
        btn1.setTranslateX(100);
        btn1.setTranslateY(150);

        // creating pause button and set coordinate
        Button btn2 = new Button("Pause");
        btn2.setTranslateX(145);
        btn2.setTranslateY(150);

        //Instantiating TranslateTransition class to create animation
        TranslateTransition trans = new TranslateTransition();

        //set attributes for the TranslateTransition
        trans.setAutoReverse(true);
        trans.setByX(200);
        trans.setCycleCount(100);
        trans.setDuration(Duration.millis(600));
        trans.setNode(r);

        //Create EventHandler
        EventHandler<MouseEvent> handler = new EventHandler<MouseEvent>()
        {
            public void handle(MouseEvent event)
            {
                if(event.getSource()==btn1)
                {
                    trans.play(); //animation will play when click on the play button
                }
                if(event.getSource()==btn2)
                {
                    trans.pause(); //animation will pause when click on the pause button
                }
                event.consume();
            }
        };
    }
}

```

```

};

//Add Handler for the play and pause button
btn1.setOnMouseClicked(handler);
btn2.setOnMouseClicked(handler);

//Create Group and scene
Group root = new Group();
root.getChildren().addAll(r,btn1,btn2);
Scene scene = new Scene(root,200,100,Color.TRANSPARENT);
stage.setScene(scene);
stage.setTitle("Animation with Event Handling");
stage.centerOnScreen();
stage.show();
}
}

```

Now, you can compile and execute the above program from the command prompt. The output of the program is displayed as shown in figure-8

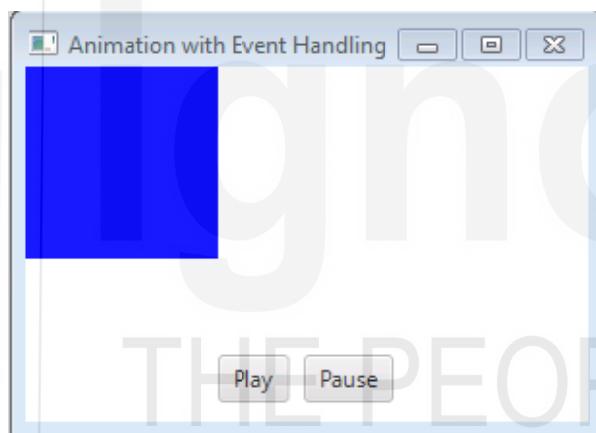


Figure 8: Output screen for the example of event handling with animation

## Media in JavaFX

You have seen many rich internet applications (RIA) on the internet or even used such applications like Google Maps and Google Docs. JavaFX provides media API that enables you to integrate audio and video into rich internet applications. The package **javafx.scene.media** covers all the essential classes for making an attractive, interactive JavaFX application. It consists of three prime classes such as **Media**, **MediaPlayer** and **MediaView**. The package is used for media playback.

The **Media** class denotes a media resource which could be an audio or a video, whereas **MediaPlayer** class is used for playing media. The **MediaPlayer** class is used in association with the **Media** and **MediaView** classes to display and control media playback. **MediaPlayer** provides media-controlling methods such as `pause()`, `play()`, `stop()` and `seek()` which apply to all types of media. **MediaView** arranges a view of **Media** being played by a **MediaPlayer**.

It is very easy to incorporate video or audio in JavaFX application. For integrating media in JavaFX, you can use the following steps:

- ... You can instantiate **javafx.scene.media.Media** class by passing the path of the video or audio file in its constructor like the following:

```
Media media = new Media("source of video/audio");
```

- ... Now, you can pass the media class object to the new instance of `javafx.scene.media.MediaPlayer` object like the following:

```
Mediaplayer player = new MediaPlayer(media);
```

- ... Invoke the `MediaPlayer` object's by `play()` method

```
player.play();
```

- ... You can instantiate `MediaView` class and pass `player` object into its constructor like the following:

```
MediaView mediaView = new MediaView (mediaPlayer)
```

- ... Add the `MediaView` object and configure Scene.

```
Scene scene = new Scene(new Pane(mView), 500, 400);
stage.setScene(scene);
stage.setTitle("Video Example");
stage.show();
```

Following example shows you how to run video file in a javaFX application. Demonstrated Video in this example which is available on the YouTube ([https://www.youtube.com/watch?v=CtwJ1Mgf\\_ic](https://www.youtube.com/watch?v=CtwJ1Mgf_ic)) or you can use your own video for running this example.

#### **Example-7: Source Code for ExampleMedia.java**

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;
import java.io.File;
public class ExampleMedia extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    public void start(Stage stage) throws Exception
    {
        File mFile = new File("Computer.MP4");

        //Instantiating Media class
        Media media = new Media(mFile.toURI().toString());

        //Instantiating MediaPlayer class
        MediaPlayer player = new MediaPlayer(media);
```

## Working with UI Controls

```
//Instantiating MediaView class  
MediaView mView = new MediaView(player);  
player.play();  
  
//configure scene  
Scene scene = new Scene(new Pane(mView), 500, 400);  
stage.setScene(scene); // Setting the scene to stage  
stage.setTitle("Video Example");  
stage.show();  
}  
}
```

Now, you can compile and execute the above program from the command prompt. The output of the program is display as shown in figure-9.



Figure 9: Example for video incorporating in JavaFX application

The above example is for video including in JavaFX application. In a similar way, you can include your audio also.

## ☛ Check Your Progress-2

1. Explain the term event. What steps are needed to be followed in order to handle the events?

---

---

---

---

2. What is the difference between foreground and background events?

---

---

---

3. Give names of animation in JavaFX. Explain any two of them with example.

## 2.6 SUMMARY

This unit explained about the JavaFX UI Controls, which are used in different perspectives such as creating and linking with CSS, building UI with FXML, event handling, effects, animation and media incorporating with UI. Also, in this unit it is explained that how to create and apply CSS on UI control elements with examples. You have also learnt how to build javaFX UI with FXML. The FXML is a XML based language, and it is used for constructing Java object graphs. FXML offers an appropriate alternative to constructing such graphs in procedural code. You also got knowledge about events handling, and it is used when users interact with UI controls.

This unit explained about giving effects and animation to JavaFX UI controls elements. At the end of this unit, you have learnt about the media integration in JavaFX application using API that enables you to integrate audio and video into the applications. The package `javafx.scene.media` covers all the essential classes for making JavaFX applications. This package contains three main classes such as `Media`, `MediaPlayer` and `MediaView`. The package is used for media playback. Now you are able to work with UI controls.

## **2.7 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS**

## ☛ Check Your Progress 1

- 1) CSS stands for Cascading Style Sheets which are used for adding style such as fonts, colors, backgrounds and spacing between the paragraphs in HTML documents. It is used to control the presentation of one or more web pages. JavaFX CSS is based on the W3C CSS with some additional features. The objective of JavaFX CSS is to allow developers (who are already at ease with CSS for HTML) to use CSS for customizing JavaFX controls. There are two ways to add CSS to JavaFX application by your own Style Sheet and/or inline Style Sheet.

You can add your own external style sheet to a scene in JavaFX application like the following:

```
Scene sc = new Scene(root, 400, 200);  
sc.getStylesheets().add("path/name of stylesheet.css");
```

Using inline style, you can set CSS styles for a specific element by setting the CSS properties directly on the element by using the `setStyle()` method, for example, inline style sheet for a button.

```
Button button = new Button("Reset");
button.setStyle("-fx-background-color: #FFbbbb");
For more details, you can refer section 2.2 in this Unit.
```

- 2) The difference between an ID and a class selector is that an ID is only used to identify a single element in javaFX application whereas class selector is used to identify more than one element. The IDs are only used when one element in the web page should have a particular style applied to it. The name of class selector is preceded by dot(.) character, while the name of ID is preceded by the hash(#) symbol to select the element.
- 3) A scene graph is a data structure that denotes the contents of a window. The scene graph consists of nodes that show graphical components in the window, such as buttons, radio buttons and checkbox. A scene graph is represented by package `javafx.scene` in a JavaFX application. The scene class holds all the contents of a scene graph. Some of the nodes behave as containers that contains other nodes. The GUI in a window is created by building a scene graph. For more details, you may refer section 2.2 of this unit.
- 4) FXML is an XML-based user interface markup language for defining the user interface of a JavaFX application. It is created by Oracle Corporation. It is another form of designing user interfaces using procedural code. You may refer to section 2.3 of this unit for viewing the example.

## ☞ Check Your Progress 2

- 1) An event is an object which describes a state change in a source. It generates whenever a user interacts with UI control elements. It is generated by clicking a mouse, pressing a button, entering a character using the keyboard. It can also generate by without human interaction, such as timer expires, a software or hardware failure etc.

The event handling consists of four stages: target selection, route construction, Event capturing, and bubbling. In the target selection phase, when any user interacts with UI controls then an action occurs, at that time the system decides which node gets focus; this happens in target selection phase. Whenever an event is generated, an event dispatch route is created to handle the events in the stage of route construction. This dispatch route comprises the path from the stage to the node on which the event is generated. The event route is regulated by the event dispatch chain which was formed in the operation of the `buildEventDispatchChain()` method of the selected event target.

In Event capturing stage, the event is transmitted from the source node. The generated event is traversed all the nodes in the route from top to bottom. If the event filter is registered with any of these nodes which are in the dispatch route then event will be executed otherwise, it will be transferred to the target node. The target node processes the event in that case. When the event is captured and processed by the target node or registered filter, the event starts navigationing all the nodes again from the bottom to the root node. If any of the node in the dispatch chain has handler which is registered for the type of event happened then that handler is called, and it will get executed otherwise the process is returned to the next node up in the route and repeat the process. If a handler does not consume the event, then the root node ultimately receives such event and processing is finished. These happen in the event bubbling phase.

- 2) Foreground Events are based on the direct interaction of a user (e.g. button is pressed), while Background Events involve the interaction of end-user such as hardware or software failure, timer expiry. Foreground Events are created by a person directly interacting with the graphical components in a GUI. For example, double-clicking on a button, moving the mouse, input character through a keyboard, selecting an item from the menu, scrolling the page, etc. The hardware or software failure, operation completion interruptions are the example of background events.
- 3) The names of animations are Fade Transition, Fill Transition, Rotate Transition, Scale Transition, Stroke Transition, Translate Transition, Path Transition, Sequential Transition, Pause Transition, Parallel Transition, etc. All these transitions are represented by individual classes in the package javafx.animation.

**Translate transition** is used to create movement/transition from one point to another point within a defined duration. It is done by keep changing the translateX and translateY properties of the node at the regular interval. In JavaFX, this animation is represented by the class javafx.animation.TranslateTransition. Using **Rotate transition** feature, you can rotate an object. You can set rotation by using ‘toAngle’ and ‘byAngle’ method. The ‘toAngle’ indicates what angle the node should rotate, and ‘byAngle’ indicates how much it should rotate from the current angle of rotation.

## 2.8 FURTHER READINGS

- ... Carl Dea, Gerrit Grunwald, José Pereda, Sean Phillips and Mark Heckler “JavaFX 9 by Example”, Apress,2017.
- ... Sharan, Kishori. Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs. Apress, 2014.
- ... <https://docs.oracle.com/javase/8/javafx/get-started-tutorial/index.html><https://www.w3.org/Style/CSS/Overview.en.html>
- ... <https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>
- ... [https://docs.oracle.com/javafx/2/css\\_tutorial/jfxpub-css\\_tutorial.htm](https://docs.oracle.com/javafx/2/css_tutorial/jfxpub-css_tutorial.htm)
- ... [https://docs.jboss.org/richfaces/latest\\_4\\_5\\_X/Developer\\_Guide/en-US/html/chap-Developer\\_Guide-Skinning\\_and\\_theming.html](https://docs.jboss.org/richfaces/latest_4_5_X/Developer_Guide/en-US/html/chap-Developer_Guide-Skinning_and_theming.html)
- ... [https://docs.oracle.com/javafx/2/get\\_started/fxml\\_tutorial.htm](https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm)
- ... [https://docs.oracle.com/javafx/2/api/javafx.fxml/doc-files/introduction\\_to\\_fxml.html](https://docs.oracle.com/javafx/2/api/javafx.fxml/doc-files/introduction_to_fxml.html)
- ... <https://docs.oracle.com/javafx/2/events/processing.htm>
- ... <https://docs.oracle.com/javafx/2/events/filters.htm>
- ... <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/effect/Effect.html>
- ... <https://docs.oracle.com/javase/8/javafx/get-started-tutorial/animation.htm>
- ... <https://docs.oracle.com/javafx/2/api/javafx/scene/media/package-summary.html>

---

## **UNIT 3     JDBC PART 1**

---

### **Structure**

- 3.0 Introduction
- 3.1 Objectives
- 3.2 JDBC Introduction
  - 3.2.1 Core JDBC components
- 3.3 JDBC Driver
  - 3.3.1 Type 1/JDBC-ODBC Bridge
  - 3.3.2 Type 2/Java to Native API
  - 3.3.3 Type 3/ Java to Network Protocol
  - 3.3.4 Type 4/ Java to Database Protocol
- 3.4 JDBC Database Connection Steps
- 3.5 JDBC SQLEXCEPTION
- 3.6 JDBC Driver Manager Class
- 3.7 JDBC Connection Interface
  - 3.7.1 Statement Interface
  - 3.7.2 PreparedStatement Interface
  - 3.7.3 CallableStatement Interface
- 3.8 Summary
- 3.9 Solutions/ Answer to Check Your Progress
- 3.10 References/Further Reading

---

### **3.0     INTRODUCTION**

---

In this unit, we will discuss the Database Connectivity concept for Java applications. We know, that we can save our data in the file system, but it doesn't offer any capability to querying on data efficiently. At the same time, we know that the various databases like Oracle, MySQL, Sybase etc., provide file-processing capabilities and facilitates efficient query processing. We will explore the Java Database Connectivity (JDBC) Application Program Interface (API) and the interaction with a database using the JDBC feature of Java.

We can write Java applications using JDBC to manage different programming activities, starting from establishing a database connection and then sending a query and retrieving result updates from the database. In general, using JDBC API, we can create the table, update and insert values, fetch records, create prepared statements, perform transactions and catch exceptions.

We can visualize the working of the JDBC API where we are writing a typical Java program and using standard library routines. First, a database connection has been established; subsequently, we use JDBC to query the database. After processing, the result is returned to the application, and finally, we close the connection.

In this unit, we will also explore various JDBC components like Driver, Driver Manager, Connection, Statement etc., which are used for database interaction. A JDBC Driver is used to open the database connection for query execution and receiving results with the Java application. The Driver Manager ensures the transparent access of each database using the correct Driver, which is specific to that database. The Connection interface creates the session between the application and the database and, finally, the SQL query execution and subsequent fetching of results through some pre-defined libraries stored as Statement object, PreparedStatement object and CallableStatement object.

---

### 3.1 OBJECTIVES

---

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

- ... write the steps for database connectivity within the Java application using JDBC,
  - ... write Java application to execute SQL query and fetch the results using JDBC API,
  - ... explain various JDBC components required for database interactions,
  - ... use drivers to open the connection for the execution of database commands ,
  - ... use Driver manager, which loads all the system drivers and select the most appropriate Driver, and
  - ... use Connection interface, which offers different methods for database communication
- 

### 3.2 JDBC INTRODUCTION

---

JDBC API used provide database connectivity between Java applications and various databases. It defines how to access Relational Database (RDBMS) through standard interfaces and classes. Precisely JDBC API helps us to write Java applications that can access tabular data.

JDBC was created by Sun Microsystems and is SQL-based API to access databases, but now it is own by Oracle. JDBC has been developed as an alternative to the Open Database Connectivity (ODBC), a C-based API created by Microsoft and is used to access databases via SQL.

JDBC API allows the Java application to access various types of databases like Oracle, MS Access, Sybase, MySQL and SQL Server etc. It offers the platform-independent interface between the Java application and the RDBMS and allows the application to execute SQL statements. Various tasks are associated with database usage, such as establishing a connection, query creations, query executions, and viewing and modifying the resulting records. Primarily, JDBC allows mobile access to an underlying database, and for seamless performance, it provides a complete set of interfaces.

To better understand the concept, let us first discuss the Java database connectivity architecture. The JDBC Architecture provisions both 2-tier and 3-tier models but in general, it consists of two layers where through JDBC API, the first layer manages the application-to-JDBC Manager connection and the other layer JDBC Manager-to-Driver Connection.

The two-tier architecture provides direct communication of Java applications with the database where the JDBC driver facilitates communication with a specific database. The Driver receives a request from the application and transforms it into a vendor-specific database call, which is passed to the database. This configuration is termed the client-server architecture, where the client is the user machine, and the database acts as a server. This architecture facilitates the user to send the query to the database, and results are retrieved back by the user.

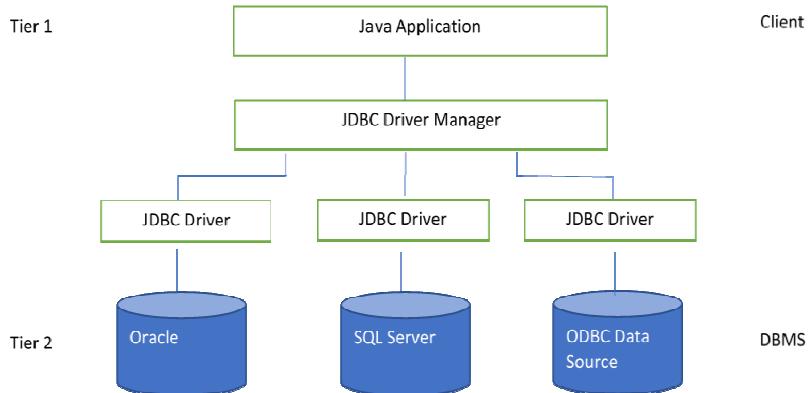


Figure 3.1 Two-tier architecture

In the three-tier architecture, the middle tier is used to take care of the business logic. In this architecture, the user machine sends a command to the middle-tier, which directs it for processing to the database. After processing the commands, database sends back the results to the middle tier, and subsequently, middle-tier sends it to the user machine. The JDBC API is used in the Application business layer of a three-tier architecture.

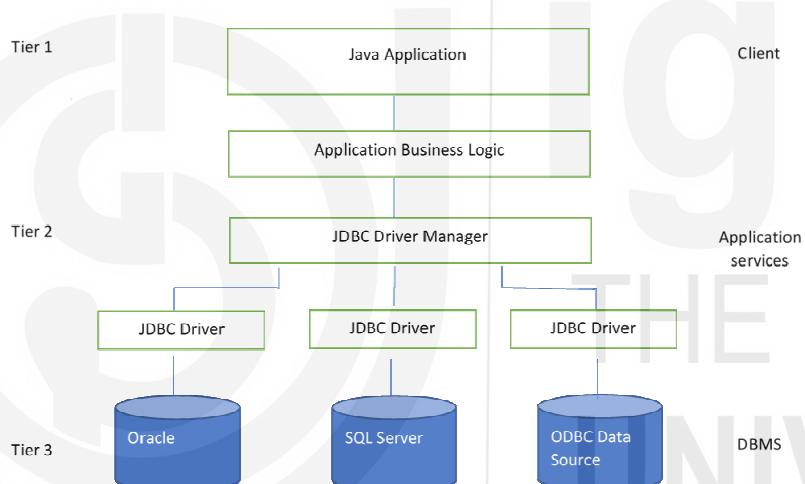


Figure 3.2 Three-tier architecture

The Java application communicates with a database, and the JDBC API facilitates the Java application to execute the SQL queries. The application then retrieves the processed results in a consistent and database independent manner. It uses a driver manager and database-specific Driver and ensures that the driver manager uses the correct Driver to access each database. The JDBC API provides various classes and interfaces that represent objects for this purpose. It includes Connection interface, Statements, PreparedStatement, CallableStatements Interface, ResultSet, Metadata Interface, BLOBS, CLOBS, Database drivers, DriverManager etc.

The JDBC API uses a Driver Manager class to ensure the transparent access of each database. The transparent access to a database by facilitating the correct Driver specific to that database. The JDBC drivers enable the Java applications to make use of different operating systems and hardware platforms.

The API technology provides the standards for Java applications to use the database independently due to its WORA capabilities (“Write Once, Run Anywhere”). Database independence is achieved by using various methods and interfaces that set

up an accessible communication with the database. The application can thus execute queries, retrieve results and update data. JDBC API is part of the Java Standard Edition (SE) and Java Enterprise Edition (EE) platform and provides `Java.sql.*` and `Javax.sql.*` packages.

### 3.2.1 Core JDBC Components

JDBC consists of the following core components through which it can interact with a database:

1. JDBC Driver Manager
2. JDBC Driver
3. Connection Interface
4. Statement Interface
5. ResultSet Interface

**JDBC Driver manager:** It is the main class that defines an object which is used to connect the application to a JDBC driver. The Driver Manager manages various types of drivers running on an application. The primary responsibility is to correctly loads all the system drivers and select the suitable Driver for opening a database connection. There are Standard Extension packages `Javax.naming` and `Javax.sql` which are used to create the database connection. We can make use of any package for the connection, but it is recommended to use a Data Source object registered with a Java Naming and Directory Interface™ (JNDI) naming service.

**Driver:** A JDBC driver opens the database connection for the execution of database commands and receiving results with the Java application. A JDBC driver establishes a connection with a database and subsequently send an update and retrieve queries. The JDBC API requires drivers for each database and ensures database independence.

**Connection Interface:** The connection interface offers different methods for database communication, and all the database communication is possible only through the connection object.

**Statement Interface:** The statement interface object submits a SQL query to the database. It provides two sub-interfaces `PreparedStatement` and `CallableStatement`. These interfaces execute queries and fetch the processed result of the executed SQL queries.

**ResultSet Interface:** The `ResultSet` interface provides the outcome of a query. The `ResultSet` object maintains a cursor that points at the current row in the result set data. The result set refers to the row and column data possessed by a `ResultSet` object of the underlying database.

---

## 3.3 JDBC Driver

---

A JDBC driver is a software component that translates standard JDBC calls into a database library API call and enables interaction of the Java application with the database. The JDBC drivers open the database connection to execute database commands and receive results with the Java application. The advantage of using these drivers is to ensure database independence, which implies that we can easily change the backend database by only changing the database driver and a few lines of code.

There are four different types of JDBC drivers having diverse implementations. Let us discuss these drivers:

### 3.3.1 Type 1/JDBC-ODBC Bridge

The JDBC-ODBC bridge driver acts as a bridge where ODBC drivers are used for accessing JDBC. This bridge driver connects to the database by converting JDBC method calls into ODBC function calls. First, the Java statement transformed into a JDBC statement which subsequently calls the ODBC. This transformation is done by using the Type-I Driver and, finally, the query execution. Oracle does not support this Driver, and this JDBC-ODBC bridge driver is removed from Java 8 onwards.

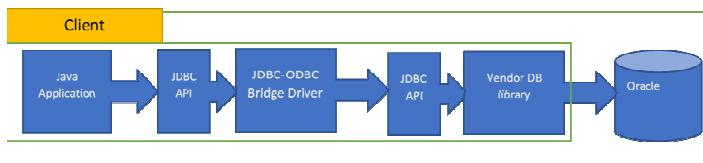


Figure 3.3 JDBC-ODBC bridge driver

**Advantage:** The JDBC-ODBC Bridge driver was easy to use and offered easy connectivity with any database.

**Disadvantage:** The Driver was required to install on the client machine.

### 3.3.2 Type 2/Java to Native API

The Java to Native API driver uses the Java Native Interface (JNI) to make calls to a local database library API. The Type 2 driver connects with the database by converting the JDBC method calls into a native call of the database API. It requires some native code to communicate directly with the database server, but not entirely written in Java.

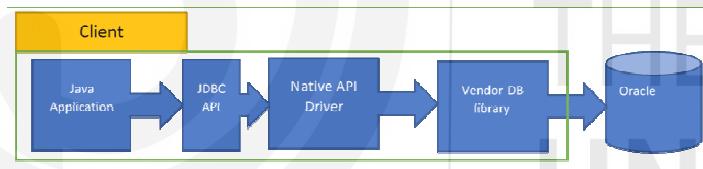


Figure 3.4 Java to Native API driver

**Advantage:** The Type 2 driver offers better performance than the JDBC-ODBC Driver.

**Disadvantage:** The native Driver and the vendor library need to install on every client machine. Also, we need to change the native API if we change the database.

### 3.3.3 Type 3/ Java to Network Protocol

The Network Protocol driver connect with the middleware using a three-tier approach. The Driver connects with the database by converting the JDBC method calls into the vendor-specific database protocol. It is also called an All-Java Driver as entirely written in Java (pure Java drivers).

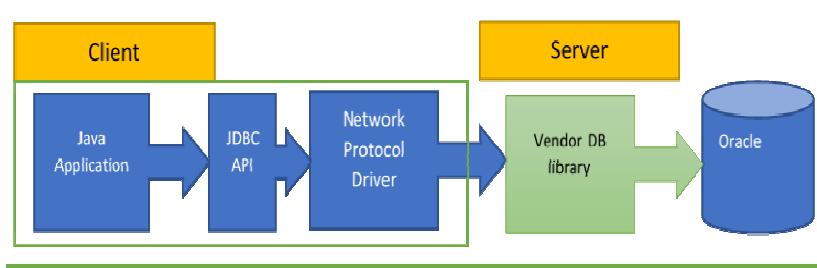


Figure 3.5 Java to Network Protocol driver

**Advantage:** The Type 3 driver offers a flexible JDBC solution as the native database libraries are not required to install on the client-side. Another advantage of this Driver is that a single driver can be used to access multiple databases.

**Disadvantage:** The maintenance of the Type 3 driver is costly as the coding at the middleware is database-specific. The server performs various tasks like load balancing, logging etc., but it will require to have network support on the client machine.

### 3.3.4 Type 4/ Java to Database Protocol

The Thin Driver is the fastest pure Java driver and communicates directly through socket connections without requiring any middleware or native library. The Driver connects with the database by converting the JDBC method calls into the vendor-specific database-protocol directly. This Driver converts the Java statements to SQL statements directly without the need of any intermediate library, and so it is also called a thin driver.

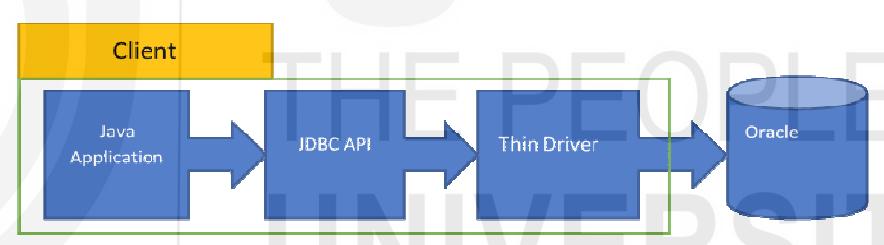


Figure 3.6 Thin driver

**Advantage:** These drivers are deployed online without requiring any software installation on the client or server.

**Disadvantage:** The Type 4 driver is database dependent which means if our backend database changes, we need to deploy a new driver compatible with the database.

Generally, the choice of the Driver depends on many factors like the requirements of the project, cost, flexibility, interoperability, performance etc. Type 1 and Type 2 driver are available mostly free of charge, but they require software installation and software configuration on each client. Type 3 and Type 4 are not open-source. Still, the Type 3 driver offers access to different types of databases, i.e. it is suitable for the application which requires access to multiple databases at the same time. In contrast, Type 4 drivers usually provide better performance.

### ☛ Check Your Progress 1

1. Briefly explain the necessary steps to create a JDBC application?

- 
- 
- 
- 
2. Which JDBC driver is the fastest Driver, and why?
- 
- 
- 
- 

3. How many packages are available in JDBC API?
- 
- 
- 
- 

---

### 3.4 JDBC DATABASE CONNECTION STEPS

---

Let us first discuss and explore the various components used in the JDBC Database Connection steps. The JDBC API provides a set of interfaces and classes to establish a connection with the database, create and execute SQL queries, to fetch the results and processing that ResultSet.

JDBC Steps for connecting a Java application with the database can be summarized as follows:

- ... Load the Driver
- ... Create a connection
- ... Create Statement
- ... Query execution
- ... ResultSet processing
- ... Close connection

The JDBC provides driver interface, a Connection interface, Statement interface, and ResultSet interface for the execution.

**Step 1: Loading the Driver:** The JDBC drivers are acting as a gateway to a database. The first thing we need to do is to initialize a driver before its usage in the program. We need to load the Driver or register the Driver once to open a communication channel with the database. For MYSQL JDBC Driver, we can download MySQL-connector-Java-8.0.21-bin.jar (current), extract the .jar file and add its path into \$CLASSPATH.

There are two ways through which we can register the Driver.

1. **DriverManager.registerDriver():** This is an inbuild Java class with register as its static member and will call the driver class constructor at the compile time.

To register the MYSQL Driver, we can use the following syntax

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver())
```

2. **Class.forName():** This method will dynamically load the Driver's class into memory at the runtime.

```
Class.forName("com.mysql.jdbc.Driver");
```

Generally, the JVM automatically loads the classes used in the program. But, when the driver class is not used explicitly, we need to explicitly inform the JVM to load the driver class to be used in the program. Since Java version 6 onwards, the JDBC driver is not required to explicitly load; instead, we just need to have the appropriate jar in the classpath. For this reason, that driver loading step is very much non-compulsory from Java 6 onwards. However, it is highly recommended that you check whether automatic loading is available with the Driver used in the application.

**Step 2: Create the connections:** After we load the Driver, it is required to open a connection which is done by creating a connection object through the static getConnection() method. The connection object establishes a physical connection with the database of the DriverManager class.

```
Connection jdbcconn = DriverManager.getConnection(URL, user, password)
```

jdbcconn is a reference to Connection interface, "user" is the username used to access the SQL command prompt, "password" is the password used to access the SQL command prompt, and URL is Uniform Resource Locator. We can define it as follows:

```
URL = "jdbc:mysql://localhost:3306/emp", user = "root" and password = "admin"
```

We can connect with the database using the credentials as user "root" with password "admin", and the schema is emp through the port number 3306 of host localhost (or we may use the address of database server).

**Step 3: Create a statement:** After establishing a connection to interact with the database, the createStatement() method is used to send commands to the database. The Statement object subsequently used to execute the query.

The JDBC Statement used is as follows:

```
Statement jdbcstmt = jdbcconn.createStatement();
```

jdbcconn is a reference to the Connection interface

**Step 4: Query Execution:** The query execution to retrieve the result from the database, we can use the executeQuery() method of the Statement class. There are multiple types of queries in the database like an update, insert or retrieval of data. This executeQuery() method returns the ResultSet object, which is used to fetch the records of a table.

The executeUpdate(SQL query) method of Statement class is used to execute update or insert query. It returns an integer that represents the number of affected rows by the SQL statement.

The JDBC Statement to create a table is as follows:

```
jdbcstmt.executeUpdate("CREATE TABLE EMP (id INTEGER not NULL, first
VARCHAR(30), last VARCHAR(30), age INTEGER, city VARCHAR(30), salary
INTEGER, PRIMARY KEY ( id ))");
```

The JDBC Statement to get the data is as follows:

```
ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from Emp");
```

**Step 5: ResultSet Processing:** After query execution, we have the ResultSet, and the data access through the ResultSet object over a cursor. The cursor is a pointer and initially placed before the first row. Now, using the next() method, it can advance to the subsequent rows of the ResultSet. There are different getter methods of the ResultSet are available to access the values of the current row, depending on datatypes.

For example, to access the records of all the Employees the ResultSet can be iterated like:

```
while(jdbcresultset.next())
{
    System.out.println("id : " + jdbcresultset.getInt("id") + " First : " +
    jdbcresultset.getString("first") + " Last : " + jdbcresultset.getString("last") + " Age : " +
    + jdbcresultset.getInt("age") + " City : " + jdbcresultset.getString("city") + " Salary : " +
    + jdbcresultset.getInt("salary"));
}
```

**Step 6: Close the connections:** After fetching the records, we will close the connection. The Connection interface close() method used to close the connection and the Statement and ResultSet objects will be closed automatically.

Syntax :

```
jdbcconn.close();
```

Let us now take an example, in which we create a JDBC application which establishes a database connection to consolidate our understanding of the concept.

**Example: 3.5.1:**

```
import Java.sql.*;
import Java.sql.DriverManager;
public class connection
{
    public static void main(String[] args)
    {
        Connection jdbcconn = null;
        try
        {
            //STEP 1: The JDBC driver is registered
            Class.forName("com.mysql.cj.jdbc.Driver");
```

```
// Alternatively, DriverManager.registerDriver(new
com.mysql.cj.jdbc.Driver());

//STEP 2: Open connection
System.out.println("Connecting to database...");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
"root", "admin");

//STEP 3: Create the Statement object
Statement jdbcstmt = jdbcconn.createStatement();

//STEP 4: Execute query to show table
System.out.println("Executing statement...");
ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from
Emp");

//STEP 5: The ResultSet is processed
while(jdbcresultset.next())
{
    //Retrieve data
    System.out.println("id : " + jdbcresultset.getInt("id") + " First
    : " + jdbcresultset.getString("first") + " Last : " +
    jdbcresultset.getString("last") + " Age : " +
    jdbcresultset.getInt("age") + " City : " +
    jdbcresultset.getString("city") + " Salary : " +
    jdbcresultset.getInt("salary"));
}
//STEP 6: Clean-up environment
jdbcconn.close();
}
catch(SQLException sqlex)
{
    //This will take care of JDBC errors
    sqlex.printStackTrace();
}
catch (Exception ex)
{
    // This will take care of Class.forName errors
    ex.printStackTrace();
}
finally
{
    // This will take care of closing all the resources
    Try
    {
        if (jdbcconn !=null)
            jdbcconn.close();
    }
    catch (SQLException sqlex)
    {
        sqlex.printStackTrace();
    }
    // end finally
}
// end try
System.out.println("Executed!");
}
//end main
}
// end Example
```

**Output:**

```

Connecting to database...
Executing statement...
id : 501 First : Aman Last : Sharma Age : 40 City : Delhi Salary : 8000
id : 502 First : Ajay Last : Tandon Age : 50 City : Mumbai Salary : 9000
id : 505 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000
id : 521 First : Aman Last : Sharma Age : 40 City : Delhi Salary : 8000
id : 522 First : Ajay Last : Tandon Age : 50 City : Mumbai Salary : 9000
id : 525 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000

```

Executed!

In this example: first, the import statements are used to import required classes in the code.

In step 1, We registered the JDBC driver (MySQL's Connector/J driver /Type 4 driver), and the Java virtual machine has loaded the desired driver implementation.

In step 2, we have used the getconnection() method. It opens a connection using database URL, username and password.

In step 3, we have created the statement object.

In step 4, We executed the query, where we have accessed an existing database through the connection object, and then data is inserted and accessed through the JDBC driver.

In step 5, we have extracted the data through the result set object and

In step 6, we have closed all the database connections to end the sessions.

## 3.5 JDBC SQLEXCEPTION

Exception handling is advantageous as it separates the Error-Handling Code from the main logic of the program. In the programming constructs, an exception occurs whenever we come across an abnormal condition, which interrupts the normal flow of the main logic code. The exceptions are handled in a controlled manner, and this stops the normal flow of execution and the control redirects to the possible catch clause. In JDBC, the exception instance generally thrown is SQLException. An SQLException means when a connection object is unable to find an appropriate driver to connect with the database, an exception is thrown. An SQLException thrown when we encounter an error during database interaction may be an inappropriate Driver or URL.

Following are the commonly used methods for JDBC SQLException:

**getmessage( ):** This method used to get the error message. For database error, it will provide Oracle error number and message, and for the driver error, it will provide JDBC driver error message.

**getErrorCode( ):** This method is used to get the exception error code, and generally, it is a vendor-specific exception.

**setNextException(SQLException sqlex):** This method is used to add another SQL exception in the chain.

**getNextException( ):** This method gets the next exception object from the exception chain.

**printStackTrace( ):** This method is used to print the current exception, throwable, and backtrace to a standard error stream.

**printStackTrace(PrintStream s):** This method is used to print the throwable and its backtrace to the specified print stream.

**printStackTrace(PrintWriter w):** This method is used to print the throwable and backtrace to the specified print writer.

The try-catch block utilized the information from the exception object, and it will catch the exception to continue the program execution.

```
try
{
    // Error prone code
}
catch(Exception ex)
{
    // exception handling code
}
finally
{
    // must be executed code for e.g. jdbconn.close();
}
```

In the try-catch block, we write the code which has potential chances of failure. If the code fails, the try block should throw an exception object, and the catch block will take care of that exceptions. The exception in the catch block handles the exception object and may print a statement or perform a pre-defined action to handles the issue. We may use multiple catch blocks, which allows taking more than one exception. The finally-block execute all other necessary components of the program despite the exception. JDBC Exceptions are an efficient way of handling exceptions and should always be used.

Now to consolidate our understanding of Exception handling, let us take the previous example 3.5.1 and pass the incorrect credentials.

In the first case, we pass the wrong driver, and the result will be class not found exception at exceptionhandling.main(exceptionhandling.Java:8)

#### Output:

```
Java.lang.ClassNotFoundException: com1.mysql.cj.jdbc.Driver
at
Java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader
.java:602)
at
Java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoad
ers.java:178)
at Java.base/Java.lang.ClassLoader.loadClass(ClassLoader.java:522)
at Java.base/Java.lang.Class.forName0(Native Method)
at Java.base/Java.lang.Class.forName(Class.java:340)
at exceptionhandling.main(exceptionhandling.java:8)

Executed!
```

## 3.6 JDBC DRIVERMANAGER CLASS

In the second case, let we pass the incorrect database name, and the program will show the unknown database error at exceptionhandling.main(exceptionhandling.Java:13). The outcome of the program as follows:

### Output:

```
Connecting to database...
Java.sql.SQLSyntaxErrorException: Unknown database 'emp1'
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:120)
    at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:97)
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122)
    at com.mysql.cj.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:836)
    at com.mysql.cj.jdbc.ConnectionImpl.<init>(ConnectionImpl.java:456)
    at com.mysql.cj.jdbc.ConnectionImpl.getInstance(ConnectionImpl.java:246)
    at com.mysql.cj.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:197)
    at java.sql.DriverManager.getConnection(DriverManager.java:677)
    at java.sql.DriverManager.getConnection(DriverManager.java:228)
    at exceptionhandling.main(exceptionhandling.java:13)

Executed!
```

The JDBC DriverManager is a concrete Java class that provides an interface between the user and drivers. The other JDBC components are Java interfaces implemented by the various driver packages. The JDBC driver can create the database connections, but generally, the DriverManager is used to get the connection. The JDBC DriverManager provides elementary services to maintain JDBC drivers and defines an object that connects the application to the Driver.

The DriverManager manages the various drivers running on an application. It tracks the available drivers and appropriately chooses the one which connects the application with the database.

It also tracks the login time limits of the driver and recording logs and tracing messages. The DriverManager.getConnection() method is the most commonly used method of this class to establish the database connection.

It maintains driver classes registered through the DriverManager.registerDriver() method and a list of all the available drivers preserved with the DriverManager class. The Class.forName() method is used to load the Driver. The DriverManager is being informed by the JDBC drivers whenever their implementation class is loaded.

Syntax:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

There are many associated methods with this class. Let us summarize a few of the important methods:

**registerDriver(Driver driver):** This method is used to register the Driver with the DriverManager class.

**deregisterDriver(Driver driver):** This method is used to deregister or drop from the list of registered drivers in the DriverManager class.

**getConnection(String URL):** This method is used to establish a connection with the given database URL.

**getConnection(String URL, String username, String password):** This method is used to establish the connection with the given URL, username and password.

**getConnection(String URL, Properties info):** This method is used to establish a connection with the given URL.

**getDrivers(String URL):** This method attempts to locate the Driver by the given string.

**getDrivers():** This method retrieves the information of the registered drivers of the DriverManager class.

## ☛ Check Your Progress 2

1. What are two approaches for registering the Driver?

---

---

---

2. Write down the importance of the JDBC DriverManager class?

---

---

---

3. getConnection()method belongs to which class?

---

---

---

4. Why do we get the following exception Java.sql.SQLException: No suitable driver found?

---

---

---

5. Can we use multiple catch and finally blocks with a single try statement?
- 
- 
- 
- 

## 3.7 JDBC CONNECTION INTERFACE

A Connection is an interface used for creating the session between the application and the database. A Java application may require to connect with a single database or sometimes, depending on the application, may require many connections with the different databases. The Connection interface of the Java.sql package characterizes a session with the connected database.

After a connection establishment, the execution of the SQL statements takes place. This process of query execution and result fetching is within the context of the connection. The implementation of these SQL queries requires the usage of some pre-defined libraries stored as Statement object, PreparedStatement object and CallableStatement object. Based on the application requirements, we can use any one of them.

The Connection interface contains Statement interface, PreparedStatement interface and CallableStatement interface. The getConnection() method of the class DriverManager is used to get a Connection object which is subsequently used to get the Statement object of Statement, PreparedStatement and CallableStatement. The connection interface also supports various methods for transaction management, another critical concept while using databases.

The DriverManager.getConnection() method is used to establish a connection and the DriverManager class searches and locate the most appropriate driver from the list of all registered Driver classes that can connect with the database specified in the URL.

syntax:

```
Connection jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root", "admin");
```

The commonly used methods for the Connection Interface are as follows:

**createStatement():** This method creates a statement object for executing the SQL queries.

**createStatement(int resultSetType, int resultSetConcurrency):** This method creates a statement object for the ResultSet

**PreparedStatement(String SQL):** This method creates a PreparedStatement object for executing parameterized queries.

**getMetaData():** This method returns a Database MetaData object containing the connected database information.

**setAutoCommit(boolean status):** This method is used to set the commit status, which is by default always true.

**commit():** This method saves the changes which have been committed or are permanent rollback.

**rollback():** This method drops all changes when commit or permanent rollback

**close():** This method closes the connection and immediately releases the JDBC resource.

### 3.7.1 Statement Interface

The Statement interface is used for the general-purpose access of the database. It is generally used when working with static SQL queries at runtime. We connect the JDBC to interact with the database, the Java.sql package, the two interfaces Java.sql.Statement and Java.sql.PreparedStatement. are used for efficiently executing the SQL queries.

For query execution, we may use any interface, but they have a different implementation. The Statement interface is used to execute the static SQL statement, and the PreparedStatement interface is used to execute pre-compiled queries.

The Statement interface provides various methods for query execution. It executes SQL queries using the basic methods like Statement.executeQuery(String) or Statement.executeUpdate(String) for DDL operations. This interface provides the Object of ResultSet. The Connection.createStatement() method is used to get a Statement object.

syntax:

```
Statement jdbstmt = jdbconn.createStatement();
```

The PreparedStatement and CallableStatement are the two sub-interfaces offered by the Statement interface.

The PreparedStatement object stores the pre-compiled query, which allows multiple runs of the same SQL query with different parameters efficiently. On the other hand, the CallableStatements are used to execute the SQL stored procedure. We will discuss these sub-interfaces in the subsequent section.

Following are the commonly used methods for Statement interface.

**ResultSet executeQuery(String SQL):** The executeQuery method is used to execute the SELECT statement, and will return the ResultSet object.

**int executeUpdate(String SQL):** The executeUpdate method is used to execute a specific query like insert, update, delete etc. and will return the number of affected rows.

**Boolean execute(String SQL):** The execute method used to execute queries that returns a boolean value and will return true if a ResultSet object fetched the result else, it would return false.

**executeBatch():** The executeBatch method executes batch statements.

Let us write a simple example of Statement interface to perform various basic operations like insert, update and delete.

**Example 3.7.1:**

```

import Java.sql.*;
import Java.sql.DriverManager;
class statementInterface
{
    public static void main(String args[])
    {
        Connection jdbcconn = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            jdbcconn =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
            "root", "admin");
            Statement jdbcstmt = jdbcconn.createStatement();

            // Update the record using execute method
            boolean status = jdbcstmt.execute("Update Emp set age = 40 where
            id =502");
            if(status == false)
            {
                System.out.println("Number of Updated rows " +
                jdbcstmt.getUpdateCount() );
            }
            // Insert the record using execute method
            int count1 = jdbcstmt.executeUpdate("Insert into Emp(id, First,
            Last, Age, City, Salary) values(512,'Gaurav', 'Mehta', 44,
            'Delhi', 5000)");
            System.out.println("Number of Rows Inserted " + count1);

            // Update the record using executeUpdate method
            int count2 = jdbcstmt.executeUpdate("Update Emp set age = 35 where
            id = 521");
            System.out.println("Number of Updated rows" + count2);

            // Delete the record using executeUpdate method
            //count = jdbcstmt.executeUpdate("Delete from Emp where id = 41");
            //System.out.println("Number of Rows Deleted " + count);

            // Executing Query using executeQuery method
            ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from
            Emp");

            // Retrieve data using ResultSet
            while(jdbcresultset.next())
            {
                System.out.println("id : " + jdbcresultset.getInt("id") + " First
                : " + jdbcresultset.getString("first") + " Last : " +
                jdbcresultset.getString("last") + " Age : " +
                jdbcresultset.getInt("age") + " City : " +
                jdbcresultset.getString("city") + " Salary : " +
                jdbcresultset.getInt("salary"));
            }
            jdbcconn.close();
        }
        catch(Exception e)
        {
            //Handle errors for Class.forName
            System.out.println(e);
        }
    }
}

```

```
    System.out.println("Executed!");
}
//end main
}
// end Example
```

**Output:**

```
Number of Updated rows 1
Number of Rows Inserted 1
Number of Updated rows1
id : 501 First : Aman Last : Sharma Age : 40 City : Delhi Salary : 8000
id : 502 First : Ajay Last : Tandon Age : 40 City : Mumbai Salary : 9000
id : 505 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000
id : 512 First : Gaurav Last : Mehta Age : 44 City : Delhi Salary : 5000
id : 521 First : Aman Last : Sharma Age : 35 City : Delhi Salary : 8000
id : 522 First : Ajay Last : Tandon Age : 50 City : Mumbai Salary : 9000
id : 525 First : Suraj Last : Gupta Age : 45 City : Pune Salary : 8000
```

```
Executed!
```

### 3.7.2 PreparedStatement Interface

As we have discussed that for query execution, we can use any of the interfaces Java.sql.Statement and Java.sql.PreparedStatement. The Java.sql.Statement is generally used for simple DDL queries but for better efficiency where repeating SQL queries or parameterization is required, we use the Java.sql.PreparedStatement. PreparedStatement interface inherits the primary statement interface mentioned earlier and offers parameterization, and also it is much safer against SQL injection attacks.

PreparedStatement interface is generally used when we need to execute the SQL statements multiple times, and the interface accepts input parameters at the runtime. The interface corresponds to pre-compiled statements, offers the better performance of the application and will fetch the result faster as the query is compiled only once. These statements are first compiled into the database and subsequently stored as a PreparedStatement object. This object facilitates the SQL statement execution several times.

The PreparedStatement interface, when used for the parameterized queries a question mark (?) symbol as a placeholder known as parameter-marker, is passed for the values. The question mark values are set by the PreparedStatement and values supplied for every parameter before query execution. The advantage of using PreparedStatement is that we can use the same query multiple times with different parameter values.

For example, String SQL="insert into Emp values( ?, ?, ?, ?, ?, ?)"

Here, we pass these (?) parameter used as a placeholder in the query and the values set at the runtime by calling the PreparedStatement setter methods. The various setter methods are used to provide values for these placeholders based on the data-types like setInt(), setFloat(), setString() etc. The setInt(int parameterIndex, value) is the standard form of the setter method where "Int" part in setInt is varying based on the datatype float, string etc. and the parameter index position is defined as parameterIndex in the statement. The parameterIndex starts from 1, unlike Java array indices, which starts at 0.

We can use the PreparedStatement method of the Connection Interface to get the PreparedStatement object.

Syntax

```
PreparedStatement jdbcprepstmt = connection.prepareStatement(SQL);
```

The PreparedStatement is more efficient as it directly sent the compiled SQL to the database, thus is more efficient for repeated executions.

Following are the commonly used methods for PreparedStatement interface:

**setInt(int parameterIndex, int value):** The setInt method used for setting Int value as per the parameter index.

**setString(int parameterIndex, String value):** The setString method is used to set the String value as per the parameter index.

**setFloat(int parameterIndex, float value):** The setFloat method is used to set the Float value as per the parameter index.

**setDouble(int parameterIndex, double value):** The setDouble method is used to set the Double value as per the parameter index.

**executeUpdate():** The executeUpdate method used for executing a query like create, drop, insert, update, delete etc.

**ResultSet executeQuery():** This method is used for executing the select query, and it will return an instance of ResultSet.

Let us write a simple example of the parameterized interface to perform the basic operations of insert, update and delete

### Example 3.7.2:

```
import Java.sql.*;
import Java.sql.DriverManager;
public class preparedStmt
{
    public static void main(String[] args)
    {
        Connection jdbcconn = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println("Connecting to database...");
            jdbcconn =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
            "root", "admin");
            preparedStmt jdbcprepstmt = new preparedStmt();
            jdbcprepstmt.insertEmp(jdbcconn, 621, "Kushagr", "Mahajan", 31,
            "Pune", 6000);
            jdbcprepstmt.updateEmp(jdbcconn, 521, 33);
            jdbcprepstmt.displayEmp(jdbcconn, 502);
            jdbcprepstmt.deleteEmp(jdbcconn, 522);

            jdbcconn.close();
        }
        catch(Exception e)
        {
            //Handle errors for Class.forName
            System.out.println(e);
        }
    //end try
    System.out.println("Executed!");
}
```

```
}

/*end main
*/
/*
 * PreparedStatement to Insert the record
 * @parameter jdbccconn
 */
private void insertEmp(Connection jdbccconn, int id, String first,
String last, int age, String city, int salary)
throws SQLException
{
    String insertSQL = "insert into Emp values( ?, ?, ?, ?, ?, ?, ?)";
    PreparedStatement jdbcprepstmt = null;
    try
    {
        jdbcprepstmt = jdbccconn.prepareStatement(insertSQL);
        jdbcprepstmt.setInt(1, id); // here 1 is to specify that it is
        the first parameter
        jdbcprepstmt.setString(2, first);
        jdbcprepstmt.setString(3, last);
        jdbcprepstmt.setInt(4, age);
        jdbcprepstmt.setString(5, city);
        jdbcprepstmt.setInt(6, salary);

        int rowcount = jdbcprepstmt.executeUpdate();
        System.out.println("Count of rows inserted " + rowcount);
    }

    finally
    {
        if (jdbcprepstmt != null)
        {
            jdbcprepstmt.close();
        }
    }
}

/*
 * PreparedStatement to Update the record
 * @parameter jdbccconn, id, age
 */
private void updateEmp(Connection jdbccconn, int id, int age) throws
SQLException
{
    String updateSQL = "Update emp set age = ? where id = ?";
    PreparedStatement jdbcprepstmt = null;
    try
    {
        jdbcprepstmt = jdbccconn.prepareStatement(updateSQL);
        jdbcprepstmt.setInt(1, age);
        jdbcprepstmt.setInt(2, id);
        int rowcount = jdbcprepstmt.executeUpdate();
        System.out.println("Count of rows updated " + rowcount);
    }

    finally
    {
        if (jdbcprepstmt != null)
        {
            jdbcprepstmt.close();
        }
    }
}
```

```

* PreparedStatement to Delete the record
* @parameter jdbcconn, id
*/
private void deleteEmp(Connection jdbcconn, int id) throws
SQLException
{
    String deleteSQL = "Delete from emp where id = ?";
    PreparedStatement jdbcprepstmt = null;
    try
    {
        jdbcprepstmt = jdbcconn.prepareStatement(deleteSQL);
        jdbcprepstmt.setInt(1, id);
        int rowcount = jdbcprepstmt.executeUpdate();
        System.out.println("Count of rows deleted " + rowcount);
    }
    finally
    {
        if (jdbcprepstmt != null)
        {
            jdbcprepstmt.close();
        }
    }
}
/*
* PreparedStatement to Display the record
* @parameter jdbcconn, id
*/
private void displayEmp(Connection jdbcconn, int id) throws
SQLException
{
    String selectSQL = "Select * from emp where id = ?";
    PreparedStatement jdbcprepstmt = null;
    try
    {
        jdbcprepstmt = jdbcconn.prepareStatement(selectSQL);
        jdbcprepstmt .setInt(1, id);
        ResultSet jdbcresultset = jdbcprepstmt.executeQuery();
        while (jdbcresultset.next())
        {
            System.out.println("id : " + jdbcresultset.getInt("id") + " "
                First : " + jdbcresultset.getString("first") + " Last : " +
                jdbcresultset.getString("last") + " Age : " +
                jdbcresultset.getInt("age") + " City : " +
                jdbcresultset.getString("city") + " Salary : " +
                jdbcresultset.getInt("salary"));
        }
    }
    finally
    {
        if (jdbcprepstmt != null)
        {
            jdbcprepstmt.close();
        }
    }
}
}

```

**Output:**

```
Connecting to database...
Count of rows inserted 1
Count of rows updated 1
id : 502 First : Ajay Last : Tandon Age : 40 City : Mumbai Salary : 9000
Count of rows deleted 1
Executed!
```

The query execution in the relational database goes through the steps of parsing, compilation, optimization and finally, implementation of an optimized query. The Statement interface is required to go through all the steps, but with the PreparedStatement the first three steps are executed while creating the preparedStatement itself. This makes this execution quicker than Statement.

### 3.7.3 CallableStatement Interface

We have already seen that we can execute the simple SQL query using Statement interface and pre-compiled parameterized SQL query using PreparedStatement. The JDBC API also provides a CallableStatement interface used to implement store procedures and an extension of the PreparedStatement.

The Stored procedure is a set of SQL statements that is stored as a group and resided within the database and can be shared by multiple programs. These stored procedures make the performance better because these are pre-compiled. The pre-compiled procedure is executed with just one call to the database server being in the same database server space, so reduce the network traffic. Further, we can apply any business logic to the database through the concept of stored procedures and functions. For example, we can evaluate the age of the employee using the date of birth. To get the result, we can create a function that can take input as the date of birth and get the output as the age of the employee.

CallableStatement interface is used to call the stored procedures and functions. We can use the prepareCall() method of the Connection interface to get the CallableStatement object.

#### Syntax

```
CallableStatement jdbcstmt = connection.prepareCall("{call
PROCEDURE_NAME( ?, ?, ?)}");
```

The symbol “?” is a placeholder they can use to register IN, OUT, and INOUT parameters.

There are three types of parameters used in the stored procedure. The PreparedStatement object generally uses only the IN parameter, but the CallableStatement object can use all three parameters IN, OUT, and INOUT.

The In parameter is used when the SQL statement is created, and the value is unknown, i.e. the IN parameter is used to pass the values to the stored procedure. The OUT parameter value returned after the execution of the query, i.e. OUT parameter used to hold the result returned by the stored procedure. The INOUT parameter provides both input and output values, i.e. IN OUT acts as both IN and OUT parameter. We use the setter method to bind values to IN parameters, and we use the getter method to retrieve values from the OUT parameters.

In our program, we use CallableStatement method registerOutParamter() to register OUT parameter before calling the stored procedure.

Following are the commonly used methods for CallableStatement interface:

**execute():** This method is used to execute a query and return a Boolean value. The value is true if a result is a ResultSet object, and it will be false if there are no results or it's an updated count.

**executeUpdate():** This method is for DML statements or DDL statements like Insert, Update Create etc.

**executeQuery():** This method is for SQL statement that returns ResultSet.

Let us write a simple example of a callable interface. In this example, we are using a stored procedure “employee\_details” that receives “id” as the parameter and provide all other details of the employee.

### Example 3.7.3:

```
delimiter $$$
DROP PROCEDURE IF EXISTS emp.employee_details $$$
CREATE PROCEDURE emp.employee_details
(IN p_id int,
OUT p_first varchar(30), OUT p_last varchar(30), OUT p_age int, OUT p_city
varchar(30), OUT p_salary int
)
BEGIN
SELECT id, first, last, age, city, salary
INTO
p_id, p_first, p_last, p_age, p_city, p_salary
from EMP
where id = p_id;
END
$$$

import java.sql.*;
import java.sql.DriverManager;
public class callableStmt
{
    public static void main(String[] args)
    {
        Connection jdbcconn = null;
        CallableStatement jdbccallstmt = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println("Connecting to database...");
            jdbcconn =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
            "root", "admin");
            System.out.println("Creating statement...");
            String SQL = "{call employee_details ( ?,?,?,?,?,?)}";
            jdbccallstmt = jdbcconn.prepareCall(SQL);
            //First Bind the IN parameter and after that bind the OUT parameter
            int p_id = 521;
            jdbccallstmt.setInt(1, p_id); // This would set ID as 21
            // We need to register the OUT parameters
            jdbccallstmt.registerOutParameter(2, Java.sql.Types.VARCHAR);
            jdbccallstmt.registerOutParameter(3, Java.sql.Types.VARCHAR);
            jdbccallstmt.registerOutParameter(4, Java.sql.Types.BIGINT);
            jdbccallstmt.registerOutParameter(5, Java.sql.Types.VARCHAR);
```

```

jdbccallstmt.registerOutParameter(6, Java.sql.Types.BIGINT);
// Stored procedure is run using the execute method
jdbccallstmt .execute();
//Fetch employee details with getter method
String p_first = jdbccallstmt.getString(2);
String p_last = jdbccallstmt.getString(3);
Integer p_age = jdbccallstmt.getInt(4);
String p_city = jdbccallstmt.getString(5);
Integer p_salary = jdbccallstmt.getInt(6);
System.out.println("Employee Details with ID:" + p_id + " are " +
p_first + " " + p_last + " " + p_age + " " + p_city + " " +
p_salary);
jdbccconn.close();
}
catch (Exception e)
{
System.out.println(e);
}
}
}
}

```

**Output:**

```

Connecting to database...
Creating statement...
Employee Details with ID:521 are Aman Sharma 33 Delhi 8000

```

The JDBC API offers three different interfaces viz. The statement interface executes routine SQL queries, PreparedStatement to execute dynamic or parameterized SQL queries and CallableStatement to implement the stored procedures. The three interfaces significantly differ in terms of functionality and performance. Let us summarize the differences between these three interfaces.

Table: the differences between Statement, PreparedStatement, and CallableStatement interfaces

Statement	PreparedStatement	CallableStatement
execute Normal SQL queries	execute parameterized or dynamic queries	call the stored procedure
Preferred to use when query execution only once like DDL statements	preferred to use when a query executed multiple times	preferred to use when the stored procedure executed
we cannot pass the parameters	accepts the parameters at run time	accepts at run time and uses three different types of parameters IN, OUT, IN OUT
offers low performance	offers better performance when multiple queries are executed	offers very high performance

### ☛ Check Your Progress 3

1. Why is Statement object required to perform SQL query execution?

2. What is the fundamental difference between the Statement and PreparedStatement interface?

---

---

---

---

3. What is the return type of execute(), executeQuery() and executeUpdate() and Which method executes any kind of SQL statement?

---

---

---

---

4. What executeUpdate() method of PreparedStatement interface does?

---

---

---

---

---

### 3.8 SUMMARY

In this unit, we have seen how the JDBC API is used for Database Connectivity for Java applications. JDBC API provides various methods and interfaces for accessible communication with the database, which facilitates an application to execute queries, retrieve results and update data. We have discussed the JDBC drivers, which ensure the database independence and only changing the database driver, and a few lines of code allows us to change the backend database. The JDBC Database Connection Steps to understand the process starting from loading a driver, which opens a communication channel with the database to create a connection through DriverManager class—subsequently creating and executing the statements to fetch the result and finally closing the connection.

The various types of JDBC drivers are available, and there are several factors that may be critical in choosing an appropriate driver for the application. The DriverManager class provides vital services to maintain JDBC drivers and defines an object that connects Java applications to a JDBC driver. Registering and deregistering drivers, locating and storing the list of available drivers, and establishing connections are critical tasks handled by the DriverManager class.

The connection interface is used to represent a session with the connected database and offer various objects for efficient database query execution. The Statement interface generally executes a simple query, the PreparedStatement interface is used to manage repeating queries, and the CallableStatement interface is further improvising on implementing stored procedures.

---

## **3.9      SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS**

---

### **➤ Check Your Progress 1**

1. Following are the necessary steps to create a JDBC application:
  - a. The first step is to import all the required packages which contain the JDBC classes.
  - b. The second step is to register the appropriate JDBC driver, which can perform the task of opening the communications channel with the database.
  - c. The third step is to use the Driver Manager class getConnection() method, which can open a connection.
  - d. The fourth step is to use the object of Statement Interface to execute a query.
  - e. The fifth step is to extract the data using the ResultSet method.
  - f. The sixth step is to close all the connections and thus clean up the environment.
2. The Thin Driver is the fastest pure Java driver which communicates directly through socket connections. It doesn't require any middleware or native library. It thus connects with the database by converting the JDBC method calls directly into the vendor database protocol without requiring any intermediate library.
3. It provides two packages Java.sql.\* and Javax.sql.\*

### **➤ Check Your Progress 2**

1. The two approaches for registering the Driver are as follows:

- a. The first approach uses the class.forName() method for dynamically loading the Driver's class into the memory, and JVM registers the Driver automatically.
  - b. The second approach uses the DriverManager.registerDriver() method, which statically loads the Driver and is generally used in non-JDK compliant JVM.

2. The JDBC DriverManager is a concrete Java class that provides an interface between the user and drivers. The other JDBC components are Java interfaces and

are implemented by the various driver packages. The JDBC driver can create the database connections, but generally, the DriverManager is used to get the connection. The JDBC DriverManager provides elementary services to maintain JDBC drivers and defines an object that connects Java applications to a JDBC driver.

3. The `getConnection()` method is used to establish a connection with the database and is belongs to `DriverManager` class.
4. The “exception `Java.sql.SQLException: No suitable driver found`” exception is due to the unformatted SQL URL string.
5. Yes, we can use multiple catch blocks with a single try statement, and also, the try-catch blocks can be nested similar to if-else statements, but there can only be one finally block with a try-catch statement.

### Check Your Progress 3

1. The execution of the SQL queries requires some pre-defined library defined in the form of `Statement` object, `PreparedStatement` object and `CallableStatement` object. Based on the application requirement, we need to create either `Statement`, `PreparedStatement` or `CallableStatement` object.
2. The fundamental difference between the `Statement` interface and the `PreparedStatement` interface is that the `PreparedStatement` interface uses pre-compiled statements for execution and thus offers better performance and safeguards from SQL injection attacks.
3. The `execute()` return type is `Boolean`, `executeQuery()` return type is `ResultSet` object and `executeUpdate()` return type is `int`. The boolean `execute()` can execute any kind of SQL statement.
4. This method is used for executing a query like create, drop, insert, update, delete etc.

---

### 3.10 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
- ... Sharan, Kishori, “Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs”, Apress, 2014.
- ... Parsian, Mahmoud, “*DBC metadata, MySQL, and Oracle recipes: a problem-solution approach* ” Apress, 2006.
- ... <https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>
- ... <https://www.roseindia.net/jdbc/>

---

# **UNIT4      JDBC PART 2**

---

## **Structure**

- 4.0 Introduction
  - 4.1 Objectives
  - 4.2 JDBC ResultSet Interface
    - 4.2.1 ResultSet Types
    - 4.2.2 ResultSet Concurrency
    - 4.2.3 ResultSet Holdability
    - 4.2.4 ResultSet MetaData
  - 4.3 JDBC Transactions
    - 4.3.1 Rollback() and commit()
    - 4.3.2 JDBC Transaction Isolations
    - 4.3.3 JDBC Savepoints
  - 4.4 JDBC Batch Processing
  - 4.5 JDBC RowSet Interface
  - 4.6 Introduction to Java Data Object (JDO)
  - 4.7 Summary
  - 4.8 Solutions/ Answer to Check Your Progress
  - 4.9 References/Further Reading
- 

## **4.0 INTRODUCTION**

---

In this unit, we will discuss the concept of the ResultSet interface, which contains tabular data of a database query. The ResultSet objects maintain a cursor and associated navigational methods to manage the data. There are three different ResultSet types that control the cursor movement and allow access to the row data through relative and absolute positions. The ResultSet concurrency provides read-only and updatable-mode, which contains access for the modification of the data. The ResultSet holdability ensure the database update within the same context by either closing the instance on commit or holding up the connection to accommodate changes in the same ResultSet instance.

The JDBC transaction will also be discussed in this unit. The transaction concept is a fundamental concept from the batch processing point of view, where the related queries are executed in a batch, and it is essential to execute either all the queries of the set or none. If even one of the queries fails, it may lead the database to the inconsistency state. To consolidate our understanding of the topic, we will also discuss the transaction isolation and save points within the context of the Transaction. Transaction isolation is a critical concept in transaction management to control and satisfy the ACID properties. Also, the idea of save points or checkpoints has been discussed to ensure that in case of any failure, we may not require complete rollback but merely using the checkpointing system, we can keep our time and efforts.

The concept of JDBC RowSet discussed where the RowSet interface provides a wrapper around a ResultSet object and thus provide all the ResultSet capabilities. In addition to that, it also offers support to the JavaBeans component model. The RowSet interface is worthy for databases where there is no driver support for specific ResultSet properties. At the end of this unit, the idea of Java Data Objects and the advantages over the JDBC API has been explained in brief.

## 4.1 OBJECTIVES

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

- ... explain the concept of ResultSet interface,
- ... understand the basic concepts of Transaction,
- ... use of commit() and rollback() methods appropriately for any transaction,
- ... describe the concept of transaction isolation levels for concurrent execution,
- ... explain the concept of SavePoints,
- ... write the program using batch processing in JDBC,
- ... write the program using the RowSet interface to fetch the data in the tabular form, and
- ... understand the basics of Java Data Objects.

## 4.2 JDBC RESULTSET INTERFACE

Resultset Interface is used to store and fetched the executed outcome of any SQL statement. The query execution requires the data to be read from the database and subsequently keep the result after processing. The ResultSet acts like the storage; it stores the data or result of the query after the execution of the SQL statements. The java application may use this result after fetching it from the ResultSet.

We can create a ResultSet as

```
Statement jdbcstmt = jdbcconn.createStatement();
```

```
ResultSet jdbcresultset = jdbcstmt.executeQuery ("select * from Emp");
```

The ResultSet contains the tabular data where each record includes equal columns. The data is accessed through the cursor maintained by the ResultSet object, and initially, the cursor is positioned before the first row, and after calling the next() for the first time, it points at the first record. The cursor position is maintained at the current row by the Resultset object, and to access the subsequent rows, either we use next() method or previous() method.

The data in the ResultSet is accessed using the cursor, and the cursor moves to the subsequent rows in the forward direction is taken care of by next() method, and it moves to the previous rows through previous() method of the ResultSet object. These methods return true when the next record is available and return false when no more rows are in the ResultSet object that implies the cursor is pointing after the last record. Also, the next() and previous() methods need not be manually closed as they get automatically closed after the Statement object is committed.

We have seen in the previous unit that to access the data using ResultSet we can iterate through the ResultSet.

```
// Retrieve data using ResultSet
while(jdbcresultset.next())
{
    System.out.println("id : " + jdbcresultset.getInt("id") + " First : " +
    jdbcresultset.getString("first") + " Last : " + jdbcresultset.getString("last") + " Age : " +
    jdbcresultset.getInt("age") + " City : " + jdbcresultset.getString("city") " Salary : " +
    jdbcresultset.getInt("salary"));
}
```

We can access the data of the record column-wise. For that, we require the getter methods depending on the column's data type we are accessing like; for integer datatype, we can call getInt() method or for string data type, we can use getString() method. We can access the data calling the column name or column index.

The ResultSet has three attributes - Type, Concurrency and Holdability, which we generally set while creating Statement or Prepared Statement.

#### 4.2.1 ResultSet Types

The ResultSet offers different Types based on the cursor movement which provide few important characteristics but not all types may be available with all the databases and drivers which means not all methods works with all ResultSet Types. Few of the Navigation methods are first(), last(), next(), previous(), relative(), absolute(), afterLast(), beforeLast() etc. We can check their availability using DatabaseMetaData.supportsResultSetType(int type) method.

The three ResultSet types are TYPE\_FORWARD\_ONLY, TYPE\_SCROLL\_SENSITIVE and TYPE\_SCROLL\_INSENSITIVE. The TYPE\_FORWARD\_ONLY is the default ResultSet type, it confirms the movement, or it allows the user to iterate only in the forward direction like first row, second row 2, third row and so on.

TYPE\_SCROLL\_SENSITIVE and TYPE\_SCROLL\_INSENSITIVE allow the user to iterate in both directions and also permits to access relative positions concerning the current position, or even we can access an absolute position.

But, the TYPE\_SCROLL\_SENSITIVE is sensitive enough to reflect any change in the record. At the same time, it is already open, i.e. it provides a dynamic view of the underlying data, whereas TYPE\_SCROLL\_INSENSITIVE would not reflect any change in the database record with the opened Resultset. An insensitive ResultSet only provides a static view of the underlying data, i.e. columns values of rows.

Subsequently, we will require a new ResultSet object to view the database changes.

Let us summarize the three ResultSet types: 1) The ResultSet.TYPE\_FORWARD\_ONLY is neither scrollable and nor sensitive. 2) The ResultSet.TYPE\_SCROLL\_SENSITIVE is scrollable and sensitive to the database changes, and 3) The ResultSet.TYPE\_SCROLL\_INSENSITIVE is scrollable but not sensitive to the underlying changes to the database.

#### 4.2.2 ResultSet Concurrency

The ResultSet Concurrency evaluates the ResultSet status through the concurrency mode, which provides the two levels. The first mode, where the ResultSet is updatable, i.e. ResultSet.CONCUR\_UPDATABLE() and we can update the columns of each row, or we can insert new rows. The other level is the read-only, i.e. ResultSet.CONCUR\_READ\_ONLY() which is a default ResultSet concurrency. To incorporate the ResultSet Concurrency, we may require to use the database lock mechanisms.

Again, not all the JDBC drivers support the concurrency modes so that we can check the supported mode through DatabaseMetaData.supportsResultSetConcurrency(int concurrency) method.

The ResultSet Concurrency is independent of scrollability, but generally, we need both ResultSet.TYPE\_SCROLL\_SENSITIVE and ResultSet Concurrency to satisfy so that we can reach to a particular row to update. There are six ResultSet categories in total, which are as follows:

ResultSet.TYPE\_FORWARD\_ONLY ()/ ResultSet.CONCUR\_READ\_ONLY()

```
ResultSet.TYPE_FORWARD_ONLY() / ResultSet.CONCUR_UPDATABLE()  
  
ResultSet.TYPE_SCROLL_SENSITIVE / ResultSet.CONCUR_READ_ONLY()  
  
ResultSet.TYPE_SCROLL_SENSITIVE / ResultSet.CONCUR_UPDATABLE()  
  
ResultSet.TYPE_SCROLL_INSENSITIVE / ResultSet.CONCUR_READ_ONLY()  
  
ResultSet.TYPE_SCROLL_INSENSITIVE / ResultSet.CONCUR_UPDATABLE()
```

Let us take an example to understand the ResultSet interface where we fetch the last record, all records, absolute row record and relative row record. In this example, when we try to update the record of the last row, we get the error as due to ResultSet.CONCUR\_READ\_ONLY(), records are not updatable.

### Example 4.2.1

```
import java.sql.*;  
import java.sql.DriverManager;  
class scrollableRS  
{  
    public static void main(String args[])  
    {  
        Connection jdbcconn = null;  
        try  
        {  
            Class.forName("com.mysql.cj.jdbc.Driver");  
            jdbcconn =  
            DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",  
            "bpit");  
  
            // Scrollable ResultSet but insensitive to any change made by others  
            // Read-only Resultset, not updatable  
            Statement jdbstmt =  
            jdbcconn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
            ResultSet.CONCUR_READ_ONLY);  
            {  
                ResultSet jdbresultset = jdbstmt.executeQuery("select * from  
                emp10");  
  
                jdbresultset.last();  
                System.out.println("--- Show the Last Row ---");  
                System.out.println(jdbresultset.getRow() + ":" +  
                + jdbresultset.getInt("id") + ", "  
                + jdbresultset.getString("first") + ", " +  
                jdbresultset.getString("last") + ", "  
                + jdbresultset.getInt("age") + ", " + jdbresultset.getString("city")  
                + ", "  
                + jdbresultset.getInt("salary"));  
  
                jdbresultset.beforeFirst();  
                System.out.println("--- Show all the Rows ---");  
                while (jdbresultset.next())  
                {  
                    System.out.println(jdbresultset.getRow() + ":" +  
                    + jdbresultset.getInt("id") + ", "  
                    + jdbresultset.getString("first") + ", " +  
                    jdbresultset.getString("last") + ", "  
                    + jdbresultset.getInt("age") + ", " + jdbresultset.getString("city")  
                    + ", "
```

```

+ jdbcresultset.getInt("salary"));
}

jdbcresultset.absolute(5); // starting row number is 1
System.out.println("--- Show Absolute Row 5 ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+
jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

jdbcresultset.relative(-2);
System.out.println("--- Show Relative Row -2 ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

// Update a row
jdbcresultset.last();
System.out.println("--- Try to Update a row ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));
jdbcresultset.updateInt("salary", 9000); // update cells via column
name
jdbcresultset.updateInt("age", 50);
jdbcresultset.updateRow(); // update the row in the data source
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

}
jdbccconn.close();
}
catch (Exception e)
{
//Handle errors for Class.forName
System.out.println(e);
} // end try
System.out.println("Executed!");
}// end main

} // end Example

```

**Output:**

```
--- Show the Last Row ---
8: 621, Kushagr, Mahajan, 31, Pune, 6000
--- Show all the Rows ---
1: 501, Aman, Sharma, 40, Delhi, 8000
2: 502, Ajay, Tandon, 40, Mumbai, 9000
3: 505, Suraj, Gupta, 45, Pune, 8000
4: 509, Gaurav, Pant, 49, Delhi, 5000
5: 512, Gaurav, Mehta, 44, Delhi, 5000
6: 521, Aman, Sharma, 33, Delhi, 8000
7: 525, Raghav, Gupta, 32, Pune, 8000
8: 621, Kushagr, Mahajan, 31, Pune, 6000
--- Show Absolute Row 5 ---
5: 512, Gaurav, Mehta, 44, Delhi, 5000
--- Show Relative Row -2 ---
3: 505, Suraj, Gupta, 45, Pune, 8000
--- Try to Update a row ---
8: 621, Kushagr, Mahajan, 31, Pune, 6000
com.mysql.cj.jdbc.exceptions.NotUpdatable: Result Set not
updatable. This result set must come from a statement that was created
with a result set type of ResultSet.CONCUR_UPDATABLE, the query must
select only one table, can not use functions and must select all
primary keys from that table. See the JDBC 2.1 API Specification,
section 5.6 for more details.
Executed!
```

### 4.2.3 ResultSet Holdability

The ResultSet Holdability evaluates the ResultSet instance if it has a closed status when the underlying connection commit(). The commit() method used to ensure the consistent database states. In the case of ResultSet Holdability, there are two types of holdability. The first one is where all the ResultSet instances are closed on commit(), i.e. CLOSE\_CURSORS\_OVER\_COMMIT and the other one where all the ResultSet instances are not closed but kept open to accommodate the updates with the same ResultSet on commit(), i.e. HOLD\_CURSORS\_OVER\_COMMIT.

The ResultSet Interface methods are as follows:

boolean next(): The next() method is used for advancing the cursor to next row.

boolean previous(): The previous() method precedes the cursor to the previous row.

boolean first(): The first() method moves the cursor to the first row.

boolean last(): The last() method move the cursor to the last row.

boolean absolute(int row): The absolute() method moves the cursor to the specific row.

boolean relative(int row): The relative() method moves the cursor to the relative row number from the current row.

Again, not all the JDBC drivers support the concurrency modes so that we can check the supported mode through a DatabaseMetaData.supportsResultSetHoldability(int holdability) method. The Connection interface getHoldability() method is used to return an integer value 1 or 2. The ResultSet.HOLD\_CURSORS\_OVER\_COMMIT returns the value 1, and the ResultSet.CLOSE\_CURSORS\_AT\_COMMIT returns the value 2.

We can check the Default cursor holdability using

DatabaseMetaData.getResultSetHoldability() and also check the driver support for HOLD\_CURSORS\_OVER\_COMMIT and CLOSE\_CURSORS\_AT\_COMMIT using DatabaseMetaData.supportsResultSetHoldability(ResultSet.HOLD\_CURSORS\_OVER\_COMMIT) and DatabaseMetaData.supportsResultSetHoldability(ResultSet.CLOSE\_CURSORS\_AT\_COMMIT) respectively.

Let us take an example where the holdability value is set to HOLD\_CURSORS\_OVER\_COMMIT. In this example, first, we have to set the auto-commit to false and then retrieve the employee table's content to a ResultSet object and insert a new row in the ResutlSet and to the employee table. At the end, we will commit the execution, but due to the holdability property HOLD\_CURSORS\_OVER\_COMMIT, we can check that the ResultSet object is open.

#### Example 4.2.3:

```
import java.sql.*;
import java.sql.DriverManager;
class holdabilityRS
{
    public static void main(String args[])
    {
        Connection jdbcconn = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            jdbcconn =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
            "root", "admin");
            // By default auto commit is true so setting the auto commit false
            jdbcconn.setAutoCommit(false);
            // Holdability is set to HOLD_CURSORS_OVER_COMMIT
            jdbcconn.setHoldability(ResultSet.HOLD_CURSORS_OVER_COMMIT);
            // Creating a Statement object
            Statement jdbcstmt =
            jdbcconn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
            // Execute query to fetch the data
            ResultSet jdbcresultset = jdbcstmt.executeQuery("select * from
            emp10");
            System.out.println("Employee table");
            while (jdbcresultset.next())
            {
                System.out.print("ID: " + jdbcresultset.getInt("id") + ", ");
                System.out.print("First_Name: " + jdbcresultset.getString("first")
                + ", ");
                System.out.print("Last_Name: " + jdbcresultset.getString("last"));
                System.out.print("Age: " + jdbcresultset.getInt("age") + ", ");
                System.out.print("City: " + jdbcresultset.getString("city") + ",
                ");
                System.out.print("Salary: " + jdbcresultset.getInt("salary"));
                System.out.println("");
            }
            // Insert new row
            jdbcresultset.moveToInsertRow();
            jdbcresultset.updateInt(1, 622);
            jdbcresultset.updateString(2, "Poonam");
            jdbcresultset.updateString(3, "Sharma");
            jdbcresultset.updateInt(4, 34);
            jdbcresultset.updateString(5, "Delhi");
        }
    }
}
```

```
jdbcresultset.updateInt(6, 6000);
jdbcresultset.insertRow();
// commit transaction
jdbcconn.commit();
boolean status = jdbcresultset.isClosed();
if (status)
{
    System.out.println("ResultSet object close");
}
else
{
    System.out.println("ResultSet object open");
}
}
catch (Exception e)
{
    //Handle errors for Class.forName
    System.out.println(e);
}
// end try
System.out.println("Executed!");
}
// end main
}
// end Example
```

**Output:**

```
Employee table
ID: 501, First_Name: Aman, Last_Name: SharmaAge: 40, City: Delhi,
Salary: 8000
ID: 502, First_Name: Ajay, Last_Name: TandonAge: 40, City: Mumbai,
Salary: 9000
ID: 505, First_Name: Suraj, Last_Name: GuptaAge: 45, City: Pune,
Salary: 8000
ID: 509, First_Name: Gaurav, Last_Name: MehtaAge: 44, City: Delhi,
Salary: 5000
ID: 512, First_Name: Gaurav, Last_Name: MehtaAge: 44, City: Delhi,
Salary: 5000
ID: 521, First_Name: Aman, Last_Name: SharmaAge: 33, City: Delhi,
Salary: 8000
ID: 525, First_Name: Suraj, Last_Name: GuptaAge: 45, City: Pune,
Salary: 8000
ID: 621, First_Name: Kushagr, Last_Name: MahajanAge: 31, City: Pune,
Salary: 6000
ResultSet object open
Executed!
```

#### 4.2.4 ResultSet MetaData

ResultSetMetaData interface provides data about the data available in the ResultSet. It includes all the column and their characteristics. We can get the metadata object through the method from the ResultSet object.

syntax:

```
public ResultSetMetaData getMetaData()
```

The ResultSetMetaData interface supports the following methods:

ResultSetMetaData.getColumnCount(): This method used to fetch the number of columns in the ResultSet object.

ResultSetMetaData.getColumnName(int indexno): This method used to fetch the name of the columns from the ResultSet object.

ResultSetMetaData getColumnTypeName (int indexno): This method used to get the column datatype from the ResultSet object.

ResultSetMetaData getColumnDisplaySize (int indexno): This method used to get the column size from the ResultSet object.

Let us take an example to understand the concept of ResultSetMetaData:

#### Example 4.2.4:

```
import java.sql.*;
import java.sql.DriverManager;
public class resultMeta
{
    public static void main(String[] args)
    {
        Connection jdbccconn = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            jdbccconn =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
            "root", "admin");
            Statement jdbcstmt = jdbccconn.createStatement();

            ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from
            Emp");
            ResultSetMetaData jdbcrsmd = jdbcresultset.getMetaData();

            int countno = jdbcrsmd.getColumnCount();
            System.out.println("Total number of columns are " + countno);
            // Display column name, data type and the column size
            for (int i = 1; i <= countno; i++)
            {
                System.out.println(jdbcrsmd.getColumnName(i) + " " +
                jdbcrsmd.getColumnTypeName(i) + " "
                + jdbcrsmd.getColumnDisplaySize(i));

                System.out.println();
            }
            jdbccconn.close();
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
        System.out.println("Success!");
    }
}
```

#### Output:

```
Total number of columns are 6
id INT 10
```

```
first VARCHAR 255
```

```
last VARCHAR 255
```

```
age INT 10
```

```
city VARCHAR 255
```

```
salary INT 10
```

```
Success!
```

### ☛ Check Your Progress 1

1. What is a ResultSet Object, and how is it used to fetch the data from the database?

---

---

---

2. Which is the default ResultSet type in the JDBC application?

---

---

---

3. What is ResultSet holdability in JDBC?

---

---

---

4. How to iterate the data in both forward and backward direction?

---

---

---

## 4.3 JDBC TRANSACTIONS

A transaction is a logical unit that may contain one or more SQL query executed in totality to perform a specific task. Generally, any change to the database state is due to a transaction execution like creating, updating or deleting a tuple from the table. It is essential to run the transactions in a controlled environment to ensure system integrity and consistency.

We can understand the transaction concept by taking a simple example of updating the fund transfer from one account to another. If the balance from one account is deducted but couldn't transfer to another, then we would like the first execution to roll back; otherwise, the amount is lost in the cyber-space. This example clarifies that at times we don't like even the first statement should take effect if the second statement didn't execute correctly; else, we will end up with inconsistent data.

The mechanism to ensure that either all dependent queries or batch of queries execute successfully or none is through transactions. In general, a transaction is referred to as a single unit, and the transaction management supports ACID properties – Atomicity, Consistency, Isolation and Durability.

### 4.3.1 Rollback() and commit()

In JDBC by default, after the query execution result will be automatically saved, i.e. after the connection establishment with the database, the connection remains in an auto-commit mode. The auto-commit mode here leads to each statement treated as a transaction, and all updates due to query execution are made permanent. However, at times, to improve the efficiency, we may require to group two or more statements, and for that, we need to turn-off the auto-commit mode. This turning-off of the auto-commit mode is achieved through the setAutoCommit() method of the connection interface.

syntax:

```
jdbcconn.setAutoCommit(false);
```

When we disable the auto-commit mode, no SQL query will commit automatically, and we need to explicitly call the commit() method. This explicit call of jdbcconn.commit() method is through the connection object. Also, if any problem occurs during commit, then a simple set of queries will roll back using jdbcconn.rollback() method.

We can summarize the necessary step needed for any transaction management in JDBC as:

1. We need to change the auto-commit option as false for our connection object.
2. When all the statements in a transaction are completed, we use commit() method to make all the changes permanent.
3. When there is some error while executing statements in a transaction, we need rollback all the changes using rollback() method.

Let us take a simple example to understand the concept of transaction commit and rollback. In this example, we consider the transaction case where the amount from one account is transferred to another account. The MYSQL database is used in this example with the schema “transaction” and table name “account”. The two methods used for transferring amounts are debit() to deduct and credit() to add the amount and

combinedly form a transaction. If we get some error while executing any one of these methods, there will be rollback through connection.rollback() else if all goes well it will commit using connection.commit(). In this example, account number 309 is not available, so the query debited the amount from account 1 is also rollback.

**Example 4.3.1:**

```
import java.sql.*;
import java.sql.DriverManager;
public class transaction
{
    public static void main(String[] args)
    {
        transaction jdbctrans = new transaction();
        jdbctrans.transAmount(1, 309, 10000);
    }
    // Establishing connection using getConnection() Method
    public static Connection getConnection()
    {
        Connection jdbcconn = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            jdbcconn =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/transaction",
            "root", "admin");
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
        return jdbcconn;
    }

    // Transfer Amount @parameter from_Account, to_Account, value
    public void transAmount(int from_Account, int to_Account, int value)
    {
        Connection jdbcconn = null;
        try
        {
            jdbcconn = getConnection();
            // For Transaction execution we make Auto-commit disable
            jdbcconn.setAutoCommit(false);
            // Start the transaction
            debit(jdbcconn, from_Account, value); // deduct the amount from the
            account
            credit(jdbcconn, to_Account, value); // add the amount to the account
            // Complete the transaction
            jdbcconn.commit(); // Commit the transaction
            System.out.println("Transaction Completed successfully");
        }
        catch (SQLException e)
        {
            e.printStackTrace();
            if (jdbcconn != null)
            {
                try
                {
                    jdbcconn.rollback();
                }
                catch (SQLException e1)
                {
                    e1.printStackTrace();
                }
            }
        }
    }
}
```

```

{
System.out.println("Transaction Rollback due to error");
jdbcconn.rollback();
}
catch (SQLException e1)
{
e1.printStackTrace();
}
}
// if condition
}
finally
{
if (jdbcconn != null)
{
// close the connection
try
{
jdbcconn.close();
}
catch (SQLException e)
{
e.printStackTrace();
}
}
// if condition
}
// finally
}

private void debit(Connection jdbcconn, int accountno, int
debit_amount) throws SQLException
{

String debitSQL = "update account as Tab_Deduct JOIN "
+ " (select accountno, (amount - ?) as balance from account" + " where
accountno = ?) As tab "
+ " ON Tab_Deduct.accountno = tab.accountno" + " set
Tab_Deduct.amount = tab.balance"
+ " where Tab_Deduct.accountno = ?";

PreparedStatement jdbcpstmt = null;
try
{
jdbcpstmt = jdbcconn.prepareStatement(debitSQL);
jdbcpstmt.setInt(1, debit_amount);
jdbcpstmt.setInt(2, accountno);
jdbcpstmt.setInt(3, accountno);
int countno = jdbcpstmt.executeUpdate();
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcpstmt != null)
{
jdbcpstmt.close();
}
}
}

```

```
        }
    private void credit(Connection jdbcconn, int accountno, int
    credit_amount) throws SQLException
    {
        String creditSQL = "update account as Tab_Credit JOIN "
        + "(select accountno, (amount + ?) as balance from account" + " where
        accountno = ?) As tab "
        + " ON Tab_Credit.accountno = tab.accountno" + " set
        Tab_Credit.amount = tab.balance"
        + " where Tab_Credit.accountno = ?";

        PreparedStatement jdbcprepstmt = null;
        try
        {
            jdbcprepstmt = jdbcconn.prepareStatement(creditSQL);
            jdbcprepstmt.setInt(1, credit_amount);
            jdbcprepstmt.setInt(2, accountno);
            jdbcprepstmt.setInt(3, accountno);
            int countno = jdbcprepstmt.executeUpdate();
            if (countno == 0)
            {
                throw new SQLException("Account number not found " + accountno);
            }
        }
        finally
        {
            if (jdbcprepstmt != null)
            {
                jdbcprepstmt.close();
            }
        }
    }
}
```

**Output:**

```
java.sql.SQLException: Account number not found 309
Transaction Rollback due to error
    at transaction.credit(transaction.java:97)
    at transaction.transAmount(transaction.java:34)
    at transaction.main(transaction.java:8)
```

### 4.3.2 JDBC Transaction Isolations

The auto-commit disable is essential during the transaction mode as otherwise, we need to hold database locks during the execution of multiple queries and may end up with some sort of deadlock situation. This situation avoided using the concept of transactions, and even transaction management provides some kind of protection for conflicting statements. Generally, the transaction management uses the lock mechanism to deal with the problem of conflicting statements. DBMS uses locks to check transactional access and manage concurrency control by avoiding problems like dirty read etc.

During the concurrent access of the database generally, the dirty read, non-repeatable reads and phantom reads make the database inconsistent. The problem of dirty read occurs when a transaction reads an uncommitted value. The non-repeatable read problem occurs when one Transaction reread the data after an update made by another

transaction. The phantom reads problem happens when records are added or removed by another transaction; then, the transaction is executed on the database. These problems are resolved using a locking mechanism by applying locks and is controlled by the JDBC transaction Isolation level.

There are different transaction levels ranging from TRANSACTION\_NONE, which does not support transactions to TRANSACTION\_SERIALIZABLE, which strictly supports transactions following access rules and handles problems like dirty read non-repeatable reads or phantom reads.

JDBC provides five different transaction-level through Connection interface.

**TRANSACTION\_NONE:** This level does not support Transaction and is denoted by 0.

**TRANSACTION\_READ\_UNCOMMITTED:** - This level support transaction but has no control over dirty read, non-repeatable and phantom reads and is denoted by 1.

**TRANSACTION\_READ\_COMMITTED:** - This level support transaction and prevent dirty read but allows non-repeatable and phantom reads. This level is denoted by 2.

**TRANSACTION\_REPEATABLE\_READ:** - This level support transaction and allow phantom read but prevent dirty and non-repeatable reads. This level is denoted by 4

**TRANSACTION\_SERIALIZABLE:** - This level support transaction and strictly prevents dirty read, non-repeatable reads or phantom reads. This level is denoted by 8.

Let us take an example to understand the isolation level numbers and set the level for the underlying database.

#### Example 4.3.2:

```
import java.sql.*;

public class transIsolation
{
    public static void main(String args[])
    {
        Connection jdbccnn = null;
        try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            jdbccnn =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/transaction",
                "root", "admin");
            System.out.println("Connection established:- " + jdbccnn);
            System.out.println("Transactions levels for the Connection interface
                \"TRANSACTION_NONE\":- "
                + Connection.TRANSACTION_NONE);
            System.out.println("Transactions levels for the Connection interface
                \"TRANSACTION_READ_UNCOMMITTED\":- "
                + Connection.TRANSACTION_READ_UNCOMMITTED);
            System.out.println("Transactions levels for the Connection interface
                \"TRANSACTION_READ_COMMITTED\":- "
                + Connection.TRANSACTION_READ_COMMITTED);
            System.out.println("Transactions levels for the Connection interface
                \"TRANSACTION_REPEATABLE_READ\":- "
                + Connection.TRANSACTION_REPEATABLE_READ);
            System.out.println("Transactions levels for the Connection interface
                \"TRANSACTION_SERIALIZABLE\":- "
                + Connection.TRANSACTION_SERIALIZABLE);
        }
    }
}
```

```
+ Connection.TRANSACTION_SERIALIZABLE);
// Setting the transaction isolation level
jdbccconn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
int setLevel = jdbccconn.getTransactionIsolation();
System.out.println("Set the Transaction isolation level the current
database is:- " + setLevel);
jdbccconn.close();
}
catch (Exception e)
{
System.out.println(e);
}
System.out.println("Success!");
}
```

### Output:

```
Connection established:- com.mysql.cj.jdbc.ConnectionImpl@49ec71f8
Transactions levels for the Connection interface "TRANSACTION_NONE":-
0
Transactions levels for the Connection interface
"TRANSACTION_READ_UNCOMMITTED": - 1
Transactions levels for the Connection interface
"TRANSACTION_READ_COMMITTED": - 2
Transactions levels for the Connection interface
"TRANSACTION_REPEATABLE_READ": - 4
Transactions levels for the Connection interface
"TRANSACTION_SERIALIZABLE": - 8
Set the Transaction isolation level the current database is:- 8
Success!
```

Again, not all the JDBC drivers support the transaction isolation levels so that we can check the sustained status through DatabaseMetaData.supportsTransactionIsolationLevel method. If a driver request of setTransactionIsolation is not supporting a specific isolation level, then the driver can substitute a higher and more restrictive transaction isolation level. Also, if the substitution to higher transaction level fails, it throws an SQLException.

### 4.3.3 JDBC Savepoints

The process of rollback is a saviour at times, but it's not very efficient when a transaction containing a large number of queries to rollback. The Savepoint is one option that sets a Savepoint, i.e. checkpoint in the Transaction and ensures that rollback to only the marked checkpoint and not of the complete Transaction. The connection interface in the current Transaction offers two methods to set Savepoint. The setSavepoint() and setSavepoint (String name) method returns the Savepoint object and creates unnamed Savepoint and given name Savepoint respectively. The rollback() method is overloaded and takes the Savepoint argument. The Savepoint is automatically released on commit, but if we need to release it, we can use the releaseSavepoint() method, which takes the parameter Savepoint.

Let us extend our transaction example to understand the concept of Savepoint. In this example, we preserve the transaction data in another table `all_transactions` with the first column `from_account`, second column as `to_account`, and the transfer value as the third column. The Savepoint is created when the transfer of the amount is completed from

one account to another account. Now, suppose we encounter some error while preserving the transaction details in a new table. In that case, the rollback operation will take place till the checkpoint only and the portion of queries before the Savepoint need not to rollback. This process will avoid the rollback of the complete Transaction.

#### Example 4.3.3:

```

import java.sql.*;
import java.sql.DriverManager;
import java.sql.Savepoint;
public class transSavepoint
{
public static void main(String[] args)
{
transSavepoint jdbctrans = new transSavepoint();
jdbctrans.transAmount(4, 3, 10000);
}
// Establishing connection using getConnection() Method
public static Connection getConnection()
{
Connection jdbcconn = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/transaction",
"root", "admin");
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
catch (SQLException e)
{
e.printStackTrace();
}
return jdbcconn;
}

// Transfer Amount @parameter from_Account, to_Account, value
public void transAmount(int from_Account, int to_Account, int value)
{
Connection jdbcconn = null;
Savepoint jdbcsps = null;
try
{
jdbcconn = getConnection();
// For Transaction execution we make Auto-commit disable
jdbcconn.setAutoCommit(false);
// Start the transaction
debit(jdbcconn, from_Account, value); // deduct the amount from the
account
credit(jdbcconn, to_Account, value); // add the amount to the account
// Complete the transaction
// setting-up save point
jdbcsps = jdbcconn.setSavepoint("DebitCreditDoneSP");
// Preserving transaction
preserveTrans(jdbcconn, from_Account, to_Account, value);
jdbcconn.commit(); // Commit the transaction
}
catch (SQLException e)

```

```
{  
e.printStackTrace();  
if (jdbccconn != null)  
{  
try  
{  
// complete rollback if no Savepoint  
if (jdbcsp == null)  
{  
System.out.println("Transaction Rollback due to error");  
jdbccconn.rollback();  
}  
else  
{  
System.out.println("Transaction Rollback to Savepoint");  
jdbccconn.rollback(jdbcsp);  
// commit the transaction till the Savepoint  
jdbccconn.commit();  
}  
}  
catch (SQLException e1)  
{  
e1.printStackTrace();  
}  
}  
// if condition  
}  
finally  
{  
if (jdbccconn != null)  
{  
// closing the connection  
try  
{  
jdbccconn.close();  
}  
catch (SQLException e)  
{  
e.printStackTrace();  
}  
}  
// if condition  
}  
// finally  
}  
  
private void debit(Connection jdbccconn, int accountno, int debit_amount) throws SQLException  
{  
String debitSQL = "update account as Tab_Deduct JOIN "  
+ " (select accountno, (amount - ?) as balance from account" + " where "  
accountno = ?) As tab "  
+ " ON Tab_Deduct.accountno = tab.accountno" + " set "  
Tab_Deduct.amount = tab.balance"  
+ " where Tab_Deduct.accountno = ?";  
  
PreparedStatement jdbcprepstmt = null;  
try  
{  
jdbcprepstmt = jdbccconn.prepareStatement(debitSQL);  
jdbcprepstmt.setInt(1, debit_amount);  
jdbcprepstmt.setInt(2, accountno);
```

```

jdbcprepstmt.setInt(3, accountno);
int countno = jdbcprepstmt.executeUpdate();
System.out.println("Amount debited and Count of rows updated " +
countno);
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
}

private void credit(Connection jdbcconn, int accountno, int
credit_amount) throws SQLException
{
String creditSQL = "update account as Tab_Credit JOIN "
+ "(select accountno, (amount + ?) as balance from account" + " where
accountno = ?) As tab "
+ " ON Tab_Credit.accountno = tab.accountno" + " set
Tab_Credit.amount = tab.balance"
+ " where Tab_Credit.accountno = ?";

PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(creditSQL);
jdbcprepstmt.setInt(1, credit_amount);
jdbcprepstmt.setInt(2, accountno);
jdbcprepstmt.setInt(3, accountno);
int countno = jdbcprepstmt.executeUpdate();
System.out.println("Amount credited and Count of rows updated " +
countno);
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
}

private void preserveTrans(Connection jdbcconn, int from_account, int
to_account, int value) throws SQLException
{
String preserve = "Insert into all_transaction (from_account,
to_account, value) values ( ?, ?, ?)";
PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(preserve);
}

```

```
jdbcprepstmt.setInt(1, from_account);
jdbcprepstmt.setInt(2, to_account);
jdbcprepstmt.setInt(3, value);
int countno = jdbcprepstmt.executeUpdate();
System.out.println("Count of rows inserted " + countno);
if (countno == 0)
{
throw new SQLException(
"Data insertion Problem - " + " From Account: " + from_account + " To
Account: " + to_account);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
}
```

**Output:**

```
Amount debited and Count of rows updated 1
Amount credited and Count of rows updated 1
java.sql.SQLSyntaxErrorException: Table 'emp.all_transaction' doesn't
exist
    at
com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:
120)
    at
com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(SQLException.java:
97)
    at
com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQL
ExceptionsMapping.java:122)
    at
com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPrepare
dStatement.java:953)
    at
com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientP
reparedStatement.java:1092)
    at
com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientP
reparedStatement.java:1040)
    at
com.mysql.cj.jdbc.ClientPreparedStatement.executeLargeUpdate(ClientPrep
aredStatement.java:1347)
    at
com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedS
tatement.java:1025)
    at transSavepoint.preserveTrans(transSavepoint.java:130)
    at transSavepoint.transAmount(transSavepoint.java:40)
    at transSavepoint.main(transSavepoint.java:8)
Transaction Rollback to Savepoint
```

## ☛ Check Your Progress 2

1. What is a Transaction, and how is it maintained in the JDBC applications?

---

---

---

---

2. What is SavePoint in the JDBC application?

---

---

---

---

3. How do the dirty read, non-repeatable reads and phantom reads problems make the database inconsistent? What mechanisms we can use to resolve them?

---

---

---

---

---

## 4.4 JDBC BATCH PROCESSING

---

A JDBC batch contains a group of SQL statements that are processed together with one database call. JDBC batch is handy when a large number of SQL statements are executed concurrently after putting them in a set. The collection of queries executed with one call. This reduces the network communication overhead and facilitates some of the queries to run in parallel and improve the overall efficiency.

Following are the commonly used methods for Batch Processing:

`addBatch()`: The `addBatch()` method of statement interface is used to add the queries to the batch.

`executeBatch()`: The `executeBatch()` method used to process the batch.

`clearBatch()`: The `clearBatch()` method is used to remove the batch, i.e. it will remove all the queries added in the batch, but we can't remove queries selectively.

The JDBC drivers support is not required for batch processing, but not all the database supports the batch processing, we can check the supported databases through DatabaseMetaData.supportsBatchUpdates() method. If this method returns true implies that the database in context supports the batch processing else, it is not.

In batch processing, the database is executing each update separately, which implies there may be a possibility that out of many queries in the set, one of them may fail. Even if one of the queries fails from the collection, it may lead to an inconsistent database state. All successfully executed queries applied to the database but not those which fails. We can solve this unstable database problem that occurs due to the failure of any query by keeping the batch update inside the JDBC transaction. This step will ensure that the Transaction will execute in totality or none will be updated.

We can summarize the basic step needed to use the JDBC Batch Processing:

1. createStatement() method creates a Statement object.
2. setAutoCommit() method used to set the auto-commit false.
3. addBatch() method to add multiple similar queries into a batch.
4. executeBatch() method to execute all the queries within the batch.
5. commit() method used to commit all the changes.

Let us take an example to understand the concept of batch processing.

#### Example 4.4:

```
import java.sql.*;
import java.sql.DriverManager;
public class batchProcess
{
public static void main(String[] args)
{
Connection jdbcconn = null;
PreparedStatement jdbcpstmt = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
System.out.println("Connecting to database...");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",
"admin");
String insertSQL = "INSERT INTO EMP10"
+ "(id, first, last, age, city, salary) VALUES"
+ "(?, ?, ?, ?, ?, ?)";
jdbcpstmt=jdbcconn.prepareStatement(insertSQL);

jdbcconn.setAutoCommit(false);

jdbcpstmt=dbcconn.prepareStatement(insertSQL);

jdbcpstmt.setInt(1, 101); // 1 specify first parameter
jdbcpstmt.setString(2, "Dinesh");
jdbcpstmt.setString(3, "Sharma");
jdbcpstmt.setInt(4, 40);
jdbcpstmt.setString(5, "Delhi");
jdbcpstmt.setInt(6, 8000);
jdbcpstmt.addBatch();
```

```
jdbcprepstmt.setInt(1, 102);
jdbcprepstmt.setString(2, "Abhimanu");
jdbcprepstmt.setString(3, "Singh");
jdbcprepstmt.setInt(4, 50);
jdbcprepstmt.setString(5, "Mumbai");
jdbcprepstmt.setInt(6, 9000);
jdbcprepstmt.addBatch();

jdbcprepstmt.setInt(1, 105);
jdbcprepstmt.setString(2, "Abhijit");
jdbcprepstmt.setString(3, "Nayak");
jdbcprepstmt.setInt(4, 45);
jdbcprepstmt.setString(5, "Pune");
jdbcprepstmt.setInt(6, 8000);
jdbcprepstmt.addBatch();

jdbcprepstmt.executeBatch();

jdbccconn.commit();
}

catch (SQLException | ClassNotFoundException e)
{
e.printStackTrace();
if(jdbccconn != null)
{
try
{
System.out.println("Transaction Rollback due to error");
jdbccconn.rollback();
}
catch (SQLException e1)
{
e1.printStackTrace();
}
}
// if condition
}
Finally
{
if(jdbccconn != null)
{
//closing the connection
try
{
jdbccconn.close();
System.out.println("Batch executed Successfully!");
}
catch (SQLException e)
{
e.printStackTrace();
}
}
// if condition
}
// finally
}
// main
}
//example
```

ignou  
THE PEOPLE'S  
UNIVERSITY

**Output:**

```
Connecting to database...
Batch executed Successfully!
```

---

## 4.5 JDBC ROWSET INTERFACE

---

The JDBC RowSet interface is an extension of the RowSet interface and is a wrapper around a ResultSet object. RowSet objects inherently derive the capabilities of ResultSet. It allows to use the ResultSet as a JavaBeans component and supports JavaBeans event notification mechanism. It provides the data storage in the tabular form with some additional functionality like sending a notification to other registered components when a particular event is triggered. These additional capabilities make it much flexible and easier to use than a ResultSet.

### RowSet types

The two types of RowSet objects are connected RowSet object and a disconnected RowSet object. A connected RowSet object establishes a connection with the database using a JDBC driver and sustains it till the application terminates. JDBCRowSet implementation is the only standard implementation that offers a connected RowSet object, and thus it is similar to ResultSet object.

On the contrary, a disconnected RowSet object only establishes a connection to the data source for reading data from the ResultSet or writing back to the data source. After executing the query, it disconnects and closes the connection. The other four implementations CachedRowSet, WebRowSet, JoinRowSet and FilteredRowSet, offers disconnected RowSet object. The advantage of a disconnected RowSet object is that they are lightweight as they need not to maintain a permanent connection with the data source but still have all other functionalities of the connected RowSet object. These require a connection to establish every time to ensure the updates, and so it is a little bit slower in comparison to the connected RowSet object. But still, these are not only lightweight but also serializable and thus very suitable for network transmission.

### Implementation Types

RowSet interface offers five implementation classes JDBCRowSet, CachedRowSet, WebRowSet, JoinRowSet and the FilteredRowSet. The RowSet objects are derived from the ResultSet, and it is the only connected RowSet. It offers additional capabilities like Scrollability and Updatability and JavaBeans Component-based development model.

The connectivity between JDBC RowSet and the data source is preserved through its life cycle, which enables JdbcRowSet to take calls that invoke and, in return, call them on the ResultSet object. A ResultSet object can be made scrollable and updatable by the JdbcRowSet object, and this is useful when the driver and database are not offering these properties. The RowSet objects are scrollable and updatable by default and facilitate the ResultSet with these capabilities after populating the RowSet object with the content of the ResultSet, and this implies that we can traverse the records through the ResultSet object.

The RowSet objects are also used as a JavaBeans component and have properties and JavaBeans Notification Mechanism. The RowSet object property have getter and setter methods to fetch and set values like setInt(), getInt() etc. RowSet objects utilize the

JavaBeans event model, in which registered components, i.e. all listeners notified when certain events like cursor movement, insert, update, delete operations or change in row content occur.

RowSets support JavaBeans events like cursorMoved, rowChanged and rowSetChanged. The cursorMoved event initiated due to the next() or previous(), i.e. the rowChanged event-triggered due to a row insert, update, or delete. Whenever the execute method is called to create or change the RowSet, the rowSetChanged event is initiated.

An application component implements a RowSet listener, and when the event occurs, it performs a triggered action. These application components must register the listener objects with a RowSet.addRowSetListener method and implement the standard RowSetListener interface. The RowSet.removeRowSetListener method is used to unregistered the listener.

A CachedRowSet object offers all the necessary capabilities of the JDBC RowSet objects and the disconnected RowSet objects. The other three implementations are the extensions of this interface. The CachedRowSet provides the following in addition to what JDBCRowSet object provides:

1. connection establishment with the data source and query execution
2. reading data from the ResultSet and populating itself with that data
3. manipulating data during disconnected state
4. reconnection and write back to the data source
5. conflict resolution if any

A WebRowSet object provides the capability of reading and writing the XML document in addition to what CachedRowSet object provides:

A JoinRowSet object offers the ability to formulate an equivalent SQL JOIN without connecting to a data source and also provides all the capabilities that WebRowSet object offers.

A FilteredRowSet object provides the filtering capability to enable the visibility of selected data and also offers all the capabilities that WebRowSet object offers.

The event handling is used by calling the instance of **RowSetListener** in the addRowSetListener() method of JdbcRowSet interface. This interface three methods cursorMoved(RowSetEvent event, rowChanged(RowSetEvent event and rowSetChanged(RowSetEvent event).

Let us take an example to understand the concept of JDBC RowSet. In this example, we have fetched the data and performed the cursor movement task; also, we have used the event handling concept, which is not possible using ResultSet.

#### **Example 4.5:**

```
import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;
public class rowSet
{
public static void main(String[] args)
{
try
```

```
{  
  
Class.forName("com.mysql.cj.jdbc.Driver");  
System.out.println("Connecting to database...");  
// RowSet Creation and Execution  
JdbcRowSet jdbcRS = RowSetProvider.newFactory().createJdbcRowSet();  
jdbcRS.setUrl("jdbc:mysql://localhost:3306/emp");  
jdbcRS.setUsername("root");  
jdbcRS.setPassword("admin");  
  
jdbcRS.setCommand("select * from emp10");  
jdbcRS.execute();  
  
// Listener is added  
jdbcRS.addRowSetListener(new MyListener());  
  
while (jdbcRS.next())  
{  
// Event generated for cursor Movement  
System.out.println("Id: " + jdbcRS.getInt(1));  
System.out.println("first: " + jdbcRS.getString(2));  
System.out.println("Salary: " + jdbcRS.getInt(6));  
}  
}  
}  
catch (Exception e)  
{  
//Handle errors for Class.forName  
System.out.println(e);  
}  
// end try  
System.out.println("Executed!");  
}  
// end main  
}  
// end Example  
  
// listener implementation  
class MyListener implements RowSetListener  
{  
public void cursorMoved(RowSetEvent event)  
{  
System.out.println("Cursor is Moved");  
}  
  
public void rowChanged(RowSetEvent event)  
{  
System.out.println("Cursor is Changed");  
}  
public void rowSetChanged(RowSetEvent event)  
{  
System.out.println("RowSet is changed");  
}  
}
```

```
Connecting to database...  
Cursor is Moved  
Id: 501  
first: Aman  
Salary: 8000  
Cursor is Moved  
Id: 502
```

```

first: Ajay
Salary: 9000
Cursor is Moved
Id: 505
first: Suraj
Salary: 8000
Cursor is Moved
Id: 509
first: Gaurav
Salary: 5000
Cursor is Moved
Executed!

```

## 4.6 INTRODUCTION TO JAVA DATA OBJECT (JDO)

Java Data Objects (JDO) is a specification which enables storage and retrieval of persistence data of Java objects. The complex applications require these persistent data or the information to be available beyond the life cycle of the program for various purposes.

JDBC API and JDO API allow us to access data through the java application. In JDBC, the relational database is used as a data source, but JDO uses any data source, maybe a relational database, object-oriented database or even flat files. JDBC mostly uses structured query language, while JDO uses Java code for the purpose.

In the case of JDBC, we have to write SQL queries and fetch the ResultSet into data objects, but JDO doesn't require execution of queries, copying JDBC ResultSet into Java objects etc. as all this is taken care of by JDO.

There are various other options available to the users to store and retrieve persistent data like JDBC, object databases and entity EJBs, but they have their limitations. JDO adds many useful features which are offered by other persistence mechanisms. The JDO provides many features like ease of creating persistence classes, supporting large datasets, also the object-oriented concepts, consistency, concurrency and query capabilities etc. JDO is flexible in terms of data storage as it may be a relational or object-oriented, or simple flat-file database, and the implementation is wholly hidden from the user.

The following are the advantage of using the JDO API:

- Portable: The application based on JDO API are portable and can run on different vendors implementations without recompiling or changing any code.
- Data Source access is transparent: The code to access the data source is independent of the database.
- Ease of use: The JDO API facilitates the users to emphasis on the domain object model and the JDO implementation for the persistence details.
- High performance: JDO implementations look for efficient data access for performance optimization.
- EJB Integration: EJB features are available for the application using the same domain object model throughout the enterprise.

The JDO offers encapsulation, transparency, and portability, and it supports various databases as per the requirements of the application. The JDO API consists of fewer interfaces and a standard for object persistence which makes it simple to implement.

These all features offer greater flexibility to choose deployment environment and fast development.

### Check Your Progress 3

1. What is a JDBC RowSet, and how it is different from ResultSet?

---

---

---

2. What is batch processing in JDBC, and how it improves efficiency?

---

---

---

3. What is Java Data Object API, and how it is different from JDBC API?

---

---

---

## 4.7 SUMMARY

In this unit, we have seen how the JDBC ResultSet interface is used to store the tabular data after the query execution. The data access using cursor subsequently used by the java application. The cursor movement is controlled through various navigational methods. The ResultSet interface also provides concurrency mode to keep the data read-only or in the updatable mode as per the requirements. The ResultSet holdability provide another essential consideration of maintaining the context of data updates through the ResultSet instance.

The batch processing requires the execution of the transaction management, where we have discussed the usage of the commit and rollback methods. The Transaction management for Java applications is needed to satisfy the ACID property and to maintain the consistency we have discussed the various transaction isolation levels. These transaction isolation levels offer multiple methods to check the database inconsistencies and are vital while concurrent access of the database is taking place.

The JDBC RowSet interface provides connected and disconnected RowSet objects. The connected RowSet objects work like the ResultSet object to maintain the connectivity throughout the execution. In contrast, the disconnected one provides the advantage of establishing the connection only while accessing the data read/write and thus saves network usage and also is very lightweight for this reason. We have also discussed various implementation types of the RowSet interface. The Java Data objects are discussed briefly and providing an insight into their usage and advantages over JDBC.

## 4.8 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

### ☞ Check Your Progress 1

1. ResultSet is an object which is automatically created when we execute an SQL query and manage the fetched data from the database in the JDBC applications. With this ResultSet object, a cursor is automatically created to read the data from the object. We need to check the availability of the data before accessing it, so next() method is useful in it, which will return true if the next record is available and move the cursor to that record. We can fetch the data using getter methods.
2. The default ResultSet type in JDBC application is Read only and forward only. In JDBC applications, we can use ResultSet combination of Types and concurrency, and the following are the possible cases:

```
ResultSet.CONCUR_READ_ONLY(), ResultSet.TYPE_FORWARD_ONLY,  
ResultSet.CONCUR_READ_ONLY(), ResultSet.TYPE_SCROLL_SENSITIVE  
ResultSet.CONCUR_READ_ONLY(), ResultSet.TYPE_SCROLL_INSENSITIVE.  
ResultSet.CONCUR_UPDATABLE(), ResultSet.TYPE_FORWARD_ONLY,  
ResultSet.CONCUR_UPDATABLE(), ResultSet.TYPE_SCROLL_SENSITIVE  
ResultSet.CONCUR_UPDATABLE(), ResultSet.TYPE_SCROLL_INSENSITIVE.
```

We can use a particular ResultSet object type based on application requirements, and then we can choose any of the above combinations and pass as a parameter to createStatement() method.

3. The ResultSet holdability evaluates the ResultSet instance if it has a closed status when the underlying connection commit(). There are two types of holdability:
  - a. CLOSE\_CURSORS\_OVER\_COMMIT: All the ResultSet instances are closed on commit()
  - b. HOLD\_CURSORS\_OVER\_COMMIT: All the ResultSet instances are not closed but kept open to accommodate the updates with the same ResultSet on commit().
4. The ResultSet object will use ResultSet.TYPE\_SCROLL\_SENSITIVE and the following two methods to iterate the data in the forward direction and backward direction

Boolean next() and Boolean previous()

The following example explains this concept:

```
import java.sql.*;
import java.sql.DriverManager;
public class forwBack
{
    public static void main(String[] args)
    {
        Connection jdbcconn = null;
        Try
        {
            Class.forName("com.mysql.cj.jdbc.Driver");
            jdbcconn = DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",
            "admin");
            Statement jdbstmt =
            jdbcconn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPD
            ATABLE);

            ResultSet jdbresultset = jdbstmt.executeQuery("Select * from Emp");
            System.out.println("data in forward direction");

            System.out.println("ENO      ENAME      ESAL      EADDR");

            System.out.println("*****");
            while(jdbresultset.next())
            {
                //Retrieve data
                System.out.println("id : " + jdbresultset.getInt("id") + " First : " +
                jdbresultset.getString("first") + " Last : " + jdbresultset.getString("last") + " Age : " +
                jdbresultset.getInt("age") + " City : " + jdbresultset.getString("city") + " Salary : " +
                jdbresultset.getInt("salary"));
            }

            System.out.println("data in forward direction");

            System.out.println("ENO      ENAME      ESAL      EADDR");

            while(jdbresultset.previous())
            {
                //Retrieve data
                System.out.println("id : " + jdbresultset.getInt("id") + " First : " +
                jdbresultset.getString("first") + " Last : " + jdbresultset.getString("last") + " Age : " +
                jdbresultset.getInt("age") + " City : " + jdbresultset.getString("city") + " Salary : " +
                jdbresultset.getInt("salary"));
            }
            jdbcconn.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("Success!");
    }
}
```

## ☛ Check Your Progress 2

1. A transaction is a logical unit containing one or more SQL queries and is executed in totality to perform a specific task. This mechanism is to ensure that either all dependent queries or batch of queries executed successfully or none. In general, transaction management supports ACID properties – Atomicity, Consistency, Isolation and

Durability. By default, every query execution result automatically saves in JDBC applications, i.e. after the connection establishment with the database, the connection remains in an auto-commit mode. But at times, to improve the efficiency, we may require to group two or more statements to form as a part of the Transaction, and for that, we need to turn-off the auto-commit mode. This turning-off of the auto-commit mode is through setAutoCommit() method of the connection interface. This non-commit nature maintained the transactions in our JDBC applications.

2. The SavePoint option sets a Savepoint, i.e. checkpoint in the Transaction, which ensures that if rollback is required in the Transaction, it will only be up to the checkpoint instead of the complete rollback of the Transaction.

3. The concurrent access of the database may lead to the problem of the dirty read, non-repeatable reads and phantom reads which may make the database inconsistent. When a transaction reads an uncommitted value, it is a dirty read problem. When a transaction re-read a data item that is updated by another transaction, it is the non-repeatable read problem. The phantom reads problem happens when records are added or removed by another transaction, and then the transaction is executed on the database. These problems were resolved using a locking mechanism and controlled by the JDBC transaction Isolation level.

There are different transaction levels ranging from TRANSACTION\_NONE, which does not support transactions to TRANSACTION\_SERIALIZABLE, which strictly supports transactions following access rules and handles problems like dirty read, non-repeatable reads or phantom reads. The other Isolation level is TRANSACTION\_READ\_COMMITTED which support transactions and prevent dirty read but allows non-repeatable and phantom reads. The other one is the TRANSACTION\_READ\_UNCOMMITTED which support transactions but has no control over dirty read, non-repeatable and phantom reads. The fifth isolation level is TRANSACTION\_REPEATABLE\_READ which also supports transactions and allow phantom read but prevent dirty and non-repeatable reads.

### Check Your Progress 3

1. The JDBC RowSet interface is an extension of the ResultSet interface. It is a wrapper around a ResultSet object, and inherently derive the capabilities of ResultSet and thus offer more flexibility. A RowSet is broadly divided into Connected RowSet Objects and Disconnected RowSet Objects.

The Connected RowSet Object always remains connected with the database, and it is very similar to the ResultSet object. The Disconnected RowSet Objects are not always connected to a database which makes it very lightweight and serializable.

2. A JDBC batch contains a group of SQL statements that are processed together with one database call. This is very useful when a large number of SQL statements are executed together after putting them in a batch. The batch queries are executed with one call, and this not only reduce the network communication overhead but also some of the queries may run in parallel and thus improve the overall efficiency.

3. JDO API implicitly enables the user to access a database, offering an abstraction of persistence through a standard interface-based Java model. JDBC API and JDO API allow us to access data through the java application. In JDBC, the relational database is used as a data source, but JDO may be using any data source may be a relational database, object-oriented database or even flat files. JDBC mostly uses structured query language, while JDO uses Java code for the purpose. In the case of JDBC, we

have to write SQL queries and fetch the ResultSet into data objects, but JDO doesn't require execution of queries copying JDBC ResultSet into Java objects etc. as all this is taken care by JDO.

## 4.9 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
- ... Sharan, Kishori, "Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs", Apress, 2014.
- ... Parsian, Mahmoud, "DBC metadata, MySQL, and Oracle recipes: a problem-solution approach " Apress, 2006.
- ... <https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>
- ... <https://www.roseindia.net/jdbc/>

