

JavaFX and Advanced Java



JavaFX and Advanced Java

© 2023 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2023



BE AHEAD OF EVERYONE ELSE READ ARTICLES



Online **varsity**

Preface

Welcome to the comprehensive guide on JavaFX and Advanced Java. This book is meticulously crafted to provide a deep dive into Java's advanced concepts, as well as the creation of rich Internet applications using JavaFX. From foundational Java elements such as utility APIs and generics to complex JavaFX features such as UI design, event handling, and more, this book covers a broad spectrum of topics. This book will serve as a valuable companion in your journey to master Java and JavaFX, enabling you to build powerful and immersive applications.

This book is the culmination of the dedicated efforts of the Design Team, consistently committed to delivering the finest and most cutting-edge content in Information Technology. The process of creating this book has adhered to the rigorous standards outlined in the ISO 9001 certification for Aptech-IT Division, Education Support Services.

Your suggestions are highly appreciated in our continuous effort to enhance the quality of our content.

Design Team



Onlinevarsity App for Android devices

Download from Google Play Store

Table of Contents

Sessions

Session 1: Java Utility APIs

Session 2: Generics

Session 3: File Handling, Stream API, and Related Concepts

Session 4: Threading

Session 5: Multithreading and Concurrency

Session 6: JDBC API

Session 7: Advanced JDBC

Session 8: Design Patterns and Other Advanced Features

Session 9: Java Data Structures

Session 10: New and Advanced Features of Java 20

Session 11: Introduction to JavaFX

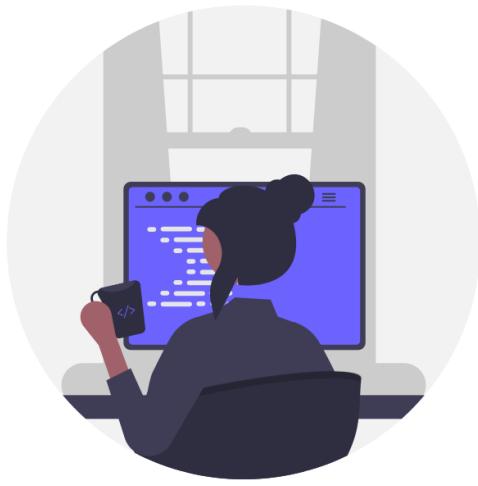
Session 12: JavaFX Text, Transformation, and Shapes

Session 13: JavaFX Layouts, UI, and Charts

Session 14: JavaFX Event Handling

Session 15: Media with JavaFX

Appendix



Session 1

Java Utility APIs

Welcome to the Session, **Java Utility APIs**.

This session will give you an insight into various utility APIs such as Collections API in Java. Collections API is a unified architecture for representing and manipulating collections.

In this Session, you will learn to:

- Explain `java.util` package
- Explain List classes and interfaces
- Explain Set classes and interfaces
- Explain Map classes and interfaces
- Explain Queues and Arrays

1.1 Introduction

The Collections framework consists of collection classes, interfaces, and types that are the primary means by which data collections are manipulated. The framework also includes wrappers and general-purpose implementations. Adapter implementation helps adapt one collection to another. Besides these, there are convenience implementations and legacy implementations.

1.2 `java.util` Package

The `java.util` package contains the definitions of a number of useful classes that provide a broad range of functionality. The package mainly contains collection classes that are useful for working with groups of objects. The package also contains a definition of classes that provides date and time facilities and many other utilities, such as a calendar and dictionary. It also contains a list of classes and interfaces to manage a collection of data in memory. Figure 1.1 displays some of the classes present in `java.util` package.

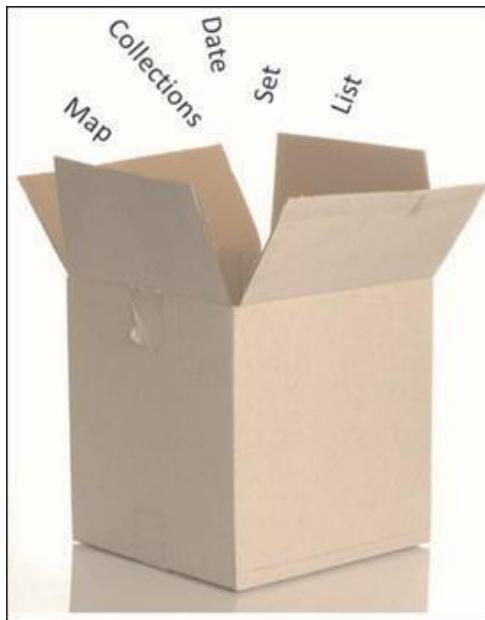


Figure 1.1: `java.util` Package

The classes in this package, such as `Date` and `Calendar`, are explained in following sections.

→ **Date Class, Its Constructors, and Methods**

The `Date` class object represents date and time and provides methods for manipulating the date and time instances. The date is represented as a `long` type that counts the number of milliseconds since January 1, 1970, 00:00:00 GMT. A `Date` object cannot be printed without being converted to a `String` type. The format should conform to the end user's locale, for example, 12.2.95 or 02/12/95.

The constructors of the `Date` class are listed in Table 1.1.

Constructor	Description
<code>Date ()</code>	The constructor creates a <code>Date</code> object using today's date.
<code>Date (long dt)</code>	The constructor creates a <code>Date</code> object using the specified number of milliseconds since January 1, 1970, 00:00:00 GMT.

Table 1.1: Constructors of the Date Class

→ **Calendar Class, Its Constructors, and Methods**

Based on a given `Date` object, the `Calendar` class can retrieve information in the form of integers such as `YEAR`, `MONTH`, and `DAY`. It is an abstract class and, hence, cannot be instantiated like the `Date` class. A `Calendar` object provides all the necessary time field values required to implement the date-time formatting for a particular language and calendar style (for example, German-Gregorian, German-Traditional).

Note: `GregorianCalendar` is a subclass of `Calendar` that implements the Gregorian form of a calendar.

→ **Random Class**

The `Random` class is used to generate random numbers. It is used whenever there is a requirement to generate numbers in an arbitrary or unsystematic fashion. For instance, in a dice game, the outcome of the dice throw is unpredictable. The game can be simulated using a `Random` object.

Two constructors are provided for this class, one taking a seed value (a seed is a number that is used to begin random number generation) as a parameter and the other taking no parameters and using the current time as a seed.

1.2.1 Collections Framework

A collection is a container that helps group multiple elements into a single unit. Collections help to store, retrieve, manipulate, and communicate data.

A Collections framework represents and manipulates collections. It includes the following:

- **Algorithms:** These are methods that are used for performing computation, such as sorting, on objects that implement collection interfaces.

Note: The same method can be applied to different implementations of the appropriate collection interface.

- **Implementations:** These are reusable data structures.
- **Interfaces:** These are abstract data types and represent collections. They allow independent manipulation of collections without requiring the details of their representation.

In Java, the Collections Framework provides a set of interfaces and classes for storing and manipulating groups of objects in a standardized way. Table 1.2 lists some commonly used interfaces and classes in the Collections Framework.

Interface/Class	Description
<code>Collection</code>	The root interface in the collection hierarchy.
<code>List</code>	An ordered collection (also known as a sequence).
<code>Set</code>	A collection that does not allow duplicate elements.
<code>Queue</code>	A collection used to hold elements before processing.
<code>Deque</code>	A double-ended queue that allows insertion and removal at both ends.
<code>Map</code>	An object that maps keys to values (key-value pairs).
<code>SortedSet</code>	A set that maintains its elements in sorted order.
<code>SortedMap</code>	A map that maintains its entries in sorted order.
<code>LinkedList</code>	A doubly-linked list implementation of the <code>List</code> interface.
<code>ArrayList</code>	A resizable array implementation of the <code>List</code> interface.
<code>HashSet</code>	A hash table-based implementation of the <code>Set</code> interface.

Interface/Class	Description
TreeSet	A NavigableSet implementation based on a TreeMap.
PriorityQueue	An unbounded priority queue based on the heap data structure.
ArrayDeque	A resizable array-based implementation of the Deque interface.
HashMap	A hash table-based implementation of the Map interface.
TreeMap	A NavigableMap implementation based on a TreeMap.
LinkedHashMap	A hash table-based implementation of the Map interface with predictable iteration order.
EnumSet	A specialized Set implementation for enum types.
Collections	A utility class that provides static methods for operating on collections.
Arrays	A utility class that provides static methods for manipulating arrays.
Iterator	An interface to traverse through a collection.
ListIterator	An iterator for lists that allows bidirectional traversal.
Comparable	An interface that enables objects to be compared.
Comparator	An interface for custom object comparison.

Table 1.2: List of Collection Interfaces and Classes

The Collections Framework was developed with following objectives in mind:

- It should be high-performance.
- Different types of collections should work in tandem with each other and have interoperability.
- It should be easy to extend a collection.
- There should be minimal effort to learn and use new APIs.
- There should be minimal effort to design new APIs.
- It should reduce the programming effort.
- It should increase the program's speed and quality.
- Software reuse should be promoted.

1.2.2 Collection Interface

The Collections Framework consists of interfaces and classes for working with groups of objects. At the top of the hierarchy lies the Collection interface.

The Collection interface helps to convert the collection's type. This is achieved by using the conversion constructor, which allows the initialization of the new collection to contain the elements present in the specified collection, irrespective of the collection's sub interface.

Note: There are ordered and unordered collections. Certain collections allow duplicate elements.

JDK provides implementations of specific sub interfaces such as Set and List, which are used to

pass collections and manipulate them wherever it is required to be general.

The Collection interface is extended by following sub interfaces:

- **Set**
- **List**
- **Queue**

Collection classes are standard classes that implement the Collection interface and sub interfaces.

Some of the classes provide full implementations of the interfaces, whereas some of the classes provide skeletal implementations and are abstract. Some of the Collection classes are as follows:

- **HashSet**
- **LinkedHashSet**
- **TreeSet**

Figure 1.2 displays the Collection interface and sub interfaces hierarchy.

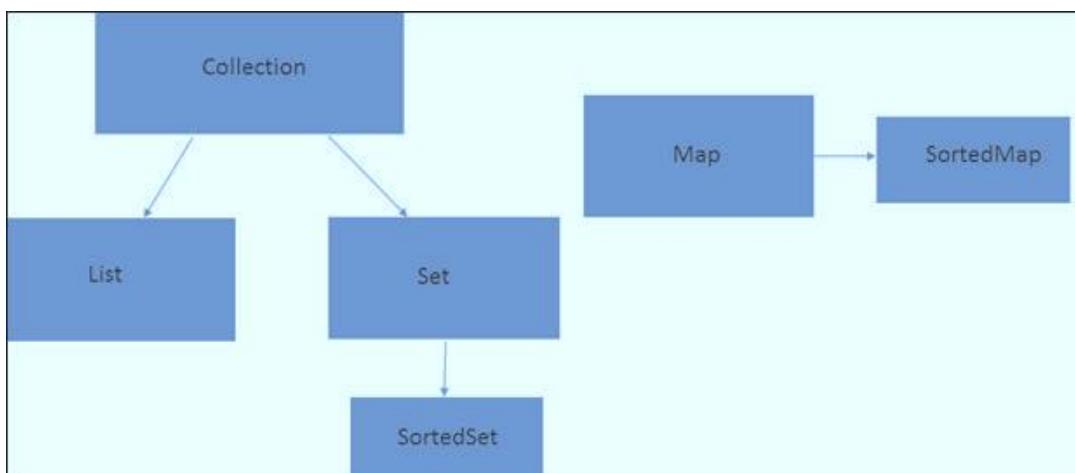


Figure 1.2: Collection Interface

Note: There are two other interfaces, Map and SortedMap, that represent mappings between elements but do not extend Collection interface.

1.2.3 Methods of Collection Interface

A Collection interface helps to pass collections of objects with maximum generality. A collection class holds several items in the data structure and provides various operations to manipulate the contents of the collection, such as adding items, removing, and finding the number of elements.

The interface includes following methods:

- `size()`, `isEmpty()`: Use this to inform about the number of elements that exist in the collection.

- `Contains ()`: Use this to check if a given object is in the collection.
- `add(), remove()`: Use this to add and remove an element from the collection.
- `Iterator ()`: Use this to provide an iterator over the collection.

A few important methods in the `Collection` interface are as follows:

- `add(E obj)`

The method returns true if the specified element is added to the collection.

Syntax

```
public boolean add(E obj)
```

- `contains(Object obj)`

The method returns true if this collection contains the specified element.

Syntax

```
public boolean contains(Object obj)
```

- `isEmpty()`

The method returns true if this collection does not contain any elements.

Syntax

```
public boolean isEmpty()
```

- `size()`

The method returns the number of elements in the collection.

Syntax

```
public int size()
```

Some of the other important methods supported by the `Collection` interface are listed in Table 1.3.

Method	Description
<code>clear()</code>	The method clears or removes all the contents from the collection
<code>toArray()</code>	The method returns an array containing all the elements of this collection

Table 1.3: Methods Supported by Collection Interface

Traversing Collections

Following section describes the two ways to traverse collections:

- **Using the for-each construct:** This helps to traverse a collection or array using a `for` loop. Code Snippet 1 illustrates the use of the `for-each` construct to print out each element of a collection on a separate line.

Code Snippet 1:

```
for (Object obj: collection)
System.out.println(obj);
```

Here, in this snippet, `collection` is assumed to be an object of type `Collection`, such as a `Vector` object or an `ArrayList` object.

The `for-each` construct does not allow to remove the current element by invoking the `remove()` method as it hides the iterator.

- **Using Iterator:** These help to traverse through a collection. They also help to remove elements from the collection selectively.

The `iterator()` method is invoked to obtain an `Iterator` for a collection. The `Iterator` interface includes following methods:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Following points describe the `Iterator` interface:

- The `hasNext()` method returns true if the iteration has more elements
- The `next()` method returns the next element in the iteration.
- The `remove()` method removes the last element that was returned by the `next()` method from the collection. The `remove()` method can be invoked only once for every call to the `next()` method. If this is not followed, the `remove()` method throws an exception.

Note: Only the `Iterator.remove()` method safely modifies a collection during iteration.

Bulk Operations

Bulk operations perform shorthand operations on an entire `Collection` using the basic operations. Table 1.4 describes the methods for bulk operations.

Method	Description
<code>containsAll</code>	This method will return true if the target <code>Collection</code> contains all elements that exist in the specified <code>Collection</code> .
<code>addAll</code>	This method will add all the elements of the specified <code>Collection</code> to the target <code>Collection</code> .
<code>removeAll</code>	This method will remove all the elements from the target <code>Collection</code> that exist in the specified <code>Collection</code> .
<code>retainAll</code>	This method will remove those elements from the target <code>Collection</code> that do not exist in the specified <code>Collection</code> .

Table 1.4: Bulk Operation Methods

The `addAll()`, `removeAll()`, and `retainAll()` methods return true if the target collection was modified in the process of executing the operation.

1.2.4 Collection to Array

One of the basic data structures in Java is an array. A linear data structure that contains elements whose size is defined at the time of creation is called an array. It can hold objects or primitive homogeneous data. A predefined class retaining only heterogeneous object types, but primitive is called a collection. The programmer can use the `List.add()` or the `List.toArray()` methods to convert a Collection into arrays.

Approach 1: Using `List.add()` method

An element E is inserted at a specified position index in the list using `List.add()`.

Syntax

```
public void add (int index, E element);
```

where,

index is where the element is to be inserted.

E is the element that is to be inserted.

The method may cause `IndexOutOfBoundsException` when the index is not in the range.

Code Snippet 2 shows a Java program to change a collection of data in a list to an array. In this example, an `ArrayList` is being created dynamically and the elements "Let's ", "start ", "Power ", "Programming ", "With ", and "Java " are added using `List.add()` method. Subsequently, the list is converted to an array using `toArray()` method which is executed with the help of a `for` loop by printing elements one by one.

Code Snippet 2:

```
import java.util.Arrays;
import java.util.ArrayList;
import java.util.Arrays;
// Or simply add all generic Java libraries
import java.util.*;

public class GFG {
// Main driver method
    public static void main(String[] args)
    {
        // Creating arrayList list dynamically
        List<String> list = new ArrayList<String>();
        // List is created
        // Adding elements to the list
        list.add("Let's ");
        list.add("start ");
```

```

        list.add("Power ");
        list.add("Programming ");
        list.add("With ");
        list.add("Java ");
        // Converting list to an array
        String[] str = list.toArray(new String[0]);
        // Iterating over elements of array
        for (int i = 0; i < str.length; i++) {
            String data = str[i];
            // Printing elements of an array
            System.out.print(data);
        }
    }
}

```

Approach 2: Using `list.toArray()` method

This method is present in the `List` interface that returns all the elements of the list in sequential order as an array.

Syntax

```
public Object[] toArray()
```

`list.toArray()` has following features:

- It is specified by `toArray()` in interface `Collection` and interface `List`
- It overrides `toArray()` in class `AbstractCollection`
- It returns an array containing all the elements in this list in the correct order

Example shown in Code Snippet 3 helps to understand `list.toArray()` method and its usage.

Code Snippet 3:

```

// Importing generic Java libraries
import java.util.*;
import java.io.*;
public class GFG {
    public static void main(String[] args) {
        // Reading input from the user
        // via BufferedReader class
        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));
        // 'in' is object created of this class
        // Creating object of Scanner class
        Scanner sc = new Scanner(System.in);
        // Creating ArrayList to store user input
        List<String> list = new ArrayList<String>();
        // Taking input from user
        // adding elements to the list
        while (sc.hasNext()) {
            String i = sc.nextLine();

```

```

        list.add(i);
    }
    // Converting list to an array
    String[] str = list.toArray(new String[0]);
    // Iteration over array
    for (int i = 0; i < str.length; i++) {
        String data = str[i];
        // Printing the elements
        System.out.print(data);
    }
}
}
}

```

Similar to Code Snippet 2, even in Code Snippet 3, you are converting the list to array. The only difference here is that input is taken from a standard input device by reading the buffer.

After reading the buffer and storing that into the object `in`, the value is put into a list whenever the user hits Enter or types a new line.

1.2.5 UnModifiable Collections

The method `unmodifiableCollection()` of `java.util.Collections` class returns an unmodifiable view of the specified Collection. It allows different modules to offer read-only access to the internal collections. Query operations and any attempt to modify the returned Collection whether via its iterator or directly, will throw an `UnsupportedOperationException`.

In order to preserve the contracts, if the backing collection is a set or a list, the returned Collection will not pass the `hashCode()` or `equals()` operations. It will depend solely on the Object's `hashCode()` and `equals()` methods.

The returned Collection is serializable when the specified Collection is serializable.

Syntax

```

public static <T> Collection<T>
    unmodifiableCollection(Collection<? extends T> c)

```

This method takes the collection as a parameter for which an unmodifiable view is to be returned and returns an unmodifiable view of the specified collection.

Code Snippet 4 illustrates the `unmodifiableCollection()` method.

Code Snippet 4 – Case A: For `unmodifiableCollection()`

```

// Java program to demonstrate
// unmodifiableCollection() method
// for <Character> Value
import java.util.*;

```

```

public class GFG1 {
    public static void main(String[] argv) throws Exception {
        try {
            // creating object of ArrayList<Character>
            List<Character> list = new ArrayList<Character>();
            // populate the list
            list.add('X');
            list.add('Y');
            // printing the list
            System.out.println("Initial list: " + list);
            // getting unmodifiable list
            // using unmodifiableCollection() method
            Collection<Character> immutablelist =
                Collections.unmodifiableCollection(list);
        }
        catch (UnsupportedOperationException e) {
            System.out.println("Exception thrown : " + e);
        }
    }
}

```

Code Snippet 4 – Case B: For UnsupportedOperationException

```

// Java program to demonstrate unmodifiableCollection() method
// for UnsupportedOperationException

import java.util.*;

public class GFG1 {
    public static void main(String[] argv) throws Exception {
        try {
            // creating object of ArrayList<Character>
            List<Character> list = new ArrayList<Character>();
            // populate the list list.add('X');
            list.add('Y');
            // printing the list
            System.out.println("Initial list: " + list);
            // getting unmodifiable list
            // using unmodifiableCollection() method
            Collection<Character> immutablelist =
                Collections.unmodifiableCollection(list);
            // Adding element to new Collection
            System.out.println("\nTrying to modify"+ " the
                unmodifiableCollection");
            immutablelist.add('Z');
        }
        catch (UnsupportedOperationException e) {
            System.out.println("Exception thrown : " + e);
        }
    }
}

```

```
}
```

Code Snippet 4 demonstrates how an exception is thrown when you attempt to add into a collection that cannot be modified. Case A of Code Snippet 4 is placed inside try block, where an attempt to call `list.add()` method to append character is made. However, the collection `list` is `unmodifiableCollection` for character value; therefore, it will throw an exception.

1.3 Lists

The `List` interface is an extension of the `Collection` interface. It defines an ordered collection of data and allows duplicate objects to be added to a list. Its advantage is that it adds position-oriented operations, enabling programmers to work with a part of the list.

The `List` interface uses an index for ordering the elements while storing them in a list. List has methods that allow access to elements based on their position, search for a specific element and return its position, in addition to performing arbitrary range operations. It also provides the `ListIterator` to take advantage of its sequential nature.

1.3.1 Methods of List Interface

The methods supported by the `List` interface are as follows:

→ **`add(int index, E element)`**

The method adds the specified element, at the position specified by index in the list.

Syntax

```
public void add(int index, E element)
```

→ **`addAll(int index, Collection<? extends E> c)`**

The method adds all the elements of the specified collection, `c`, into the list starting at a given index position.

Syntax

```
public boolean addAll(int index, Collection<? extends E> c)
```

→ **`get (int index)`**

The method retrieves an element from the specified index position.

Syntax

```
public E get (int index)
```

→ **`set (int index, E element)`**

The method replaces the element present at the location specified by index in the list with the specified element.

Syntax

```
public E set (int index, E element)
```

→ **`remove(int index) :`**

The method removes the element at given index position from the list.

Syntax

```
public E remove(int index)
```

→ **subList(int start, int end)**

The method returns a list containing elements from start to end – 1 of the invoking list.

Syntax

```
public List<E> subList(int start, int end)
```

Some of the other methods supported by the List interface are listed in Table 1.5.

Method	Description
indexOf(Object o)	The method returns the index of the first occurrence of the specified element in the list, or returns -1 in case the list does not contain the given element.
lastIndexOf(Object o)	The method returns the index of the last occurrence of the specified element in the list, or returns -1 in case the list does not contain the given element.

Table 1.5: Methods of List Interface

1.3.2 ArrayList Class

ArrayList class is an implementation of the List interface in the Collections Framework.

The ArrayList class creates a variable-length array of object references. The list cannot store primitive values such as double. In Java, standard arrays are of fixed length and they cannot grow or reduce their size dynamically. Array lists are created with an initial size. Later, as elements are added, the size increases, and the array grows as required.

The ArrayList class includes all elements, including null. In addition to implementing the methods of the List interface, this class provides methods to change the size of the array that is used internally to store the list. Each ArrayList instance includes a capacity that represents the size of the array. A capacity stores the elements in the list and grows automatically as elements are added to an ArrayList. ArrayList class is best suited for random access without inserting or removing elements from any place other than the end.

An instance of ArrayList can be created using any one of following constructors:

→ **ArrayList()**

The constructor creates an empty list having an initial capacity of 10.

→ **ArrayList(Collection <? extends E> c)**

The constructor creates an array list containing the elements of the specified collection. The elements are stored in the array in the order they were returned by the collection's iterator.

→ **ArrayList(int initialCapacity)**

The constructor creates an empty array list with the specified capacity. The size will grow automatically as elements are added to the array list.

Code Snippet 5 displays the creation of an instance of the `ArrayList` class.

Code Snippet 5:

```
import java.util.ArrayList;
import java.util.List;
public class ListDemo{
    public static void main(String[] args) {
        List<String> listObj = new ArrayList<String> ();
        for (int ctr=1; ctr <= 10; ctr++){
            listObj.add("Value is : " + new Integer(ctr));
        }
        System.out.println("The size is : " + listObj.size());
    }
}
```

In Code Snippet 5, an `ArrayList` object is created, and the first ten numbers are added to the list as objects. The primitive int data type is wrapped by the respective wrapper class.

1.3.3 Methods of `ArrayList` Class

The `ArrayList` class inherits all the methods of the `List` interface. The important methods in the `ArrayList` class are as follows:

→ `add(E obj)`

The method adds a specified element to the end of this list.

Syntax

```
public boolean add(Object E obj)
```

→ `trimToSize()`

The method trims the size of the `ArrayList` to the current size of the list.

Syntax

```
public void trimToSize()
```

→ `ensureCapacity(int minCap)`

The method is used to increase the capacity of the `ArrayList` and ensures that it can hold the least number of specified elements.

Syntax

```
public void ensureCapacity(int minCap)
```

→ `clear()`

The method removes all the elements from this list.

Syntax

```
public void clear()
```

→ `contains(Object obj)`

The method returns true if this list contains the specified element.

Syntax

```
public boolean contains(Object obj)
```

→ `size()`

The method returns the number of elements in this list.

Syntax

```
public int size()
```

Code Snippet 6 displays the use of `ArrayList` class.

Code Snippet 6:

```
import java.util.ArrayList;
import java.util.List;
public class ListDemo{
    public static void main(String[] args) {
        List<String> listObj = new ArrayList<String> ();
        System.out.println("The initial size is : " + listObj.size());

        for (int ctr=1; ctr <= 10; ctr++) {
            listObj.add("Value is : " + new Integer(ctr));
        }
        System.out.println("The size after adding elements is : " +
                           listObj.size());
        listObj.set(5, "Hello World");
        System.out.println("Value of fifth element is: "
                           +(String)listObj.get(5));
    }
}
```

In the code, an empty `ArrayList` object is created, and the initial size of the `ArrayList` object is printed. Then, objects of type `String` are added to the `ArrayList` and the size is printed again. The element present at the fifth position is replaced with the specified element and displayed using the `get()` method.

The initial size is : 0

The size after adding elements is : 10

Value of fifth element is: Hello World

1.3.4 Vector Class

The `Vector` class is similar to an `ArrayList` as it also implements a dynamic array. `Vector` class stores an array of objects, and the size of the array can increase or decrease. The elements in the

Vector can be accessed using an integer index.

Each vector maintains a capacity and a capacityIncrement to optimize storage management. The vector's storage increases in chunks specified by the capacityIncrement as components are added to it.

Note: The amount of incremental reallocation can be reduced by increasing the capacity of a vector before inserting a large number of components.

The difference between the Vector and ArrayList classes is that the methods of Vector are synchronized and thread-safe. This increases the overhead cost of calling these methods. Unlike ArrayList, the Vector class also contains legacy methods that are not part of the collections framework.

The constructors of this class are as follows:

→ **Vector()**

The constructor creates an empty vector with an initial array size of 10.

Syntax

```
public Vector()
```

→ **Vector(Collection<? extends E> c)**

The constructor creates a vector that contains the element of the specified collection, **c**. The elements are stored in the order it was returned by the collection's iterator.

Syntax

```
public Vector(Collection<? extends E> c)
```

→ **Vector(int initCapacity)**

The constructor creates an empty vector whose initial capacity is specified in the variable **initCapacity**.

Syntax

```
public Vector(int initCapacity)
```

→ **Vector(int initCapacity, int capIncrement)**

The constructor creates a vector whose initial capacity and increment capacity is specified in the variables **initCapacity** and **capIncrement** respectively.

Syntax

```
public Vector(int initCapacity, int capIncrement)
```

Code Snippet 7 displays the creation of an instance of the `Vector` class.

Code Snippet 7:

```
 . . .
Vector vecObj = new Vector();
. . .
```

1.3.5 Methods of Vector Class

The vector class defines three protected members. They are as follows:

- int capacityIncrement, which stores the increment value.
- int elementCount, which stores the number of valid components in the Vector.
- Object [] elementData, which is the array buffer into which the components of the vector are stored.

The important methods in the `Vector` class are as follows:

`addElement(E obj)`

The method adds an element at the end of the vector and increases the size of the Vector by 1. The capacity of the Vector increases if the size is greater than the capacity.

Syntax

`public void addElement(E obj)`

`capacity()`

The method returns the present capacity of the vector.

Syntax

`public int capacity()`

`toArray()`

The method returns an array containing all the elements present in the Vector in the correct order.

Syntax

`public Object[] toArray()`

`elementAt(int pos)`

The method retrieves the object stored at the specified location.

Syntax

`public Object elementAt(int pos)`

`removeElement(Object obj)`

The method removes the first occurrence of the specified object from the vector.

Syntax

`public boolean removeElement(Object obj)`

`clear()`

The method removes all the elements of the Vector.

Syntax

`public void clear()`

Code Snippet 8 displays the use of the `Vector` class.

Code Snippet 8:

```
import java.util.Vector;

public class VectorDemo{
    public static void main(String[] args) {
        Vector<Object> vecObj = new Vector<Object>();
        vecObj.addElement(new Integer(5));
        vecObj.addElement(new Integer(7));
        vecObj.addElement(new Integer(45));
        vecObj.addElement(new Float(9.95));
        vecObj.addElement(new Float(6.085));
        System.out.println("The fourth value is: "
            +(Object)vecObj.elementAt(3));
    }
}
```

In Code Snippet 8, a `Vector` is created and initialized with `int` and `float` values after converting them to their respective object types. Finally, the fourth value, which is actually stored at the third position (since index starts from 0), is retrieved and displayed.

1.4 Sets

The `Set` interface creates a list of unordered objects. It creates a non-duplicate list of object references.

1.4.1 Set Interface

The `Set` interface inherits all the methods from the `Collection` interface except those that allow duplicate elements.

The Java platform contains three general-purpose `Set` implementations. They are as follows:

→ **HashSet**

`HashSet` stores its elements in a `Hashtable` and does not guarantee the order of iteration.

→ **TreeSet**

`TreeSet` stores its elements in a tree and orders its elements based on their values. It is slower when compared with `HashSet`.

→ **LinkedHashSet**

`LinkedHashSet` implements `Hashtable` and a linked list implementation of the `Set` interface. It has a doubly linked list running through all its elements and is thus, is different from `HashSet`. Linked list iterates through the elements in which they were inserted into the set (insertion-order).

Comparison of Set with List

The `Set` interface is an extension of the `Collection` interface and defines a set of elements. The difference between `List` and `Set` is that the `Set` does not permit duplication of elements. `Set` is used to create a non-duplicate list of object references. Therefore, the `add()` method returns `false` if duplicate elements are added.

1.5 Maps

A `Map` object stores data in the form of relationships between keys and values. Each key will map to at least a single value. If key information is known, its value can be retrieved from the `Map` object. Keys should be unique but values can be duplicated.

1.5.1 Map Interface

The `Map` interface does not extend the `Collection` interface.

Maps have their own hierarchy, for maintaining the key-value associations. The interface describes a mapping from keys to values, without duplicate keys.

The Collections API provides three general-purpose `Map` implementations:

- `HashMap`
- `TreeMap`
- `LinkedHashMap`

`HashMap` is used for inserting, deleting, and locating elements in a `Map` whereas, `TreeMap` is used to arrange the keys in a sorted order. `LinkedHashMap` is used for defining the iteration ordering which is normally the order in which keys were inserted into the map.

The important methods of a `Map` interface are as follows:

→ `put(K key, V value)`

The method associates the given value with the given key in the invoking map object. It overwrites the previous value associated with the key and returns the previous value linked to the key. The method returns null if there was no mapping with the key.

Syntax

V `put(Object key, Object value)`

→ `get(Object key)`

The method returns the value associated with the given key in the invoking map object.

Syntax

V `get(Object key)`

→ `containsKey(Object Key)`

The method returns true if this map object contains a mapping for the specified key.

Syntax

public boolean `containsKey(Object key)`

→ `containsValue(Object Value)`

The method returns true if this map object maps one or more keys to the specified value.

Syntax

```
public boolean containsValue(Object value)
```

→ size()

The method returns the number of key-value mappings in this map.

Syntax

```
public int size()
```

→ values()

The method returns a collection view of the values contained in this map.

Syntax

```
Collection<V> values()
```

1.6 Stack, Queue, and Arrays

In the `Stack` class, the stack of objects results in a Last-In-First-Out (LIFO) behavior. It extends the `Vector` class to consider a vector as a stack.

`Stack` only defines the default constructor that creates an empty stack. It includes all the methods of the `Vector` class. This class includes following five methods:

`empty()`

This tests if the stack is empty and returns a boolean value of true and false.

`peek()`

This views the object at the top of the stack without removing it from the stack.

`pop()`

This removes the object at the top of this stack and returns that object as the value of the function.

`push(E item)`

This pushes an item on the top of the stack.

`int search(
Object o)`

This returns the 1-based position where an object is on the stack.

Note: When a stack is created, it contains no items.

1.6.1 Queue Interface

A `Queue` is a collection for holding elements that must be processed. In `Queue`, the elements are normally ordered in First In First Out (FIFO) order. The priority queue orders the element according to their values.

A queue can be arranged in other orders too. Every `Queue` implementation defines ordering properties. In a FIFO queue, new elements are inserted at the end of the queue. LIFO queues or

stacks order the elements in LIFO pattern. However, in any form of ordering, a call to the `poll()` method removes the head of the queue.

Queues support the standard collection methods and also provide additional methods to add, remove, and review queue elements.

A queue helps track asynchronous message requests while a stack helps traverse a directory tree structure.

1.6.2 Deque

A double ended queue is commonly called deque. It is a linear collection that supports insertion and removal of elements from both ends.

Note: Deque is pronounced as deck.

Usually, Deque implementations have no restrictions on the number of elements to include. However, it does support capacity-restricted deques. The methods are inherited from the Queue interface. A deque when used as a queue results in FIFO behavior. Elements are added at the end of the deque and removed from the start. A deque when used as LIFO stack should be used in preference with the legacy Stack class. Deques when used as stack has their elements pushed and popped from the beginning of the deque. Stack methods are equivalent to Deque methods.

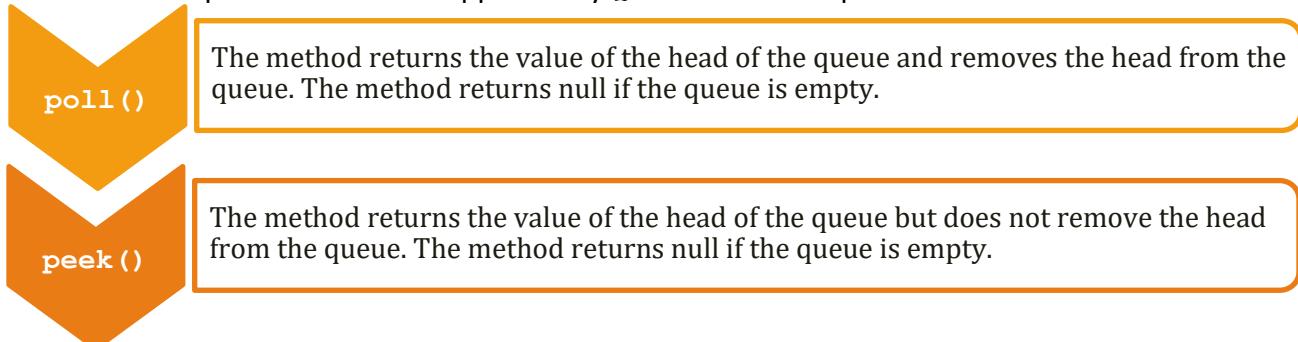
The Deque interface and its implementations when used with the Stack class provides a consistent set of LIFO stack operations. Code Snippet 9 displays this.

Code Snippet 9:

```
Deque<Integer> stack1 = new ArrayDeque<Integer>();
```

1.6.3 Methods Supported by Queue

Some of the important methods supported by Queue and its implementations are as follows:





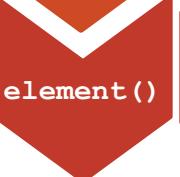
`remove()`

The method returns the value of the head of the queue and removes the head from the queue. It throws an exception if the queue is empty.



`offer(E obj)`

The method inserts the specified element into the queue and returns true if it was possible to add the element else it returns false.



`element()`

The method returns the value of the head of the queue but does not remove the head from the queue. The method throws an exception if the queue is empty. Each of the described methods can throw an exception if the operation fails or return a null or

1.6.4 PriorityQueue Class

Priority queues are similar to queues but the elements are not arranged in FIFO structure. They are arranged in a user-defined manner. The elements are ordered either by natural ordering or according to a comparator. A priority queue does not allow adding of non-comparable objects nor does it allow null elements. A priority queue is unbound and allows the queue to grow in capacity.

When the elements are added to a priority queue, its capacity grows automatically. The constructors of this class are as follows:

→ `PriorityQueue()`

The method constructs a `PriorityQueue` and orders its elements according to their natural ordering. The default capacity is 11.

→ `PriorityQueue(Collection<? extends E> c)`

The constructor creates a `PriorityQueue` containing elements from the specified collection, `c`. The initial capacity of the queue is 110% of the size of the specified collection.

→ `PriorityQueue(int initialCapacity)`

The constructor creates a `PriorityQueue` with the specified initial capacity and orders its elements according to their natural ordering.

→ `PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`

The constructor creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.

→ `PriorityQueue(PriorityQueue<? extends E> c)`

The constructor creates a `PriorityQueue` containing elements from the specified collection. The initial capacity of the queue is 110% of the size of the specified collection.

→ `PriorityQueue(SortedSet<? extends E> c)`

The constructor creates a `PriorityQueue` containing the elements from the specified collection. The initial capacity of the queue is 110% of the size of the specified collection.

1.6.5 Methods of PriorityQueue Class

The `PriorityQueue` class inherits the method of the `Queue` class.

The other methods supported by the `PriorityQueue` class are listed in Table 1.6.

Method	Description
<code>add(E e)</code>	The method adds the specific element to the priority queue and returns a boolean value.
<code>clear()</code>	The method removes all elements from the priority queue.
<code>comparator()</code>	The method returns the comparator used to order this collection. The method will return null if this collection is sorted according to its elements natural ordering.
<code>contains (Object o)</code>	The methods return a boolean value of true if the queue contains the specified element.
<code>iterator()</code>	The method returns an iterator over the elements in the queue.
<code>toArray()</code>	The method returns an array of objects containing all the elements in the queue.

Table 1.6: Methods Supported by `PriorityQueue`

Code Snippet 10 displays the use of the `PriorityQueue` class.

Code Snippet 10:

```
import java.util.*;
public class PriorityDemo{
public static void main(String[] args) {
    PriorityQueue<String> queue = new PriorityQueue<String>();
    queue.offer("New York");
    queue.offer("Kansas");
    queue.offer("California");
    queue.offer("Alabama");
    System.out.println("1. " + queue.poll()); // removes item
    System.out.println("2. " + queue.poll()); // removes item
    System.out.println("3. " + queue.peek());
    System.out.println("4. " + queue.peek());
    System.out.println("5. " + queue.remove()); // removes item
    System.out.println("6. " + queue.remove()); // removes item
    System.out.println("7. " + queue.peek());
    System.out.println("8. " + queue.element()); // throws
                                                //Exception
}
}
```

Output

1. Alabama
2. California
3. Kansas
4. Kansas
5. Kansas

6. New York

7. null

```
Exception in thread "main" java.util.NoSuchElementException at  
java.util.AbstractQueue.element(Unknown Source)  
at QueueTest.main(QueueTest.java:24)
```

In the code, a `PriorityQueue` instance is created which will store `String` objects. The `offer()` method is used to add elements to the queue. The `poll()` and `remove()` method is used to retrieve and return the values from the queue. The `peek()` method retrieves the value but does not remove the head of the queue. When at the end, the `element()` method is used to retrieve the value, an exception is raised as the queue is empty.

1.7 Summary

- The `java.util` package contains the definitions of a number of useful classes providing a broad range of functionality.
- The `List` interface is an extension of the `Collection` interface.
- `ArrayList` is an implementation of the `List` interface and is part of the Java Collections Framework in the `java.util` package.
- `ArrayList` can dynamically resize itself to accommodate more elements when its capacity is reached, using an internal array.
- `Vector` is a legacy class in the Java Utilities API that predates `ArrayList` but provides similar functionality.
- `Vectors` can store elements of any type, including objects and primitives (using their wrapper classes).
- The `Set` interface creates a list of unordered objects.
- A `Map` object stores data in the form of relationships between keys and values.
- A `Queue` is a collection for holding elements before processing.
- `ArrayDeque` class does not put any restriction on capacity and does not allow null values.

1.8 Check Your Progress

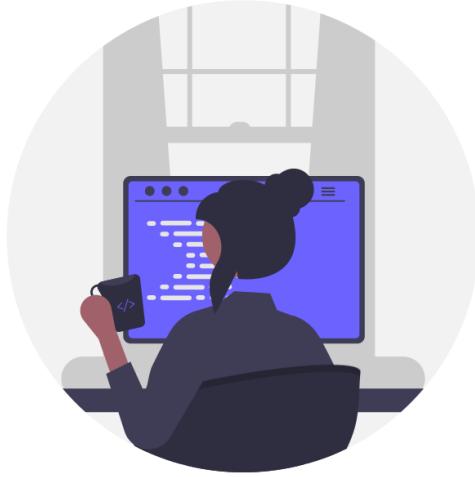
1. Which of these statements related to `java.util` package are true?
 - A. Collection is a container that helps to group multiple elements into a single unit.
 - B. Collections Framework provides classes and interfaces for storing and manipulating groups of objects in a standardized way.
 - C. Collections Framework was developed to have low degree of interoperability.
 - D. Collections Framework does not contain `Collection` interface at the top of the hierarchy.
2. Which of these statements regarding Lists are true?
 - A. Lists allow access to elements based on their position.
 - B. Lists do not have a method to search for a specific element.
 - C. `ArrayList()` creates an empty array.
 - D. In a `Vector`, the size of the vector can be increased or decreased.
3. The `getFirst()` method ____.
 - A. Retrieves, but does not remove the last element of the list.
 - B. Retrieves, but does not remove the first element of the list.
 - C. Removes and returns the first element of the list.
 - D. Removes and returns the last element of the list.
4. Which of these statements about different Set classes are true?
 - A. Set class does not permit duplication of elements.
 - B. SortedSet class allows duplication of elements.
 - C. HashSet class guarantee the order of elements.
 - D. LinkedHashSet maintains the order of the items added to the Set.
5. Which of these statements related to different classes implementing Map interface are true?
 - A. TreeMap is used to arrange the keys in a sorted order.
 - B. HashMap allows null values to be used as keys.
 - C. TreeMap stores elements in a tree structure.
 - D. LinkedHashMap is a combination of hash map and linked list.
 - E. LinkedHashMap returns values in the order they were added to the Map.

1.8.1 Answers

1. A, B
2. A
3. B
4. A
5. B, C

Try It Yourself

1. Create an ArrayList of integers and add following numbers: 5, 10, 15, 20, 25. Print the elements of the ArrayList in reverse order.
2. Create a LinkedList of strings and add following names: "Alice", "Bob", "Charlie", "David", "Eva". Remove "Charlie" from the LinkedList and then, print the updated list.
3. Create a HashSet of integers and add following numbers: 10, 20, 30, 40, 50, 10, 30. Print the elements of the HashSet and notice if any duplicates are present.
4. Create a TreeSet of strings and add following names: "Alice", "Bob", "Charlie", "David", and "Eva". Print the elements of the TreeSet and observe that they are sorted in alphabetical order.



Session 2 Generics

Welcome to the Session, **Generics**.

This session describes Generics feature that was first added to the Java programming language as a part of J2SE 5.0. The `java.util` package contains the collections framework, legacy collection classes, event model, date time facilities, and internationalization.

In this Session, you will learn to:

- Identify the necessity for Generics
- List the advantages and limitations of Generics
- Explain generic class declaration and instantiation
- Define generic methods
- Describe working with generic methods
- Explain the wildcard argument
- Describe the use of inheritance with Generics
- Explain type inference

2.1 *Introduction*

Genericity is a way by which programmers can specify the type of objects that a class can work with via parameters passed at declaration time and evaluated at compile time. Generic types can be compared with functions that are parameterized by type variables and can be instantiated with different type arguments depending on the context.

2.1.1 *Generics Overview*

Generics in Java code generates one compiled version of a generic class. The introduction of Generics in Java classes will help remove the explicit casting of a class object so the `ClassCastException` will not arise during compilation. Generics will help to remove type inconsistencies during compile time rather than at runtime.

Generics are added to the Java programming language because they enable:

Getting more information about a collection's type.

Keeping track of the type of elements a collection contains.

Using casts all over the program.

Note: Generics are checked at compile time for type correctness. In Generics, a collection is not considered as a list of references to objects. You can distinguish the difference between a collection of references to integers and bytes. A generic type collection requires a type parameter that specifies the type of element stored in the collection.

Generic is a technical term in Java that denotes the features related to the use of generic methods and types. To know the Collection's element type was the main motivation of adding Generics to Java programming language. Earlier, Collections treated elements as a collection of objects. To retrieve an element from a Collection required an explicit cast, as downward cast could not be checked by the compiler. Thus, there was always a risk of runtime exception, `ClassCastException`, to be thrown if you cast a type, which is not a supertype of the extracted type.

Generics allow the programmer to communicate the type of a collection to the compiler so that it can be checked. Thus, using Generics is safe as during compilation of the program, the compiler consistently checks for the element type of the collection and inserts the correct cast on elements being taken out of the collection. Code Snippet 1 displays the code for non-generic code.

Code Snippet 1: (for Non-generic code)

```
import java.util.LinkedList;  
...  
LinkedList list = new LinkedList();  
list.add(new Integer(1));  
Integer num = (Integer) list.get(0);
```

In the code, an instance of `LinkedList` is created. An element of type `Integer` is added to the list. While retrieving the value from the list, an explicit cast of the element was required.

Code Snippet 2 displays the code for generic code.

Code Snippet 2:

```
import java.util.LinkedList;  
...  
LinkedList<Integer> list = new LinkedList<Integer>();  
list.add(new Integer(1));  
Integer num = list.get(0);
```

In the code, the `LinkedList<>` is a generic class which accepts an `Integer` as type parameter. The compiler checks for the type correctness during compile time. It is not necessary to cast an `Integer` because the compiler inserts the correct cast on elements being retrieved from the list

using the `get()` method.

2.1.2 Advantages and Limitations of Generics

Advantages and limitations of Generics are as follows:

Advantages of Generics

Generics allow flexibility of dynamic binding.

Generic type helps the compiler to check for type correctness of the program at the compile time.

In Generics, the compiler-detected errors are less time-consuming to fix than runtime errors.

The code reviews are simpler in Generics as the ambiguity is less between containers.

In Generics, codes contain lesser casts and thus help to improve readability and robustness.

Limitations of Generics

In Generics, you cannot create generic constructors.

A local variable cannot be declared where the key and value types are different from each other.

2.2 Generic Classes

Generic class enables a Java programmer to specify a set of related types with a single class declaration. A generic type is parameterized over types. It is a generic class or interface.

A generic class is a mechanism to specify the type relationship between a component type and its object type. During the creation of class instances the concrete type for a class parameter will be determined. So, the component type defined by class parameter depends on a concrete instantiation. A subclass relationship cannot be established between a generic class and its instances. The runtime polymorphism on a generic class is not possible, so variables cannot be specified by generic class.

A generic class declaration resembles a non-generic class declaration. However, in the generic class declaration, the class name is followed by a type parameter section.

The syntax for declaring a generic class is same as ordinary class except that in angle brackets (`<>`) the type parameters are declared. The declaration of type parameters follows the class name. Type parameters are like variables and can have the value as a class type, interface type, or any other type variable except primitive data type. The class declaration such as `List<E>` denotes a class of

generic type.

A generic class allows a series of objects of any type to be stored in the generic class, and makes no assumptions about the internal structure of those objects. The parameter to the generic class (`Integer` in an Array [`Integer`]) is the class given in the array declaration and is bound at compile time. A generic class can thus generate many types, one for each type of parameter, such as `Array` [`Tree`], `Array` [`String`], and so on. Generic classes can accept one or more type parameters. Therefore, they are called parameterized classes or parameterized types. The type parameter section of a generic class can include several type parameters separated by commas.

2.2.1 Declare and Instantiate Generic Class

To create an instance of the generic class, the `new` keyword is used along with the class name except that the type parameter argument is passed between the class name and the parentheses.

The type parameter argument is replaced with the actual type when an object is created from a class. A generic class is shared among all its instances.

Syntax

```
class NumberList <Element> { ... }
```

Code Snippet 3 displays the declaration of a generic class.

Code Snippet 3:

```
public class NumberList <T> {
    private T obj;
    public void add(T val) {
        . . .
    }
    public static void main(String [] args) {
        NumberList<String> listObj = new NumberList<String> ();
        . . .
    }
}
```

The code creates a generic type class declaration with a type variable, `T` that can be used anywhere in the class. To refer to this generic class, a generic type invocation is performed which replaces `T` with a value such as `String`.

A user can specify a type variable as any non-primitive type, which can be any class type, any array type, any interface type, or even another type variable.

Typically, type parameter names are single, uppercase letters. Following are the commonly used type parameter names:

- K - Key
- T - Type

- V - Value
- N - Number
- E - Element
- S, U, V and so on

Code Snippet 4 illustrates how a class can be declared and initialized.

Code Snippet 4:

```
import java.util.*;
public class TestQueue <DataType> {
    private LinkedList<DataType> items = new
        LinkedList<DataType>();
    public void enqueue(DataType item) {
        items.addLast(item);
    }
    public DataType dequeue() {
        return items.removeFirst();
    }
    public boolean isEmpty() {
        return (items.size() == 0);
    }
    public static void main(String[] args) {
        TestQueue<String> testObj = new TestQueue<>();
        testObj.enqueue("Hello");
        testObj.enqueue("Java");
        System.out.println((String) testObj.dequeue());
    }
}
```

The required type arguments can be replaced to invoke the constructor of a generic class with an empty set of type arguments (<>). The pair of angle brackets (<>) is called diamond. It is important to note that the compiler should determine the type arguments from the context when using the empty set of type arguments. A fragment of code from Code Snippet 4 is highlighted in Code Snippet 5. Here, an instance of `TestQueue` will accept `String` as a type parameter.

Code Snippet 5:

```
TestQueue<String> testObj = new TestQueue<>();
```

In the code shown in Code Snippet 4, a generic class is created that implements the concept of queue, and dequeue on any datatype such as `Integer`, `String`, or `Double`. The type variable or type parameter, `<DataType>`, is used for the argument type and return type of the two methods. Type parameters can have any name. Type parameters can be compared to formal parameters in subroutines. The name will be replaced by the actual name when the class will be used to create an instance. Here, `<DataType>` has been replaced by `String` within the `main()` method while instantiating the class. Figure 2.1 displays the output of Code Snippet 4.

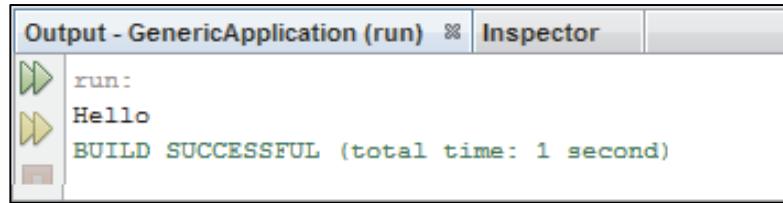


Figure 2.1: TestQueue – Output

2.3 Generic Methods

Java also supports generic methods. Generic methods are defined for a particular method and have the same functionality as the type parameter has for generic classes. Generic methods can appear in generic classes as well as in non-generic classes. Generic method can be defined as a method with type parameters. Generic methods are best suited for overloaded methods that perform identical operations for different argument types. The use of generic methods makes the overloaded methods more compact and easier to code.

A generic method allows type parameters used to make dependencies among the type of arguments to a method and its return type. The return type does not depend on the type parameter, or any other argument of the method. This shows that the type argument is being used for polymorphism. The scope of the method's type parameter is the method declaration. The scope of type parameters can also be typing parameters of other type parameters.

Syntax

```
public<T> void display(T[] val)
```

Code Snippet 6 displays the use of a generic method.

Code Snippet 6:

```
public class NumberList <T> {  
    public<T> void display(T[] val) {  
        for( T element : val)  
        {  
            System.out.printf("Values are as follows: %s \n" ,  
                element);  
        }  
    }  
    public static void main(String [] args) {  
        Integer[] intValue = {1, 7, 9, 15};  
        NumberList<Integer> listObj = new NumberList<> ();  
        listObj.display(intValue);  
    }  
}
```

This code uses a generic method, `display()`, that accepts an array parameter as its argument.

Figure 2.2 displays the output.

```

GenericApplication.NumberList > display > for (T element : val) >
Output - GenericApplication (run) <
run:
Values are: 1
Values are: 7
Values are: 9
Values are: 15
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 2.2: NumberList – Output

A generic class can have two or more type parameters. Code Snippet 7 demonstrates how to declare a class with two type parameters.

Code Snippet 7:

```

import java.util.*;
class TestQueue<DataType1, DataType2> {
    private final DataType2 num;
    private LinkedList<DataType1> items = new LinkedList<>();
    public TestQueue(DataType2 num) {
        this.num = num;
    }
    public void enqueue(DataType1 item) {
        items.addLast(item);
    }
    public DataType1 dequeue() {
        return items.removeLast();
    }
}

```

Like any other class, generic classes can also be subclassed by either generic or non-generic classes. A non-generic subclass can extend a superclass by specifying the required parameters and thus, making it concrete. Code Snippet 8 demonstrates how to declare a non-generic subclass and makes use of the class defined in Code Snippet 7.

Code Snippet 8:

```

public class MyTest extends TestQueue<String, Integer> {
    public MyTest(Integer num) {
        super(num);
    }
    public static void main(String[] args) {
        MyTest test = new MyTest(new Integer(10));
        test.enqueue("Hello");
        test.enqueue("Java");
        System.out.println((String) test.dequeue());
    }
}

```

In the code, a non-generic subclass is created named **MyTest** with a concrete generic instantiation **MyTest extends TestQueue<String, Integer>**. Figure 2.3 displays the output.

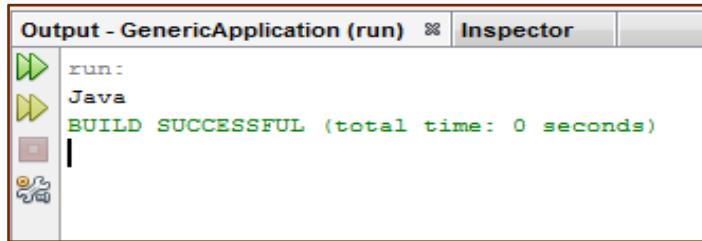


Figure 2.3: GenericApplication - Output

Code Snippet 9 demonstrates the creation of a generic subclass. Here, it is assumed that **TestQ** is defined as a generic class with a single generic type parameter.

Code Snippet 9:

```
...
class MyTestQ <DataType> extends TestQ<DataType> {
    public static void main(String[] args) {
        MyTestQ<String> test = new MyTestQ<String>();
        test.enqueue("Hello");
        test.enqueue("Java");
        System.out.println((String) test.dequeue());
    }
}
```

2.3.1 Declaring Generic Methods

To create generic methods and constructors, type parameters are declared within the method and constructor signature. The type parameter is specified before the method return type and within angle brackets. The type parameters can be used as argument types, return types, and local variable types in generic method declarations. There can be more than one type of type parameters, each separated by a comma. These type parameters act as placeholders for the actual type argument's data types, which are passed to the method. Primitive data types cannot be represented for type parameters.

Code Snippet 10 displays the generic methods present in the `Collection` interface.

Code Snippet 10:

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
}
```

In the `containsAll` and `addAll` methods shown in Code Snippet 10, the type parameter `T` is used only once. For constructors, the type parameters are not declared in the constructor, but in the header that declares the class. The actual type parameters are passed while invoking the constructor.

Code Snippet 11 demonstrates how to declare a generic class containing a generic constructor.

Code Snippet 11:

```
import java.util.*;
public class StudPair<T, U> {
    private T name;
    private U rollNumber;
    public StudPair(T nmObj, U rollNo) {
        this.name = nmObj;
        this.rollNumber = rollNo;
    }
    public T displayName() {
        return name;
    }
    public U displayNumber() {
        return rollNumber;
    }
    public static void main(String [] args) {
        StudPair<String, Integer> studObj = new
            StudPair<>("John",2);
        System.out.println("Student Name:");
        System.out.println(studObj.displayName());
        System.out.println("Student Number:");
        System.out.println(studObj.displayNumber());
    }
}
```

In the code, a generic constructor is declared containing two generic type parameters separated by a comma. Figure 2.4 displays the output.

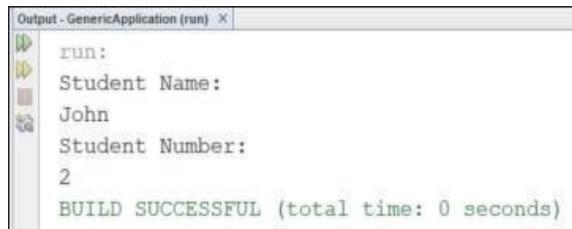


Figure 2.4: StudPair – Output

2.3.2 Return Generic Types

A method can also return generic data type. Code Snippet 12 displays a method having a generic return type.

Code Snippet 12:

```
import java.util.*;
public class GenericReturnTest {
    public static <T extends Comparable<T>> T maxValueDisplay(T val1, T
        val2, T val3) {
        T maxValue = val1;
        if (val2.compareTo(val1) > 0)
            maxValue = val2;
```

```

        if (val3.compareTo(maxValue) > 0)
    maxValue = val3;
    return maxValue;
}
/** 
 * @param args the command line arguments
 */
public static void main(String[] args) {
    System.out.println(maxValueDisplay(23, 42, 1));
    System.out.println(maxValueDisplay("apples", "oranges",
    "pineapple"));
}
}

```

In the code, the `compareTo()` method of the `Comparable` class is used to compare values which can be `int`, `char`, `String`, or any data type. The `compareTo()` method returns the maximum value.

Figure 2.5 displays the output.

```

Output - GenericReturnTest (run) ✘ Inspector Terminal
run:
42
pineapple
BUILD SUCCESSFUL (total time: 0 seconds)
>>

```

Figure 2.5: GenericReturnTest - Output

2.4 Type Inference

In generic methods, it is the type inference that helps to invoke a generic method. Here, there is no required to specify a type between the angle brackets.

Type inference enables the Java compiler to determine the type arguments that make the invocation applicable. It analyzes each method invocation and corresponding declaration to do so. The inference algorithm determines the following:

- ➔ Types of the arguments
- ➔ The type that the result is being returned
- ➔ The most specific type that works with all of the arguments

The type arguments required to invoke the constructor of a generic class can be replaced with an empty set of type parameters (`<>`) if the compiler infers the type arguments from the context. Code Snippet 13 illustrates this.

Code Snippet 13:

```

Map<String, List<String>> myMap = new HashMap<String,
List<String>>();

```

2.4.1 Generic Constructors of Generic and Non-Generic Classes

Constructors can declare their own formal type parameters in both generic and non-generic classes. Code Snippet 14 illustrates this.

Code Snippet 14:

```
class MyClass<X> {  
<T> MyClass(T t) {  
// ...  
}  
}
```

Code Snippet 15 shows the instantiation of the class `MyClass`.

Code Snippet 15:

```
MyClass objMyClass = new MyClass<Integer>("");
```

In the code snippet:

- The statement creates an instance of the parameterized type `MyClass<Integer>`.
- The statement specifies the type `Integer` for the formal type parameter, `X`, of the generic class
- `MyClass<X>`.
- The constructor for the generic class contains a formal type parameter, `T`.
- The compiler understands the type `String` for the formal type parameter, `T`, of the constructor of this generic class. This is because the actual parameter of the constructor is a `String` object.

2.4.2 Enhancements in Java SE 7 Onwards

Following are enhancements in Java SE 7 onwards:

- **Underscores in Numeric Literals:** Underscore characters (`_`) can be added anywhere between digits in a numerical literal to separate groups of digits in numeric literals. They improve the readability of the code.
- **Strings in switch Statements:** The `String` class can be used in the expression of a `switch` statement.
- **Binary Literals:** In Java SE 7 and higher versions, the integral types can be defined using the binary number system. Note: The integral types are `byte`, `short`, `int`, and `long` type. To specify a binary literal, add the prefix `0b` or `0B` to the number.
- **Better Compiler Warnings and Errors with Non-Reifiable Formal Parameters:** In Java, a type whose type information is fully available at runtime is called a **reifiable** type. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards. A non-reifiable type, on the other hand, is one whose information has been removed at compile-time by type erasure. A non-reifiable type does not have all its information available at runtime. The compiler in Java SE 7 onwards generates a warning at the declaration site of a `varargs` method or constructor with a non-reifiable `varargs` formal parameter. Java SE 7 and later version compilers suppress these warnings using the compiler option `-Xlint:varargs` and the annotations `@SafeVarargs` and `@SuppressWarnings({ "unchecked", "varargs" })`.
- **Type Inference for Generic Instance Creation:** The required type arguments can be replaced

to invoke the constructor of a generic class with an empty set of type parameters if the compiler infers the type arguments from the context.

- **Catching Multiple Exception Types:** A single `catch` block handles many types of exceptions. Users can define specific exception types in the `throws` clause of a method declaration because the compiler executes accurate analysis of rethrown exceptions.
- **The try-with-resources Statement:** This declares one or more resources, which are objects that should be closed after the programs have finished working with them. Any object that implements the `new java.lang.AutoCloseable` interface or the `java.io.Closeable` interface can be used as a resource. The statement ensures that each resource is closed at the end of the statement.

Note: `java.io.InputStream`, `OutputStream`, `Reader`, `Writer`, `java.sql.Connection`, `Statement`, and `ResultSet` classes can implement the `AutoCloseable` interface and can be used as resources in a try-with-resources statement.

2.5 Collection and Generics

A collection object in Java is an object that manages a group of objects. Collection API depends on generics for its implementation. Code Snippet 16 illustrates this.

Code Snippet 16:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class GenericArrayListExample {
    public static void main(String[] args) {
        List<Integer> partObj = new ArrayList<>(3);
        partObj.add(new Integer(1010));
        partObj.add(new Integer(2020));
        partObj.add(new Integer(3030));
        System.out.println("Part Numbers are as follows: ");
        Iterator<Integer> value = partObj.iterator();
        while (value.hasNext()) {
            Integer partNumberObj = value.next();
            int partNumber = partNumberObj.intValue();
            System.out.println("" + partNumber);
        }
    }
}
```

Figure 2.6 displays the output that displays the usage of collection API and generics.

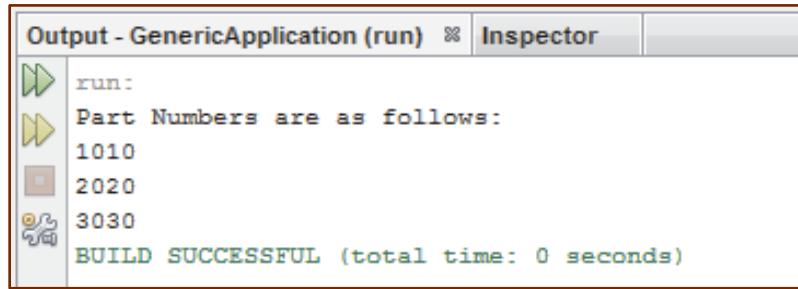


Figure 2.6: GenericArrayListExample - Output

Using an invalid value with generics results in compile time error.

2.5.1 Wildcards with Generics

Wildcards are used to declare wildcard parameterized types. A wildcard is used as an argument for instances of generic types. Wildcards are useful where no or only a little knowledge about the type argument of a parameterized type is available.

There are three types of wildcards used with Generics.

➤ ?

The wildcard character '?' represents an unknown type in Generics. It denotes the set of all types or any one type. So, `List <?>` means that the list contains unknown object type. The unbounded wildcard (?) is used as argument for instantiations of generic types. The unbounded wildcard is useful in situations where no knowledge about the type argument of a parameterized type is required.

Code Snippet 17 displays the use of unbounded wildcard.

Code Snippet 17:

```
public class Paper {
    public void draw (Shape shapeObj) {
        shapeObj.draw(this);
    }
    public void displayAll(List<Shape> shObj) {
        for(Shape s: shObj) {
            s.draw(this);
        }
    }
}
```

Consider that the `Paper` class contains a method that displays all the shapes which is represented as a list. If the method signature of `displayAll()` method is as specified in the code then, it can be invoked only on lists of type `Shape`. The method cannot be invoked on `List<Circle>`. It is assumed that `Shape` has been created earlier.

➤ ? extends Type

The bounded wildcard ? extends Type represents an unknown type that is a subtype of the bounding class. The word 'Type' specifies an upper bound, which states that the value of the type parameter must either extend the class or implement the interface of the bounding class. It denotes a family of subtypes of type Type.

So, `List<? extends Number>` means that the given list contains objects which are derived from the `Number` class. In other words, the bounded wildcard using the `extends` keyword limits the range of types that can be assigned. This is the most useful wildcard.

Code Snippet 18 displays the declaration of the bounded wildcard ? extends Type.

Code Snippet 18:

```
public void displayAll(List<? extends Shape> shape) {  
}
```

The code declares a method that accepts a list with any kind of `Shape`. The method accepts lists of any class which is a subclass of `Shape`. The `displayAll()` method can accept as its argument, `List<Circle>`.

➤ ? super Type

The bounded wildcard ? super Type represents an unknown type that is a supertype of the bounding class. The word Type specifies a lower bound. The wildcard character denotes a family of supertypes of type Type. So, `List<? super Number>` means that it could be `List<Number>` or `List<Object>`.

Code Snippet 19 displays the use of ? super Type bounded wildcard.

Code Snippet 19:

```
public class NumList {  
    public static <T> void copy(List <? super T> destObj, List<? extends T> srcObj) {  
        for (int ctr=0; ctr<srcObj.size(); ctr++) {  
            destObj.set(ctr, srcObj.get(ctr));  
        }  
    }  
}
```

In the method <? super T> indicates that the destination object, `destObj`, may have elements of any type which is a supertype of T. Similarly, the statement <? extends T> means that the source object, `srcObj`, may have elements of any type that is a subtype of T.

Note that this class does not have a `main` so you cannot execute it directly. You can call the `copy` method in another class.

Figure 2.7 displays different types of wildcards.

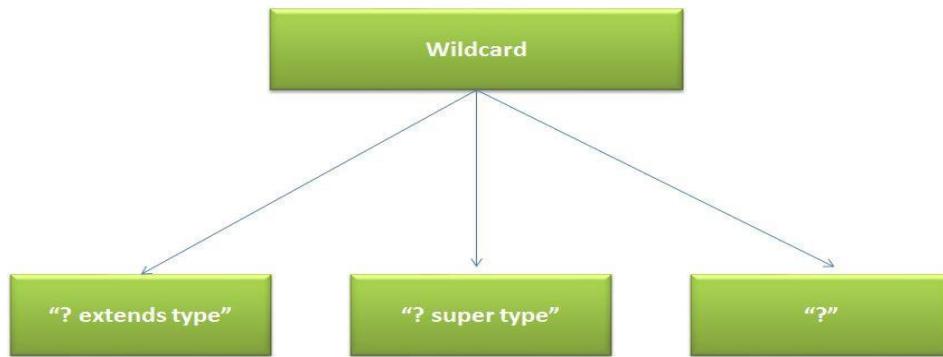


Figure 2.7: Wildcards

Note: For ? extends Type, elements can be retrieved from the structure but elements cannot be added.

2.5.2 Exception Handling with Generics

Exceptions provide a reliable mechanism for identifying and responding to error conditions. The `catch` clause present with a `try` statement checks that the thrown exception matches the given type. A compiler cannot ensure that the type parameters specified in the `catch` clause match the exception of unknown origin as an exception is thrown and caught at runtime. Thus, the `catch` clause cannot include type variables or wildcards. A subclass of `Throwable` class cannot be made generic as it is not possible to catch a runtime exception with compile time parameters intact.

In Generics, the type variable can be used in the `throws` clause of the method signature. Figure 2.8 displays the exception handling with Generics.



Figure 2.8: Exception Handling with Generics

Code Snippet 20 displays the use of generic type with exceptions.

Code Snippet 20:

```

interface Command<X extends Throwable> {
    public void calculate(Integer arg) throws X;
}

public class ExTest implements Command <ArithmaticException> {
    public void calculate(Integer num) throws ArithmaticException
    {
        int no = num.valueOf(num);
        System.out.println("Value is: " + (no/0));
    }
}

```

The code shows how to use genericity in Exception. The code uses a type variable in the throws clause of the method signature. During implementation, the type parameter X is substituted with ArithmeticException showing that the code may throw an arithmetic exception.

2.5.3 Inheritance with Generics

Inheritance is a mechanism to derive new classes or interfaces from the existing ones. Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. Classes can extend generic classes and provide values for type parameters or add new type parameters. A class cannot inherit parametric type. Two instantiations of the same generic type cannot be used in inheritance.

Code Snippet 21 displays the use of generics with inheritance.

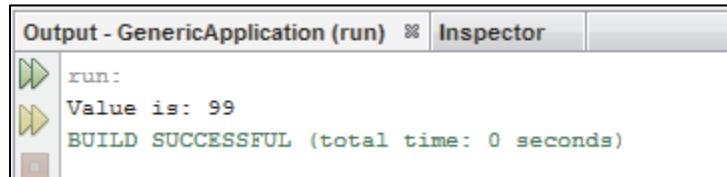
Code Snippet 21:

```
class Month<T> {
    T monthObj;
    Month(T obj) {
        monthObj = obj;
    }
    // Return monthObj
    T getob() {
        return monthObj;
    }
}
// A subclass of Month that defines a second type parameter,
// called V.
class MonthArray<T, V> extends Month<T> {
    V valObj;
    MonthArray(T obj, V obj2) {
        super(obj);
        valObj = obj2;
    }

    V getob2() {
        return valObj;
    }
}
public class HierTest {
    public static void main(String args[]) {
        MonthArray<String, Integer> month;
        // Create an object of type MonthArray.
        month = new MonthArray<>("Value is: ", 99);
        System.out.print(month.getob());
        System.out.println(month.getob2());
    }
}
```

In the code, the subclass **MonthArray** is the concrete instance of the class **Month<T>**.

Figure 2.9 displays the output.



```
Output - GenericApplication (run) ✘ Inspector
run:
Value is: 99
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 2.9: HierTest – Output

2.6 Interoperability with Generics

An existing piece of code can be modified to use Generics without making all the changes at once. The design of Generics ensures that new Java libraries can still be used for compilation of old code. In other words, the same piece of code will work with both legacy and generic versions of the library. This is known as platform compatibility. This upward compatibility provides the programmer with the freedom to move from legacy to generic version whenever required.

In Java, genericity ensures that the same class file is generated by both legacy and generic versions with some additional information about types. This is known as binary compatibility as the legacy class file can be replaced by the generic class file without recompiling. Figure 2.10 displays the adaptation between the legacy code and the new code.

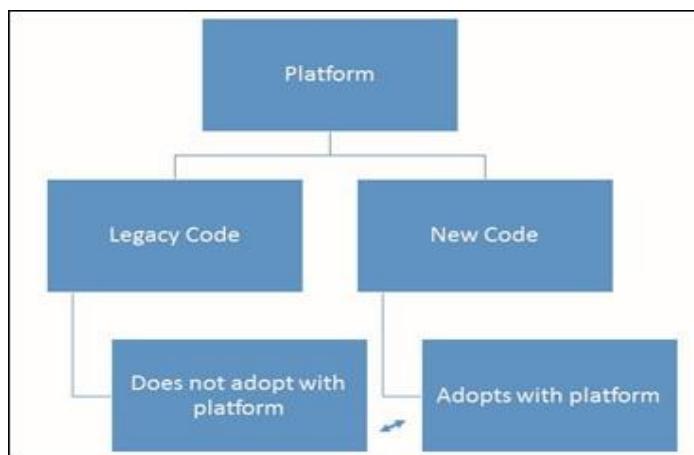


Figure 2.10: Legacy Code

Code Snippet 22 displays the use of legacy code with legacy client.

Code Snippet 22 (Legacy code with Legacy Client)

```
import java.util.ArrayList;
import java.util.List;
interface NumStack {
    public boolean empty();
    public void push(Object elt);
    public Object retrieve();
}
```

```

class NumArrayStack implements NumStack {
    private List listObj;
    public NumArrayStack() {
        listObj = new ArrayList();
    }
    @Override
    public boolean empty() {
        return listObj.size() == 0;
    }
    @Override
    public void push(Object obj) {
        listObj.add(obj);
    }

    @Override
    public Object retrieve() {
        Object value = listObj.remove(listObj.size() - 1);
        return value;
    }
    @Override
    public String toString() {
        return "stack" + listObj.toString();
    }
}
public class Client {
    public static void main(String[] args) {
        NumStack stackObj = new NumArrayStack();
        for (int ctr = 0; ctr<4; ctr++) {
            stackObj.push(new Integer(ctr));
        }
        assert stackObj.toString().equals("stack[0, 1, 2, 3]");
        int top = ((Integer)stackObj.retrieve()).intValue();
        System.out.println("Value is : " + top);
    }
}

```

Code Snippet 22 displays the use of legacy code and a legacy client. In the code, data is added to a stack and retrieved from the stack (without making use of the `Stack` class, since that was introduced only from Java 7 onwards). A newer version of this code would use the `Stack` class and other newer features too.

Note: It is difficult to write a useful program that is totally independent of its working platform. Consequently, when a platform is upgraded, the earlier code becomes a legacy code that may no longer work.

2.7 Summary

- Generics in Java code generate one compiled version of a generic class.
- Generics help to remove type inconsistencies during compile time rather than at runtime.
- There are three types of wildcards used with Generics namely, "? extends Type","? super Type", and "?".
- Generic methods are defined for a particular method and have the same functionality as the type parameters have for generic classes.
- Type parameters are declared within the method and constructor signature when creating generic methods and constructors.
- A single generic method declaration can be called with arguments of different types.
- Type inference enables the Java compiler to determine the type arguments that make the invocation applicable.

2.8 Check Your Progress

1. Which of these statements stating the advantages of Generics are true?

A.	Generics allow flexibility of dynamic binding.
B.	In Generics you cannot create generic constructors.
C.	In Generics the compiler detected errors are less time consuming to fix than runtime errors.
D.	A local variable cannot be declared where the key and value types are different from each other.
E.	The code reviews are simpler in Generics as the ambiguity is less between containers.

2. Which of the following statements explaining the use of wildcards with Generics are true?

A.	"?" denotes the set of all types or any one type in Generics.
B.	"? extends Type" denotes a family of subtypes which extends "Type".
C.	"? extends Type" denotes a family of subtypes of type "Type".
D.	"? super Type" denotes a family of supertypes which is a subtype of "Type".
E.	"? super Type" denotes a family of supertypes of type "Type".

3. Which of the following statements specifying the use of Legacy code in Generics are true?

A.	Java generates an unchecked conversion warning when a value of raw type is passed where a parameterized type is expected.
B.	Erasure adds all generic type information.
C.	When a generic type such as collection is used without a type parameter, it is called a raw type.
D.	Erasure removes all generic type information.
E.	The change in method signature can be performed by making minimal change or by creating a skeleton.

4. When an existing piece of code works with both legacy and generic versions of the library, this is called _____.

(A)	Platform compatibility	(C)	Binary compatibility
(B)	Upward compatibility	(D)	Parameterized type

5. In the generic class declaration, the class name is followed by a _____.

(A)	Binary literals	(C)	Type parameter section
(B)	String object	(D)	Numeric literals

6. Which one of the following prefix should be added to the number to specify a binary literal?

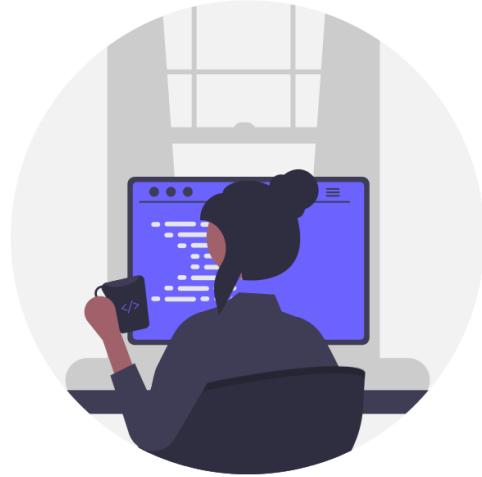
(A)	ONL	(C)	OBL
(B)	ON	(D)	OB

2.8.1 Answers

1. A and C
2. A and D
3. C
4. A
5. C
6. D

Try It Yourself

1. Create a generic class called **Box** that can hold any type of object. Implement methods to add and retrieve items from the box. Instantiate the **Box** class with different types and demonstrate how it maintains type safety.
2. Create a method that accepts a list of any type and prints its elements using wildcards. Implement a generic method to handle exceptions when dealing with collections.
3. Create a generic class **Shape<T>** and a subclass **Circle<T>** that inherits from **Shape**. Demonstrate how to use them with different types.
4. Implement a generic class called **Pair** that holds a pair of elements of any type. Include methods to set and retrieve the elements. Instantiate the **Pair** class with different types and demonstrate how to access the elements.
5. Write a generic method called **swap** that takes an array and two indices as arguments and swaps the elements at those indices. Create a generic method called **printList** that accepts a list of any type and prints its elements using a for-each loop.



Session 3

File Handling, Stream API, and Related Concepts

Welcome to the Session, **File Handling, Stream API, and Related Concepts**.

This session covers the `java.io` package that contains various classes known as Input/Output (I/O) streams. This session also describes the `Console` class.

In this Session, you will learn to:

- Define data streams
- Identify purpose of the `File` class and its methods
- Explain `DataInput` and `DataOutput` interfaces
- Outline byte stream and character streams in `java.io` package
- Explain `InputStream` and `OutputStream` classes
- Describe `BufferedInputStream` and `BufferedOutputStream` classes
- Identify Character stream classes
- Define Serialization and identify the necessity and purpose of Serialization
- Elaborate on `Console` class
- Identify Unsafe Operations on foreign memory
- Describe uses of foreign functions

3.1 Introduction

Java works with streams of data. A stream is defined as a logical entity that produces or consumes information. A physical data storage is mapped to a logical stream and then, a Java program reads data from this stream serially – one byte after another, or character after character. In other words, a physical file can be read using different types of streams, for example, `FileInputStream`, or `FileReader`. Java uses such streams to perform various input and output operations.

Necessity for Stream Classes and Interfaces

In Java, streams are required to perform all the input/output (I/O) operations. An input stream receives data from a source into a program and an output stream sends data to a destination from

the program.

Thus, stream classes and interfaces help in:

- Reading input from a stream
- Writing output to a stream
- Managing disk files
- Share data with a network of computers

3.2 Stream Classes and Interfaces

The standard input/output stream in Java is represented by three fields of the `System` class:

in	out	err
The standard input stream is used for reading characters of data. This stream responds to keyboard input or any other input source specified by the host environment or user.	The standard output stream is used to typically display the output on the screen or any other output medium.	This is the standard error stream. By default, this is the user's console.

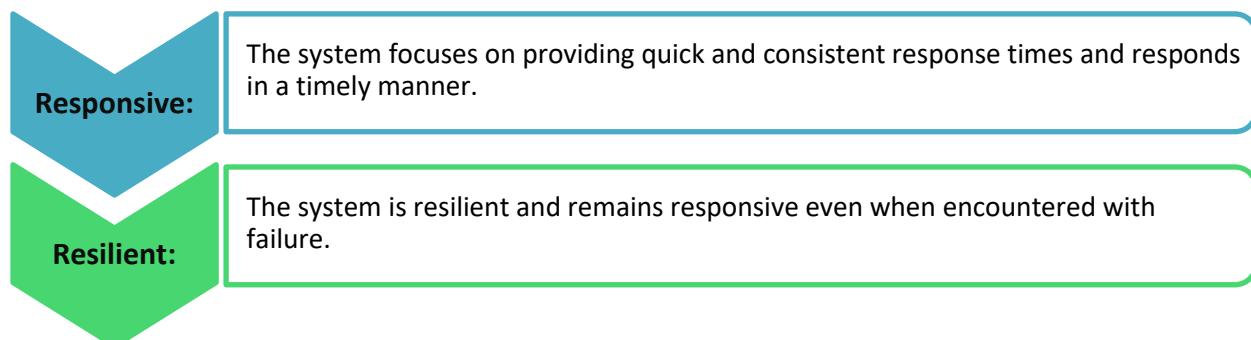
To read or write data using Input/Output streams, following steps required to be performed:

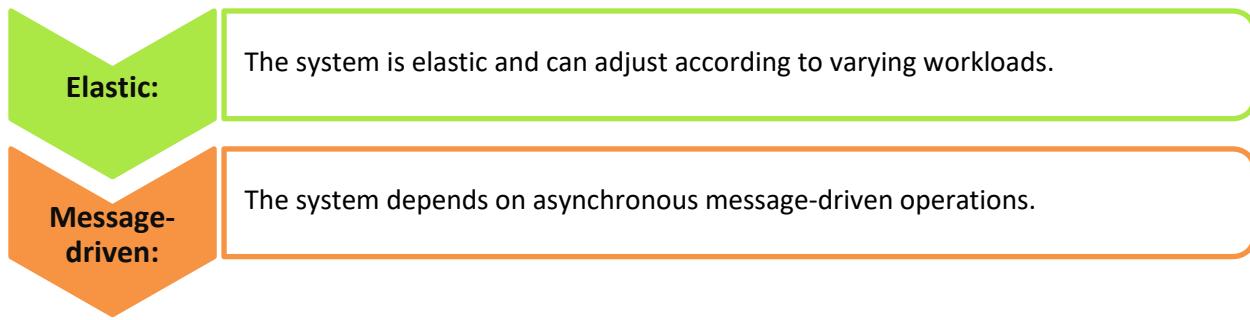
- Open a stream that points at a specific data source: a file, a socket, URL, and so on.
- Read or write data from/to this stream.
- Close the stream.

`InputStream` and `OutputStream` are abstract classes and are used for reading and writing of unstructured sequence of bytes. The other input and output streams are subclasses of these abstract classes and are used for reading and writing to a file. Different types of byte streams can be used interchangeably as they inherit the structure of `Input/OutputStream` class. For reading or writing bytes, a subclass of the `InputStream` or `OutputStream` class has to be used respectively.

3.2.1 Reactive Streams

Reactive programming is a programming paradigm, similar to how object-oriented programming or functional programming are programming paradigms. According to the Reactive Manifesto (which is a guideline to build modern, highly flexible-scale architectures), any reactive application must adhere to four key characteristics:





In Java, Reactive Streams provide a common API to implement reactive programming. RxJava and Akka Streams are the popular implementations of Reactive Streams.

3.3 takeWhile, dropWhile, ofNullable, and iterate with Condition

In Java, it is possible to perform comprehensive operations having pattern of objects with the introduction of streams. Following methods are used to refine streams:

→ **takeWhile (Predicate Interface)**

Syntax

```
default Stream<T> takeWhile (Predicate<? super T> predicate)
```

`takeWhile()` method receives values till the predicate shows false. The `takeWhile()` method picks the longest prefix of elements from a stream matching the given predicate in an ordered stream. An example for `takeWhile()` method is shown in Code Snippet 1.

Code Snippet 1:

```
import java.util.stream.Stream;
public class StreamDemo {
public static void main(String[] args) {
Stream.of("H", "o", "w", " are", " you", "", friend? ", "end", "next") .
    takeWhile(s->!s.equals("end"))
    .forEach(System.out::print);
}
}
```

The `takeWhile()` method takes all values until the string matches "end". Then, it stops executing.
The output is as follows:

How are you, friend?

The string "next" is not displayed in the output because the `takeWhile()` iteration has stopped before reaching it.

→ **dropWhile (Predicate Interface)**

The `dropWhile` method removes the entire range of values initially till the predicate shows up true. In an ordered stream, this method returns leftover elements after removing the longest prefix of elements matching the given predicate.

Syntax

```
default Stream<T> dropWhile (Predicate<? super T> predicate)
```

An example for `dropWhile` is shown in Code Snippet 2.

Code Snippet 2:

```
import java.util.stream.Stream;
public class Tester {
public static void main(String[] args) {
Stream.of("g","h","i","","k","l").dropWhile(s->!s.isEmpty())
.forEach(System.out::print);
System.out.println();
Stream.of("g","h","i","","k","","l").dropWhile(s->!s.isEmpty())
.forEach(System.out::print);
}
}
```

The `dropWhile()` method drops values g, h, and i; then, it takes all the values once the string is empty.

Output:

```
kl  
kl
```

→ iterate

The `iterate()` method stops the loop when `hasNext` predicate shows value as false.

Syntax

```
static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext,
UnaryOperator<T> next)
```

An example for `iterate` is shown in Code Snippet 3.

Code Snippet 3:

```
import java.util.stream.IntStream;
public class Tester {
public static void main(String[] args) {
IntStream.iterate(4, x->x<11, x->x+4).forEach(
System.out::println);
}
}
```

The code is simple and self-explanatory. The output will be:

```
4  
8  
10
```

In Java 10, the collectors `Collector.toUnmodifiableList()` and `Collector.toUnmodifiableSet()` were added to create unmodifiable lists and sets.

These help ensure immutability and make streams more suitable for parallel processing.

In Java 12, `Collector.teeing()` method was added, which allows combining two collectors and returning their results as a tuple. It simplifies the process of applying multiple collectors to a stream in one pass.

3.4 File Class and Its Methods

Unlike other classes that work on streams, `File` class directly works with files and the file system.

When instances of the `File` class are created, the abstract pathname represented by a `File` object never changes. In other words, objects of `File` class are immutable.

`File` class encapsulates access to information about a file or a directory. In other words, `File` class stores the path and name of a directory or file. All common file and directory operations are performed using the access methods provided by the `File` class. Methods of this class allow to create, delete, and rename files, provide access to the pathname of the file, determine whether any object is a file or directory, and checks the read and write access permissions. The directory methods in the `File` class allow creating, deleting, renaming, and listing of directories.

The interfaces and classes defined by the `java.nio.file` package helps the Java Virtual Machine to access files, file systems, and file attributes. The `toPath()` method helps to obtain a `Path` that uses the abstract path. A `File` object uses this path to locate a file. To diagnose errors when an operation on a file fails, the `Path` can be used with the `Files` class.

The constructors of the `File` class are as follows:

➤ **`File(String dirpath)`**

The `File(String dirpath)` constructor creates a `File` object with pathname of the file specified by the `String` variable `dirpath` into an abstract pathname. If the string is empty, then it results in an empty abstract pathname.

➤ **`File(String parent, String child)`**

The `File(String parent, String child)` constructor creates a `File` object with pathname of the file specified by the `String` variables `parent` and `child`. If `parent` string is null, then, the new `File` instance is created by using the given `child` pathname string.

➤ **`File(File fileobj, String filename)`**

The `File(File fileobj, String filename)` creates a new `File` instance from another `File` object specified by the variable `fileObj`, and file name specified by the `String` variable `filename`.

➤ **`File(URL urlobj)`**

The `File(URL urlobj)` converts the given `file:URI` into an abstract pathname and creates a new `File` instance. The `file:URI` is system-dependent. This makes the transformation by the constructor also system-dependent.

Code Snippet 4 displays the creation of an instance of the `File` class.

Code Snippet 4:

```
...
File fileObj = new File("/tmp/myFile.txt");
OR
File fileObj = new File("/tmp", "myFile.txt");
```

3.4.1 Methods in `File` Class

The methods in `File` class help to manipulate the file on the file system. Some of the methods in the `File` class are listed in Table 3.1.

Method	Description
<code>renameTo(File newname)</code>	The <code>renameTo(File newname)</code> method will name the existing <code>File</code> object with the new name specified by the variable <code>newname</code> .
<code>delete()</code>	The <code>delete()</code> method deletes the file represented by the abstract path name.
<code>exists()</code>	The <code>exists()</code> method tests the existence of file or directory denoted by this abstract pathname.
<code>getPath()</code>	The <code>getPath()</code> method converts the abstract pathname into a pathname string.
<code>isFile()</code>	The <code>isFile()</code> method checks whether the file denoted by this abstract pathname is a normal file.
<code>createNewFile()</code>	The <code>createNewFile()</code> method creates a new empty file whose name is the path name for this file. It is only created when the file of similar name does not exist.

Table 3.1: Methods in `File` Class

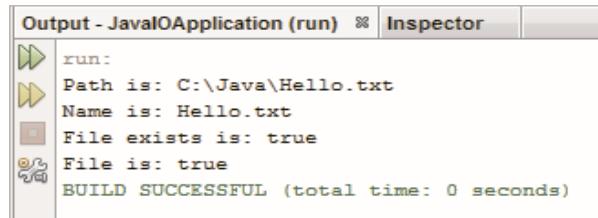
Code Snippet 5 displays the use of methods of the `File` class.

Code Snippet 5:

```
import java.io.*;
public class FileTest {
public static void main(String[] args) {
File fileObj = new File("C:/Java Codes/Hello.txt");
System.out.println("Path is: " +fileObj.getPath());
System.out.println("Name is: " +fileObj.getName());
System.out.println("File exists is: " +fileObj.exists());
System.out.println("File is: " +fileObj.isFile());
}
}
```

Code Snippet 5 displays the full path and the filename of the invoking `File` object. The code also checks for the existence of the file and returns true if the file exists, false if it does not.

The `isFile()` method returns true if called on a file and returns false if called on a directory. Figure 3.1 displays the output.



```
Output - JavaIOApplication (run)  ✘ Inspector
run:
Path is: C:\Java\Hello.txt
Name is: Hello.txt
File exists is: true
File is: true
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.1: File Class Methods - Output

Code Snippet 6 displays the use `FilenameFilter` class to filter files with a specific extension.

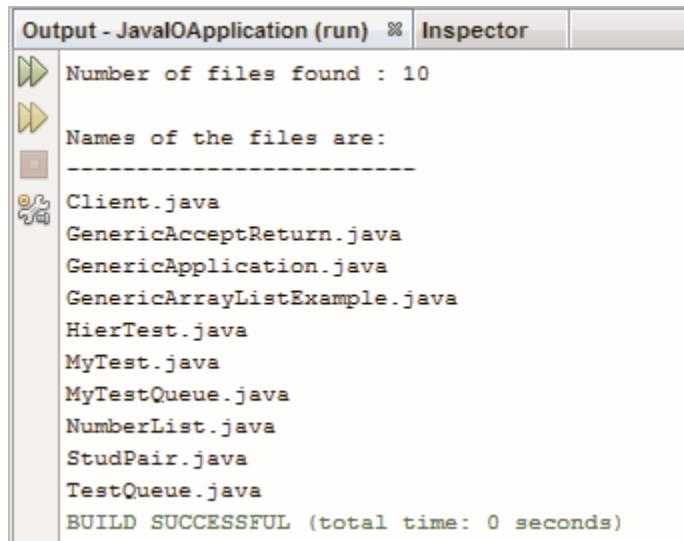
Code Snippet 6:

```
import java.io.*;
class FileFilter implements FilenameFilter {
String ext;
public FileFilter(String ext) {
this.ext = "." + ext;
}
public boolean accept (File dir, String fName) {
return fName.endsWith(ext);
}
public class DirList {
public static void main (String [] args) {
String dirName = "d:/resources";
File fileObj = new File ("d:/resources");
FilenameFilter filterObj = new FileFilter("java");
String[] fileName = fileObj.list(filterObj);
System.out.println("Number of files found : " + fileName.length);
System.out.println("");
System.out.println("Names of the files are : ");
System.out.println("      ");
for(int ctr=0; ctr<fileName.length; ctr++) {
System.out.println(fileName[ctr]);
}
}
}
```

The `FilenameFilter` interface defines an `accept()` method which is used to check if the specified file should be included in a file list. This interface is implemented by the `FileFilter` class.

The `accept()` method is implemented in this class and returns true if the filename ends with `.java` extension as stored in the variable `ext`. The `list()` method restricts the visibility of the file and displays only those files which ends with the specified extension.

Figure 3.2 displays the output.



The screenshot shows the 'Output - JavaIOApplication (run)' tab in the Android Studio interface. It displays the following text:

```
Number of files found : 10
Names of the files are:
-----
Client.java
GenericAcceptReturn.java
GenericApplication.java
GenericArrayListExample.java
HierTest.java
MyTest.java
MyTestQueue.java
NumberList.java
StudPair.java
TestQueue.java
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.2: FilenameFilter Class - Output

3.5 FileDescriptor Class

`FileDescriptor` class provides access to the file descriptors that are maintained by the operating system when files and directories are being accessed. In practical use, a file descriptor is used to create a `FileInputStream` or `FileOutputStream` to contain it. File descriptors should not be created on their own by applications as they are tied to the operating system.

The `FileDescriptor` class has following public fields:

**static final
FileDescriptor err**

The static `FileDescriptor err` acts as a handle to the standard error stream.

**static final
FileDescriptor in**

The static `FileDescriptor in` acts as a handle to the standard input stream.

**static final
FileDescriptor out**

The static `FileDescriptor out` acts as a handle to the standard output stream.

3.5.1 Methods of `FileDescriptor`

The methods of the `FileDescriptor` class are as follows:

→ **`sync()`**

The `sync()` method clears the system buffers and writes the content that they contain to the actual hardware.

```
public void sync() throws SyncFailedException
```

→ **`valid()`**

The `valid()` method checks whether the file descriptor is valid. Since the file descriptors are associated with open files, they become invalid when the file is closed.

3.6 DataInput and DataOutput Interfaces

Data stream supports input/output of primitive data types and string values. The data streams implement DataInput or DataOutput interface.

DataInput interface has methods for:

Reading bytes from a binary stream and convert the data to any of the Java primitive types.

Converting data from Java modified Unicode Transmission Format (UTF)-8 format into string form. UTF-8 is a special format for encoding 16 bit Unicode values. Note: UTF-8 assumes that in most cases, the upper eight bits of a Unicode will be zero and optimizes accordingly.

DataOutput interface has methods for:

Converting data present in Java primitive type into a series of bytes and write them onto a binary stream.

Converting string data into Java-modified UTF-8 format and write it into a stream.

3.6.1 Methods of DataInput Interface

DataInput interface has several methods to read inputs, such as, binary data from the input stream and reconstructs data from the bytes to any of the Java primitive type form. An IOException will be raised if the methods cannot read any byte from the stream or if the input stream is closed.

Some methods in the DataInput interface are listed in Table 3.2.

Method	Description and Syntax
readBoolean()	The <code>readBoolean()</code> method reads an input byte from a stream and returns true if the byte is not zero and false otherwise. <code>boolean readBoolean() throws IOException</code>
readByte()	The <code>readByte()</code> method reads one byte from a stream which is a signed value in the range from -128 to 127. <code>byte readByte() throws IOException</code>
readInt()	The <code>readInt()</code> method reads four bytes from a stream and returns the int value of the bytes read. <code>int readInt() throws IOException</code>
readDouble()	The <code>readDouble()</code> method reads eight bytes from a stream and returns a double value of the bytes read. <code>double readDouble() throws IOException</code>

Method	Description and Syntax
readLine()	The <code>readLine()</code> method reads a line of text from the input stream. It reads a byte at a time and then, converts the byte into a character and goes on reading until it encounters the end of line or end of file. The characters are then returned as a <code>String</code> . <code>String readLine() throws IOException</code>

Table 3.2: Methods of DataInput Interface

3.6.2 Methods in the DataOutput Interface

`DataOutput` interface has several methods to write outputs, such as, binary data to the output stream. An `IOException` may be thrown if bytes cannot be written to the stream.

The important methods in this interface are listed in Table 3.3.

Method	Description and Syntax
<code>writeBoolean(boolean b)</code>	The <code>writeBoolean(boolean b)</code> method writes the boolean value given as parameter to an output stream. If the argument has the value as true, then 1 will be written, otherwise, the value 0 is written. <code>public void writeBoolean(boolean b) throws IOException</code>
<code>writeByte(int value)</code>	The <code>writeByte(int value)</code> method writes the byte value of the integer given as parameter to an output stream. <code>public void writeByte(int value) throws IOException</code>
<code>writeInt(int value)</code>	The <code>writeInt(int value)</code> method writes four bytes that represent the integer given as parameter to an output stream. <code>public void write(int value) throws IOException</code>
<code>writeDouble(double value)</code>	The <code>writeDouble(double value)</code> method writes eight bytes that represent the double value given as parameter to an output stream. <code>public void writeDouble(double value) throws IOException</code>

Table 3.3: Methods of DataOutput Interface

Code Snippet 7 displays the working of byte streams using the `FileInputStream` class and `FileOutputStream` class.

Code Snippet 7:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class ByteStreamApp {
public static void main(String[] args) throws IOException {
FileInputStream inObj = null;
FileOutputStream outObj = null;
try {
inObj = new FileInputStream("c:/java/hello.txt");
outObj = new FileOutputStream("outagain.txt");
int ch;
while ((ch = inObj.read()) != -1) {
outObj.write(ch);
}
} finally {
if (inObj != null) {
inObj.close();
}
if (outObj != null) {
outObj.close();
}
}
}
}
```

In Code Snippet 7, `read()` method reads a character and returns an int value. This allows the `read()` method to indicate that the end of the stream is reached by returning a value of -1. Using this approach, the entire contents of **Hello.txt** are read and then, copied into **outagain.txt**. Note that the path for **outagain.txt** has not been specified. This means that it will be created in the current directory, which is, the directory in which this **.java** file exists.

When a stream is no longer required, it is important to close the stream, as shown here in the `finally` block. This helps to avoid resource leaks.

The Java platform uses Unicode conventions to store character values. Character stream I/O translates this format to and from the local character set. Note that in Western locales, the local character set is typically an 8-bit superset of ASCII. For most applications, input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams adapts to the local character set and is ready for internationalization. All character stream classes are derived from the `Reader` and `Writer` classes. There are character stream classes that specialize in file I/O operations such as `FileReader` and `FileWriter`.

Code Snippet 8 displays the reading and writing of character streams using the `FileReader` and `FileWriter` class.

Code Snippet 8:

```
import java.io.FileReader;
```

```

import java.io.FileWriter;
import java.io.IOException;
public class CharStreamApp {
    public static void main(String[] args) throws IOException {
        FileReader inObjStream = null;
        FileWriter outObjStream = null;
        try {
            inObjStream = new FileReader("c:/java/hello.txt");
            outObjStream = new FileWriter("charoutputagain.txt");
            int ch;
            while ((ch = inObjStream.read()) != -1) {
                outObjStream.write(ch);
            }
            outObjStream.close();
        } finally {
            if (inObjStream != null) {
                inObjStream.close();
            }
        }
    }
}

```

Character streams act as wrappers for byte streams. The character stream manages translation between characters and bytes and uses the byte stream to perform the physical I/O operations. For example, `FileWriter` uses `FileOutputStream` class to write the data.

When there are no required prepackaged character stream classes, byte-to-character bridge streams, `InputStreamReader` and `OutputStreamWriter`, are used to create character streams.

3.7 *InputStream Class and Its Subclasses*

Figure 3.3 shows the `InputStream` class hierarchy.

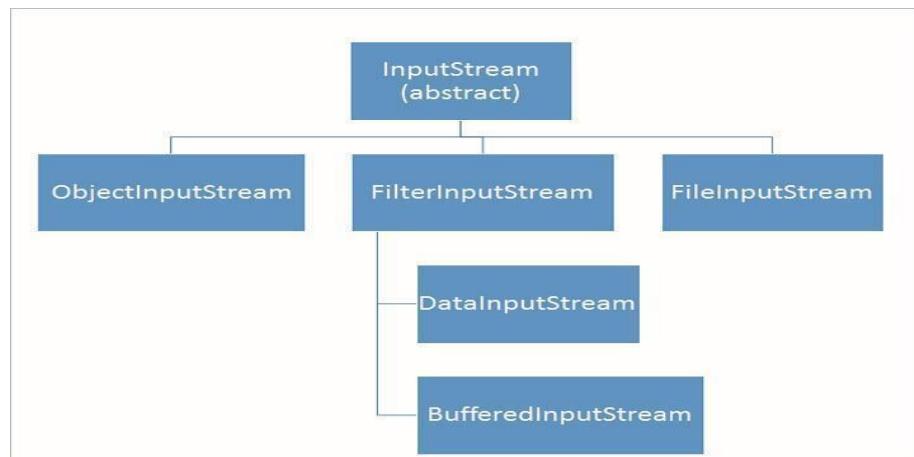


Figure 3.3: InputStream Class Hierarchy

3.7.1 Methods of InputStream Class

The `InputStream` class provides a number of methods that manage reading data from a stream. Some of the methods of this class are as follows in Table 3.4.

Method	Description and Syntax
<code>read()</code>	The <code>read()</code> method reads the next bytes of data from the input stream and returns an <code>int</code> value in the range of 0 to 255. The method returns -1 when end of file is reached. <code>public abstract int read() throws IOException</code>
<code>available()</code>	The <code>available()</code> method returns the number of bytes that can be read without blocking. In other words, it returns the number of available bytes. <code>public int available() throws IOException</code>
<code>close()</code>	The <code>close()</code> method closes the input stream. It releases the system resources associated with the stream. <code>public void close() throws IOException</code>
<code>mark(int n)</code>	The <code>mark(int n)</code> method marks the current position in the stream and will remain valid until the number of bytes specified in the variable, <code>n</code> , is read. A call to the <code>reset()</code> method will position the pointer to the last marked position. <code>public void mark(int readlimit)</code>
<code>skip(long n)</code>	The <code>skip(long n)</code> method skips <code>n</code> bytes of data while reading from an input stream. <code>public long skip(long n) throws IOException</code>
<code>reset()</code>	The <code>reset()</code> method rests the reading pointer to the previously set mark in the stream. <code>public void reset() throws IOException</code>

Table 3.4: Methods of InputStream Class

3.7.2 FileInputStream Class

File stream objects can be created by either passing the name of the file, or a `File` object or a `FileDescriptor` object respectively. `FileInputStream` class is used to read bytes from a file. When an object of `FileInputStream` class is created, it is also opened for reading. `FileInputStream` class overrides all the methods of the `InputStream` class except `mark()` and `reset()` methods. The `reset()` method will generate an `IOException`.

Code Snippet 9 displays the creation of `FileInputStream` object.

Code Snippet 9:

```
...
FileInputStream fileName = new FileInputStream("Helloworld.txt");
File fName = new File("/command.doc");
FileInputStream fileObj = new FileInputStream(fName);
```

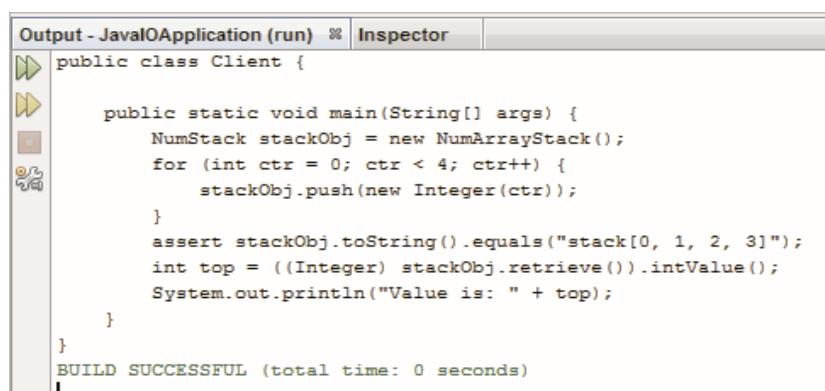
Code Snippet 10 demonstrates how to create a `FileInputStream` object using different constructors.

The code in Code Snippet 10 creates a `FileInputStream` object to which the filename is passed as an argument. The object is used to read the text characters from the specified file. The program prints out source code from a file named **Client.java** (which it is assumed to be existing on the computer).

Code Snippet 10:

```
import java.io.FileInputStream;
import java.io.IOException;
public class FIStream {
public static void main(String argv[]) {
try {
FileInputStream intest;
intest = new FileInputStream("D:/resources/Client.java");
int ch;
while ((ch = intest.read()) > -1) {
StringBuffer buf = new StringBuffer();
buf.append((char) ch);
System.out.print(buf.toString());
}
} catch (IOException e) {
System.out.println(e.getMessage());
}
}
}
```

Figure 3.4 displays the output. The contents of **Client.java** are printed out on the console.



The screenshot shows the Java IDE's Output window titled "Output - JavaIOApplication (run)". The window displays the source code of the Client class. The code defines a public class Client with a main method. Inside the main method, a NumStack object is created and populated with integers 0, 1, 2, and 3. The stack's toString method is asserted to return "stack[0, 1, 2, 3]". Then, the top integer is retrieved and printed. The output window also shows the message "BUILD SUCCESSFUL (total time: 0 seconds)" at the bottom.

```
Output - JavaIOApplication (run)  Inspector
public class Client {

    public static void main(String[] args) {
        NumStack stackObj = new NumArrayStack();
        for (int ctr = 0; ctr < 4; ctr++) {
            stackObj.push(new Integer(ctr));
        }
        assert stackObj.toString().equals("stack[0, 1, 2, 3]");
        int top = ((Integer) stackObj.retrieve()).intValue();
        System.out.println("Value is: " + top);
    }
}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.4: `FileInputStream` Class - Output

3.7.3 ByteArrayInputStream Class

`ByteArrayInputStream` contains a buffer that stores the bytes that are read from the stream. `ByteArrayInputStream` class uses a byte array as the source. `ByteArrayInputStream` class has an internal counter, which keeps track of the next byte to be read. This class does not support any new methods. It only overrides the methods of the `InputStream` class such as `read()`, `skip()`, `available()`, and `reset()`.

- **protected byte[] buf**
This refers to an array of bytes that is provided by the creator of the stream.
- **protected int count**
This refers to the index greater than the last valid character in the input stream buffer.
- **protected int mark**
This refers to the currently marked position in the stream.
- **protected int pos**
This refers to the index of the next character to be read from the input stream buffer.

Code Snippet 11 displays the use of the `ByteArrayInputStream` class.

Code Snippet 11:

```
import java.io.ByteArrayInputStream;
public class newdemo {
public static void main(String argv[]){
String content = "You may freely edit this file.";
byte [] bObj = content.getBytes();
ByteArrayInputStream inputByte = new ByteArrayInputStream(bObj);
int contentVal;
for(int y=0 ; y<1; y++) {
while(( contentVal = inputByte.read()) != -1) {
System.out.print(Character.toUpperCase((char)contentVal));
}
}
}
}
```

The code reads the content of the string variable, `content`, into a byte array and then, constructs a `ByteArrayInputStream` instance from the byte array. Then, using a combination of a `for` and `while` loop, the content of the `ByteArrayInputStream` instance is displayed on the console.

3.8 OutputStream Class and its Subclasses

The `OutputStream` class is an abstract class that defines the method in which bytes or arrays of bytes are written to streams. `ByteArrayOutputStream` and `FileOutputStream` are the subclasses of `OutputStream` class.

3.8.1 Methods in OutputStream Class

There are several methods in `OutputStream` class, which are used for writing bytes of data to a stream. All the methods of this class throw an `IOException`.

Some of the methods of this class are as follows:

→ **write(int b)**

The `write(int b)` method writes the specified byte of the integer given as parameter to an output stream.

```
public abstract void write(int b) throws IOException
```

→ **write(byte[] b)**

The `write(byte[] b)` method writes a byte array given as parameter to an output stream. The number of bytes will be equal to the length of the byte array.

```
public void write(byte[] b) throws IOException
```

→ **write(byte[] b, int off, int len)**

The `write(byte[] b, int off, int len)` method writes bytes from a byte array given as parameter starting from the given offset, `off`, to an output stream. The number of bytes written to the stream will be equal to the value specified in `len`.

```
public void write(byte[] b, int off, int len) throws IOException
```

→ **flush()**

The `flush()` method flushes the stream. The buffered data is written to the output stream. Flushing forces the buffered output to be written to the intended destination. It ensures that only those bytes that are buffered are passed to the operating system for writing. There is no guarantee that the bytes will be actually written to the physical device.

```
public void flush() throws IOException
```

→ **close()**

The `close()` method closes the output stream. The method will release any resource associated with the output stream.

```
public void close() throws IOException
```

3.8.2 FileOutputStream Class

`FileOutputStream` class creates an `OutputStream` that is used to write bytes to a file. `FileOutputStream` may or may not create the file before opening it for output and it depends on the underlying platform.

Certain platforms allow only one file-writing object to open a file for writing. Therefore, if the file is already open, the constructors in the class fail. An `IOException` will be thrown only when a read-only file is opened.

Code Snippet 12 displays the use of `FileOutputStream` class.

Code Snippet 12:

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class newdemo {
    public static void main(String argv[]) throws FileNotFoundException,
    IOException {
        String temp = "One way to get the most out of life is to look upon it as an
        adventure.";
        byte [] bufObj = temp.getBytes();
        FileOutputStream fileObj = new FileOutputStream("Thought.txt");
        fileObj.write(bufObj);
        fileObj.close();
    }
}
```

Code Snippet 12 first stores the content of the `String` variable in the byte array, `bufObj`, using the `getBytes()` method. Then, the entire content of the byte array is written to the file, `Thought.txt`. The exceptions must either be caught or else declared in the `main` method as shown here.

3.8.3 `ByteArrayOutputStream` Class

`ByteArrayOutputStream` class creates an output stream in which the data is written using a byte array. It allows the output array to grow in size so as to accommodate the new data that is written.

3.8.4 Methods in `ByteArrayOutputStream` Class

The `ByteArrayOutputStream` class inherits all the methods of the `OutputStream` class. The methods of this class allow retrieving or converting data. Methods of this class can be invoked even after the output stream has been closed and will not generate `IOException`.

Some of the methods in `ByteArrayOutputStream` class are as follows:

→ **reset()**

The `reset()` method erases all the bytes that have been written so far by setting the value to 0.

```
public void reset()
```

→ **size()**

The `size()` method returns the number of bytes written to the buffer.

```
public int size()
```

→ **toByteArray()**

The `toByteArray()` method creates a newly allocated byte array containing the bytes

that have been written to this stream so far.

```
public byte[] toByteArray()
```

→ **writeTo(OutputStream out)**

The `writeTo(OutputStream out)` method writes all the bytes that have been written to this stream from the internal buffer to the specified output stream argument.

→ **toString()**

The `toString()` method converts the content of the byte array into a string. The method converts the bytes to characters according to the default character encoding of the platform.

```
public String toString()
```

Code Snippet 13 displays the use of the `ByteArrayOutputStream` class

Code Snippet 13:

```
String strObj = "Hello World";
byte[] buf = strObj.getBytes();
ByteArrayOutputStream byObj = new ByteArrayOutputStream();
byObj.write(buf);
System.out.println("The string is: " + byObj.toString());
...
```

In Code Snippet 13, a `ByteArrayOutputStream` object is created and then, the content from the byte array is written to the `ByteArrayOutputStream` object. Finally, the content from the output stream is converted to a string using the `toString()` method and displayed.

3.9 Filter Streams

The `FilterInputStream` class provides additional functionality by using an input stream as its basic source of data. The `FilterOutputStream` class streams are over existing output streams. They either transform the data along the way or provide additional functionality.

3.9.1 `FilterInputStream` Class

It can also transform the data along the way. The class overrides all the methods of the `InputStream` class that pass all requests to the contained input stream. The subclasses can also override certain methods and can provide additional methods and fields.

Following are the fields and constructors for `java.io.FilterInputStream` class:

→ **protected InputStream in:** This input stream must be filtered.

→ **protected FilterInputStream(InputStream in):** This creates a `FilterInputStream`. The argument is assigned to the field `in` and can be recalled anytime.

Following are the methods of this class:

- **mark(int readlimit) :**
This method identifies the current position in the input stream.
- **markSupported() :**
This method checks if the input stream supports the `mark` and `reset` methods.
- **read() :**
This method reads the next byte of data from the input stream.
- **available() :**
This method returns an approximation of bytes that can be read or skipped from the input stream.
- **close() :**
This method closes the input stream and releases any system resources related with the stream.
- **read(byte[] b) :**
This method reads `b.length` bytes of data from the input stream into an array of bytes.
- **reset() :**
This method repositions the pointer to the position in the stream when the `mark` method was last invoked on the input stream.
- **skip(long n) :**
This method skips and discards `n` bytes of data from the input stream.
- **read(byte[] b, int off, int len) :**
This method reads `len` bytes of data from the input stream into an array of bytes.

Code Snippet 14 demonstrates the use of `FilterInputStream` class.

Code Snippet 14:

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FilterInputStream;
import java.io.IOException;
import java.io.InputStream;
public class FilterInputStreamApplication {
    public static void main(String[] args) throws Exception {
        InputStream inputObj = null;
        FilterInputStream filterInputObj = null;
        try {
            // creates input stream objects
```

```
inputObj = new FileInputStream("C:/Java/Hello.txt");
filterInputObj = new BufferedInputStream(inputObj);
// reads and prints from filter input stream
System.out.println((char) filterInputObj.read());
System.out.println((char) filterInputObj.read());
// invokes mark at this position filterInputObj.mark(0);
System.out.println("mark() invoked");
System.out.println((char) filterInputObj.read());
System.out.println((char) filterInputObj.read());
} catch (IOException e) {
// prints if any I/O error occurs
e.printStackTrace();
} finally {
// releases system resources associated with the stream
if (inputObj != null) {
inputObj.close();
}
if (filterInputObj != null) {
filterInputObj.close();
}
}
}
```

Figure 3.5 displays the output.

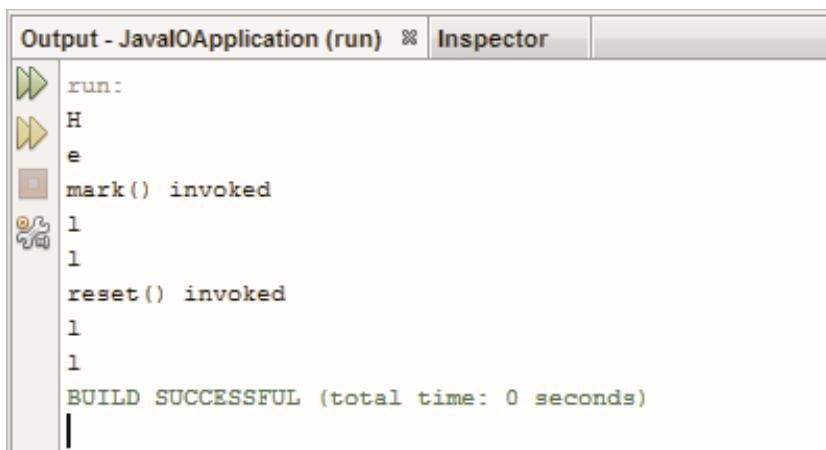


Figure 3.5: FilterInputStream Class – Output

3.9.2 FilterOutputStream Class

The `FilterOutputStream` class overrides all methods of `OutputStream` class that pass all requests to the underlying output stream. Subclasses of `FilterOutputStream` can also override certain methods and give additional methods and fields.

The `java.io.FilterOutputStream` class includes the protected `OutputStream out` field, which is the output stream to be filtered. `FilterOutputStream(OutputStream out)` is the constructor of this class. This creates an output stream filter that exist class over the defined output stream.

Code Snippet 15 demonstrates the use of `FilterOutputStream` class.

Code Snippet 15:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;
public class FilterOutputStreamApplication {
    public static void main(String[] args) throws Exception {
        OutputStream OutputStreamObj = null;
        FilterOutputStream filterOutputStreamObj = null;
        FileInputStream filterInputStreamObj = null;
        byte[] bufObj = {81, 82, 83, 84, 85};
        int i=0;
        char c;
        //encloses the creation of stream objects within try-catch block
        try{
            // creates output stream objects
            OutputStreamObj = new FileOutputStream("C:/Java/test.txt");
            filterOutputStreamObj = new FilterOutputStream(OutputStreamObj);
            // writes to the output stream from bufObj
            filterOutputStreamObj.write(bufObj);
            // forces the byte contents to be written to the stream
            filterOutputStreamObj.flush();
            //creates an input stream object
            filterInputStreamObj = new FileInputStream("C:/Java/test.txt");
            while((i=filterInputStreamObj.read())!=-1) {
                // converts integer to character
                c = (char)i;
                // prints the character read
                System.out.println("Character read after conversion is: "+ c);
            }
        }catch(IOException e){
            // checks for any I/O errors
            System.out.print("Close() is invoked prior to write()");
        }
        finally{
            // releases system resources associated with the stream
            if(OutputStreamObj!=null)
                OutputStreamObj.close();
            if(filterOutputStreamObj!=null)
                filterOutputStreamObj.close();
        }
    }
}
```

Figure 3.6 displays the output.

```

Output - JavaIOApplication (run)  ✘ Inspector
run:
Character read after conversion is: Q
Character read after conversion is: R
Character read after conversion is: S
Character read after conversion is: T
Character read after conversion is: U
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 3.6: FilterOutputStream Class - Output

3.10 Buffered Streams

A buffer is a temporary storage area for data. By storing the data in a buffer, time is saved as data is immediately received from the buffer instead of going back to the original source of the data.

Java uses buffered input and output to temporarily cache data read from or written to a stream. This helps programs to read or write small amounts of data without adversely affecting the performance of the system. Buffer allows skipping, marking, and resetting of the stream.

Filters operate on the buffer, which is located between the program and the destination of the buffered stream. Figure 3.7 displays the Object class hierarchy.

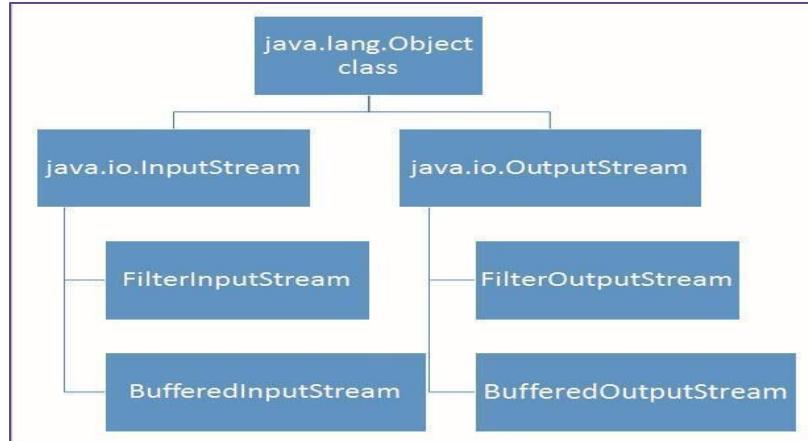


Figure 3.7: java.lang.Object Class Hierarchy

3.10.1 BufferedInputStream Class

BufferedInputStream class allows the programmer to wrap any InputStream class into a buffered stream. BufferedInputStream acts as a cache for inputs. It does so by creating the array of bytes which are utilized for future reading. The simplest way to read data from an instance of BufferedInputStream class is to invoke its `read()` method. BufferedInputStream class also supports the `mark()` and `reset()` methods. The function `markSupported()` will return true if it is supported.

Some of the methods of this class are listed in Table 3.5.

Method	Description
<code>int available()</code>	The method returns the number of bytes of input that is available for reading.
<code>void mark(int num)</code>	The method places a mark at the current position in the input stream.
<code>int read() *</code>	The method reads data from the underlying input stream. It raises an <code>IOException</code> if an I/O error takes place.
<code>int read(byte [] b, int off, int length)</code>	The method reads bytes into the specified byte array from the given offset. It raises <code>IOException</code> if an I/O error takes place.
<code>void reset()</code>	The method repositions the pointer in the stream to the point where the <code>mark</code> method was last called.

Table 3.5: Methods of BufferedInputStream Class

3.10.2 *BufferedOutputStream Class*

`BufferedOutputStream` creates a buffer which is used for an output stream. It provides the same performance gain that is provided by the `BufferedInputStream` class. The main concept remains the same, that is, instead of going every time to the operating system to write a byte, it is cached in a buffer. It is the same as `OutputStream` except that the `flush()` method ensures that the data in the buffer is written to the actual physical output device. `flush()` method of the `BufferedOutputStream` class method flushes the buffered output stream.

3.11 Character Streams

Byte stream classes provide methods to handle any type of I/O operations except Unicode characters. Character streams provide functionalities to handle character-oriented input/output operations. They support Unicode characters and can be internationalized.

`Reader` and `Writer` are abstract classes at the top of the class hierarchy that supports reading and writing of Unicode character streams. All character stream class are derived from the `Reader` and `Writer` class. Figure 3.8 displays the character stream classes.

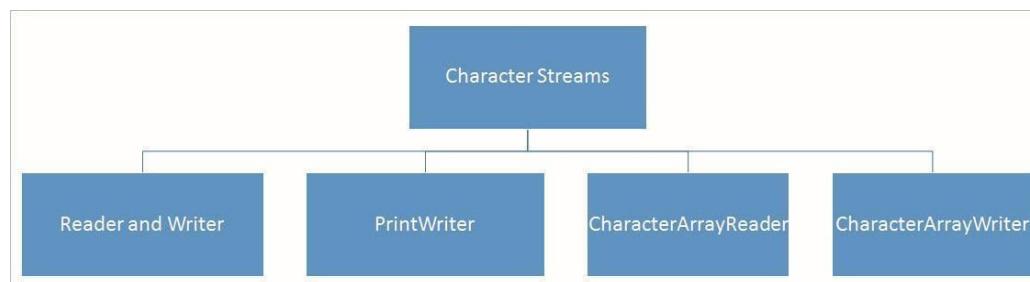


Figure 3.8: Character Streams

3.11.1 Reader Classes

Reader class is an abstract class used for reading character streams. The subclasses of this class override some of the methods present in this class to increase the efficiency and functionality of the methods. All the methods of this class throw an `IOException`. The `read()` method returns -1 when end of the file is encountered.

Some of the methods of the Reader class are listed in Table 3.6.

Method	Description
<code>int read() throws IOException</code>	The method reads a single character
<code>int read(char[] buffer, int offset, int count) throws IOException</code>	The method reads character into a section of an array and returns the number of characters that are read.
<code>int read(char[] buffer) throws IOException</code>	The method reads characters into an array and returns the number of characters read.
<code>long skip(long count) throws IOException</code>	The method skips a certain number of characters as specified by count.
<code>boolean ready() throws IOException</code>	The method returns true if the reader is ready to be read from.
<code>void close() throws IOException</code>	The method closes the input stream and releases the system resources.
<code>boolean markSupported()</code>	The method informs whether the stream supports the <code>mark()</code> operation.
<code>void reset()</code>	The method resets the stream.

Table 3.6: Methods of Reader Class

3.11.2 Writer Classes

Writer class is an abstract class and supports writing characters into streams through methods that can be overridden by its subclasses. The methods of the `java.io.Writer` class are same as the methods of the `java.io.OutputStream` class. All the methods of this class throw an `IOException` in case of errors.

Some of the methods of this class are listed in Table 3.7.

Method	Description
<code>void write(int c) throws IOException</code>	The method reads a single character.
<code>void write(char[] text) throws IOException</code>	The method writes a complete array of characters to an output stream.
<code>void write(char[] text, int offset, int length)</code>	The method writes a portion of an array of characters to an output stream starting from offset. The variable length specifies the number of characters to write.

Method	Description
<code>void write(String s) throws IOException</code>	The method writes a string to the output stream.
<code>void flush() throws IOException</code>	The method flushes the output stream so that buffers are cleared.
<code>void close() throws IOException</code>	The method closes the output stream.

Table 3.7: Methods of the Writer Class

3.11.3 PrintWriter Class

The `PrintWriter` class is a character-based class that is useful for console output. It implements all the print methods of the `PrintStream` class. It does not have methods for writing raw bytes. In such a case, a program uses unencoded byte streams. The `PrintWriter` class differs from the `PrintStream` class as it can handle multiple bytes and other character sets properly. This class provides support for Unicode characters.

The class overrides the `write()` method of the `Writer` class with the difference that none of them raise any `IOException`. The printed output is tested for errors using the `checkError()` method.

The `PrintWriter` class also provides support for printing primitive data types, character arrays, strings, and objects. It provides formatted output through its `print()` and `println()` methods. The `toString()` methods will enable the printing of values of objects.

The main advantage of the `print()` and `println()` methods is that any Java object or literal or variable can be printed by passing it as an argument. If the `autoFlush` option is set to true, then automatic flushing takes place when `println()` method is invoked. The `println()` method follows its argument with a platform dependent line separator. The `print()` method does not flush the stream automatically. Otherwise, both these methods are same.

Some of the overloaded methods of this class are listed in Table 3.8.

Method	Description
<code>boolean checkError()</code>	The method flushes the stream if it is open and check the error state.
<code>void print(boolean b)</code>	The method prints a boolean value.
<code>void print(char c)</code>	The method is used to print a character.
<code>void print(int i)</code>	The method is used to print an integer.
<code>void flush() throws IOException</code>	The method flushes the output stream so that buffers are cleared.
<code>void close() throws IOException</code>	The method closes the output stream.
<code>void write(char[] buf)</code>	The method is used for writing an array of characters.

Table 3.8: Methods of PrintWriter Class

The `PrintWriter` class implements and overrides the abstract `write()` method from the `Writer` class. The only difference is that none of the `write()` methods of the `PrintWriter` class throws an `IOException` because it is caught inside the class and an error flag is set.

Code Snippet 16 displays the use of the `PrintWriter` class.

Code Snippet 16:

```
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
public class PrintWriterDemo {
    public static void main(String[] args) throws Exception {
        InputStreamReader reader = new InputStreamReader (System.in);
        OutputStreamWriter writer = new OutputStreamWriter (System.out);
        PrintWriter pwObj = new PrintWriter (writer,true);
        int tmp = 0;
        char ch;
        try {
            while (tmp != -1) {
                tmp = reader.read ();
                ch = (char) tmp;
                pwObj.println ("echo " + ch);
            }
        }
        catch (IOException e) {
            System.out.println ("IOerror:" + e);
        }
    }
}
```

An instance of the `PrintWriter` class is created. The `println()` method is used to display the value accepted from the user.

3.11.4 *CharArrayReader Class*

`CharArrayReader` class is a subclass of `Reader` class. The class uses a character array as the source of text to be read. `CharArrayReader` class has two constructors and reads stream of characters from an array of characters.

Some of the methods of this class are listed in Table 3.9.

Method	Description
<code>long skip(long n)</code>	The method skips n number of characters before reading.
<code>void mark(int num)</code>	The method marks the current position in the stream.
<code>int read()</code>	The method reads a single character.
<code>void close()</code>	The method closes the input stream.

Table 3.9: Methods of `CharArrayReader` Class

Code Snippet 17 displays use of the CharArrayReader class.

Code Snippet 17:

```
...
String temp = "Hello World";
int size = temp.length();
char [] ch = new char[size];
temp.getChars(0, size, ch, 0);
CharArrayReader readObj = new CharArrayReader(ch, 0, 5);
```

3.11.5 CharArrayWriter Class

CharArrayWriter class is a subclass of Writer class. CharArrayWriter uses a character array into which characters are written. The size of the array expands as required. The methods `toCharArray()`, `toString()`, and `writeTo()` method can be used to retrieve the data. CharArrayWriter class inherits the methods provided by the Writer class.

Some of the methods of this class are listed in Table 3.10.

Method	Description
<code>void close()</code>	The method closes the stream.
<code>void flush()</code>	The method flushes stream.
<code>void write()</code>	The method writes a single character to the array.
<code>void write(char[] b, int off, int length)</code>	The method writes characters to the buffer.
<code>void reset()</code>	The method repositions the pointer in the buffer to the point where the mark method was last called or to the beginning of the buffer if it has not been marked.
<code>int size()</code>	The method returns current size of the buffer.
<code>char[] toCharArray()</code>	The method returns a copy of the data.
<code>String toString()</code>	The method is used to convert the input data to string.
<code>void write(String str, int off, int len)</code>	The method writes a portion of string to buffer.
<code>void writeTo(Writer out)</code>	The method is used to write the contents of buffer to a character stream.

Table 3.10: Methods of CharArrayWriter Class

Code Snippet 18 displays use of the CharArrayWriter class.

Code Snippet 18:

```
...
CharArrayWriter fObj = new CharArrayWriter();
...
String temp = "Hello World";
int size = temp.length();
char [] ch = new char[size];
```

```

temp.getChars(0, temp.length(), ch, 0);
fObj.write(ch);
char[] buffer = fObj.toCharArray();
System.out.println(buffer);
System.out.println(fObj.toString());

```

In Code Snippet 18, the content of the entire string is stored as a series of characters in the character array, **ch**, by using the **getChars()** method. Next, the instance of **CharArrayWriter** contains the content from the character array. The **toCharArray()** method is used to store the content of the **CharArrayWriter** in a character array. Finally, the content is printed.

Code Snippet 19 demonstrates the use of **CharArrayWriter** class.

Code Snippet 19:

```

import java.io.CharArrayWriter;
import java.io.IOException;
public class CharArrayWriterDemo {
public static void main(String[] args) throws IOException {

// Create a CharArrayWriter object which can hold 11 characters.
CharArrayWriter writer = new CharArrayWriter(11);

String str ="Hello Aptech";
writer.write("Hello Aptech", 6, str.length() - 6);
System.out.println("The CharArrayWriter buffer contains: " +
writer.toString());
writer.flush();

// Print out the contents of the CharArrayWriter buffer.
System.out.println("After flushing the CharArrayWriter, buffer " +
"contains:" + writer.toCharArray());
// Now reset the buffer we just populated.
writer.reset();
// Print out the contents of the CharArrayWriter buffer.
System.out.println("After reset, CharArrayWriter buffer " + "contains:
" + writer.toCharArray());
// Close the CharArrayWriter and StringWriter buffers.
writer.close();
}
}

```

The code initializes a string variable and copies a portion of the string to an instance of **CharArrayWriter** class and displays the content of the **CharArrayWriter** object.

3.12 *Serialization*

Persistence is the process of saving data to some permanent storage. A persistent object can be stored on disk or sent to some other machine for saving its data. On the other hand, a non-persistent

object exists as long as the JVM is running, Java is an object-oriented language and thus provides the facilities for reading and writing object. Serialization is the process of reading and writing objects to a byte stream.

An object that implements the `Serializable` interface will have its state saved and restored using serialization and deserialization facilities. When a Java object's class or superclass implements the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`, the Java object becomes serializable. The `java.io.Serializable` interface defines no methods. It indicates that the class should be considered for serialization.

If a superclass is serializable then, its subclasses are also serializable. The only exception is if a variable is transient and static, its state cannot be saved by serialization facilities. When the serialized form of an object is converted back into a copy of the object, this process is called deserialization.

When an object is serialized, the class file is not recorded. However, information that identifies its class is recorded in the serialized stream. The system that deserializes the object specifies how to locate and load the necessary class files. For example, a Java application can load the class definitions by using information stored in the directory.

Serialization is required to implement the Remote Method Invocation (RMI) where a Java object on one machine can invoke the method of another object present on another machine. In this remote method calling, the source machine may serialize the object and transmit whereas the receiving machine will deserialize the object.

If the underlying service provider supports, a serializable object can be stored in the directory.

An exception, `NotSerializableException`, is thrown when a field has an object reference that has not implemented `java.io.Serializable`. Fields marked with the keyword `transient` should not be serialized. Values stored in static fields are not serialized. When an object is deserialized the values in the static fields are set to the values declared in the class and the values in non-static transient fields are set to their default values.

A version number `serialVersionUID` is associated with a serializable class when it does not explicitly declare the version number. The version number is calculated based on various aspects of the class. A serializable class can declare its own version number by declaring a field named `serialVersionUID` of type `static, long, and final`.

3.12.1 ObjectOutputStream Class

`ObjectOutputStream` class extends the `OutputStream` class and implements the `ObjectOutput` interface. It writes primitive data types and object to the output stream.

3.12.2 Methods in ObjectOutputStream Class

The methods in `ObjectOutputStream` class helps to write objects to the output stream.

The methods in `ObjectOutputStream` class are as follows:

<code>writeFloat(float f)</code>	<code>writeObject(Object obj)</code>	<code>defaultWriteObject()</code>
<p>The <code>writeFloat(float f)</code> method writes a float value to the output stream. Its signature is as follows:</p> <pre>public void writeFloat(float f) throws IOException</pre>	<p>The <code>writeObject(Object obj)</code> method writes an object, <code>obj</code>, to the output stream. Its signature is as follows:</p> <pre>public final void writeObject(Object obj) throws IOException</pre>	<p>The <code>defaultWriteObject()</code> method writes non-static and non-transient fields into the underlying output stream. Its signature is as follows:</p> <pre>public void defaultWriteObject() throws IOException</pre>

Code Snippet 20 displays the use of methods of `ObjectOutputStream` class. Assume that necessary import statements are added.

Code Snippet 20:

```
public class Demo {  
    public static class Person implements Serializable {  
        public String name = null;  
        public int age = 0;  
    }  
    public static void main(String[] args) throws IOException,  
    ClassNotFoundException {  
        ObjectOutputStream objectOutputStream =  
            new ObjectOutputStream(new FileOutputStream  
                ("customer.dat"));  
        Person person = new Person();  
        person.name = "Michael Smith";  
        person.age = 44;  
        objectOutputStream.writeObject(person);  
        objectOutputStream.close();  
        System.out.println("Done");  
    }  
}
```

In Code Snippet 20, an object of `FileOutputStream` is chained to an object output stream. Then, to the object's output stream, the `writeObject()` method is invoked with an object as its argument. If you open `customer.dat`, you will find some garbled text in there along with "Michael Smith". This is because the data has been serialized.

The `ObjectOutputStream` class serializes an object into a stream to perform following actions:

1. Open one of the output streams, for example `FileOutputStream`
2. Chain it with the `ObjectOutputStream`
3. Call the method `writeObject()` providing the instance of a `Serializable` object as an argument
4. Close the streams

3.12.3 ObjectInputStream Class

`ObjectInputStream` class extends the `InputStream` class and implements the `ObjectInput` interface. `ObjectInput` interface extends the `DataInput` interface and has methods that support object serialization. `ObjectInputStream` is responsible for reading object instances and primitive types from an underlying input stream. It has `readObject()` method to restore an object containing non-static and non-transient fields.

Code Snippet 21 displays the creation of an instance of `ObjectInputStream` class. This builds on the code created in Code Snippet 20.

Code Snippet 21:

```
...
FileInputStream fObj = new FileInputStream("customer.dat");
ObjectInputStream ois = new ObjectInputStream(fObj);
Person obj = (Person) ois.readObject();
System.out.println(obj.name);
ois.close();
```

In Code Snippet 21, an instance of `FileInputStream` is created that refers to the file named **customer.dat**.

An `ObjectInputStream` instance is created from that file stream. The `readObject()` method returns an object, which deserializes the instance. Finally, the object input stream is closed.

The `ObjectInputStream` class deserializes an object. The object to be deserialized must be already created using the `ObjectOutputStream` class. Following steps have been followed in the program:

1. Opens an input stream. Chains it with the `ObjectInputStream`.
2. Calls the method `readObject()` and cast the returned object to the class that is being serialized.
3. Closes the streams.

Code Snippet 22 demonstrates use of `Serializable` interface and the stream classes.

Code Snippet 22:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
class Employee implements Serializable{
String lastName;
String firstName;
double sal;
}
```

```

public class BranchEmpProcessor {
public static void main(String[] args) {
FileInputStream fIn = null;
FileOutputStream fOut = null;
ObjectInputStream oIn = null;
ObjectOutputStream oOut = null;
try {
fOut = new FileOutputStream("E:\\NewEmp.Ser");
oOut = new ObjectOutputStream(fOut);
Employee e = new Employee();
e.lastName = "Smith";
e.firstName = "John";
e.sal = 5000.00;
oOut.writeObject(e);
oOut.close();
fOut.close();
fIn = new FileInputStream("E:\\NewEmp.Ser");
oIn = new ObjectInputStream(fIn);
//de-serializing employee
Employee emp = (Employee) oIn.readObject();
System.out.println("Deserialized - " + emp.firstName + " " +
emp.lastName + " from NewEmployee.ser");
} catch (IOException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
e.printStackTrace();
} finally {
System.out.println("finally");
}
}
}

```

3.13 Console Class

Java provides the `Console` class to enhance and simplify the development of command line applications. The `Console` class is a part of `java.io` package and has the ability to read text from the terminal without echoing it on the screen. The `Console` object provides input and output of character streams through its `Reader` and `Writer` classes.

The `Console` class provides various methods to access character-based console device. There are no public constructors for the `Console` class. To obtain an instance of the `Console` class, you required to invoke the `System.console()` method. The `System.console()` method returns the `Console` object if it is available; otherwise, it returns null. At present, the methods of `Console` class can be invoked only from the command line and not from Integrated Development Environments (IDEs), such as Eclipse, NetBeans, and so on.

The `Console` class provides methods to perform input and output operations on character streams. The `readLine()` method reads a single line of text from console. The `readPassword()` method

reads password from the console without echoing on the screen. The method returns a character array and not a `String` object to enable modification of the password. The password is removed from the memory once it is no longer required.

Table 3.11 lists various methods available in the `Console` class.

Method	Description
<code>format(String fmt, Object... args)</code>	The method displays formatted data to the console's output
<code>printf(String fmt, Object... args)</code>	The method displays formatted data to the console's output more conveniently
<code>reader()</code>	The method returns a unique <code>java.io.Reader</code> object that is associated with the console
<code>readLine()</code>	The method accepts one line of text from the console
<code>readLine(String fmt, Object... args)</code>	The method provides formatted output and accepts one line of text from the console

Table 3.11: Methods of Console Class

Code Snippet 23 shows the use of `Console` class methods.

Code Snippet 23:

```
import java.io.Console;
import java.io.IOException;
public class ConsoleDemo {
public static void main(String [] args) {
Console cons = System.console();
if (cons == null) {
System.err.println("No console device is present!");
return;
}
try {
String username = cons.readLine("Enter your username: ");
char [] pwd = cons.readPassword("Enter your secret Password: ");
System.out.println("Username = " + username);
System.out.println("Password entered was = " + new String(pwd));
}
catch(IOException ioe) {
cons.printf("I/O problem: %s\n", ioe.getMessage());
}
}
}
```

The code is designed to accept username and password from the user through a console using the `readLine()` and `readPassword()` methods. The `System.console()` method returns a

Console object (if it is available) that reads the username and password. If you run this code in NetBeans IDE, `System.console()` will return null. However, if you run the code through command prompt with `javac` and `java` commands, it accepts the username and password.

3.14 How to Read/Write Strings to and from Files?

`FileWriter` class is used to write data into text files of Java. To read data from text files, use `FileReader`. Both of these are Character stream classes. `FileInputStream` and `FileOutputStream` should not be used to read/write textual data since they are Byte stream classes which will not be apt for reading/ writing to and from the files.

→ **FileWriter**

`FileWriter` helps to generate a file and write characters in it. This class inherits from the `OutputStream` class. The `FileWriter` class accepts default character encoding and the default byte-buffer size. Upon a `FileOutputStream`, user can construct an `OutputStreamWriter` to specify the values. `FileWriter` is utilized to write character streams. `FileOutputStream` is used to write raw bytes.

Table 3.12 lists the methods of `FileWriter`.

Method	Description
<code>public void write (int c) throws IOException{}</code>	Utilized to write a single character.
<code>public void write (char [] stir) throws IOException{}</code>	Utilized to write characters array.
<code>public void write (String str) throws IOException{}</code>	Utilized to write the data of the string which is usually a combination of letters, numbers, and special characters
<code>public void write (String str, int off, int len) throws IOException{}</code>	Utilized to write a string portion. <code>off</code> – creates an Offset (from to start writing characters) <code>len</code> – indicates the number of characters present.
<code>public void flush() throws IOException{ }</code>	Used to flush the stream.
<code>public void close() throws IOException{ }</code>	Used to close the writer after flushing it.

Table 3.12: FileWriter Methods

`FileWriter` writes one character at a time and also the reader reads every single character resulting in the rise of the I/O processes. Thus, it affects the working of the system. Using `BufferedWriter` with `FileWriter` increases execution speed.

→ **FileReader**

`FileReader` reads each character from a text file.

- `FileReader` is derived from the `InputStreamReader` class.
- `FileReader` class assumes that the default character encoding and the byte-buffer size are appropriate. You can mention the values by constructing an `InputStreamReader` over a `FileInputStream`.
- `FileReader` is used to read character streams, whereas `FileInputStream` reads raw bytes data streams.

Table 3.13 lists methods of `FileReader`.

Method	Description
<code>public int read () throws IOException</code>	Use this method to read a single character. This method blocks until, <ul style="list-style-type: none"> ○ a character is available ○ any I/O error occurs ○ it moves towards the stream end
<code>public int read(char[] cbuff) throws IOException</code>	This method reads characters of an array and blocks until, <ul style="list-style-type: none"> ○ Input is available ○ Any I/O error occurs ○ It finds the stream end
<code>public abstract int read(char[] buff, int off, int len) throws IOException</code>	Use this method for reading characters in a part of an array. This method blocks until, <ul style="list-style-type: none"> ○ Input is available ○ Any I/O error occurs ○ It nears the stream end

Table 3.13: FileReader Methods

Table 3.14 lists the parameters of `FileReaders` and descriptions.

Parameter	Description
<code>cbuf</code>	Buffer for Destination
<code>off</code>	Offset at which one can start to store characters
<code>len</code>	Maximum number of characters to read

Table 3.14: Parameters

Exceptions:

`public void close() throws IOException`: Closes the reader.

`public long skip (long n) throws IOException`: Skips characters and blocks until,

- A character is available
- Any I/O error occurs
- Stream end is approached

Parameter: `n` specifies the number of characters to skip

3.15 Unsafe Operations on Foreign Memory

`Unsafe` is a class in the Java standard library that provides direct access to memory and enables developers to perform unsafe operations that bypass the built-in safety checks and type safety of the Java language. This class is generally intended for use by the Java core libraries, JVM implementors, and advanced developers who understand the risks associated with using it.

Some common unsafe operations performed using the `Unsafe` class include:

1. Direct Memory Access: Using `Unsafe`, you can allocate, read, and write data directly to and from native memory without going through Java's object model.
2. Off-heap memory management: `Unsafe` allows to allocate and deallocate memory outside the Java heap, which can be useful for certain performance-critical applications.
3. Pointer manipulation: With `Unsafe`, you can manipulate memory addresses as pointers, which is not possible using the standard Java language.
4. Object serialization: You can use `Unsafe` to directly read and write fields of an object, bypassing the standard Java serialization mechanisms.

3.16 Foreign Functions

In Java, foreign functions refer to functions that are implemented in languages other than Java, typically in native languages such as C or C++. These foreign functions are often called using a mechanism known as Java Native Interface (JNI).

JNI is a Java programming framework that allows Java code to interact with native code (code written in other languages). It provides a way for Java applications to call functions written in native code libraries and vice versa. The primary use cases for using foreign functions in Java are as follows:

Accessing platform-specific features:
Sometimes, certain platform-specific features or hardware functionalities may not be available directly through Java's standard libraries. In such cases, developers can write the functionality in a native language and then, use JNI to access and utilize those functions in the Java application.

Performance optimization: Java, being a managed and platform-independent language, might introduce some overhead due to garbage collection and other runtime checks. In performance-critical scenarios, developers can write performance-sensitive parts of their code in a native language to achieve better performance.

Here is a simple example of using a foreign function in Java with JNI shown in Code Snippets 24a and b respectively.

Code Snippet 24a: Java code

```
public class ForeignFunctionExample {  
    static {  
        System.loadLibrary("myLibrary"); // Load the native  
                                         //library  
    }  
  
    // Declare the native method  
    public native void callNativeFunction();
```

```
public static void main(String[] args) {
    ForeignFunctionExample example = new ForeignFunctionExample();
    example.callNativeFunction(); // Call the native function
}
```

Code Snippet 24b: C/C++ code (myLibrary.c)

```
#include <jni.h>

JNIEXPORT void JNICALL
Java_ForeignFunctionExample_callNativeFunction(JNIEnv *env, jobject
obj) {
    // Implement the functionality in C/C++
    // ...
}
```

3.17 Summary

- A stream is a logical entity that produces or consumes information.
- Data stream supports input/output of primitive data types and String values.
- InputStream is an abstract class that defines how data is received.
- The OutputStream class defines the way in which output is written to streams.
- File class directly works with files on the file system.
- A buffer is a temporary storage area for data.
- Serialization is the process of reading and writing objects to a byte stream.
- Java provides the Console class to enhance and simplify command line applications.
- The Console class provides various methods to access character-based console device.

3.18 Check Your Progress

1. Which of the following are true about stream classes?
- a. They help read input from a stream.
 - b. Input and Output streams are used for reading and writing of structured sequence of bytes.
 - c. They help manage disk files.
 - d. All classes that work on streams directly works with files and the file system.

(A)	a, c	(C)	c
(B)	b	(D)	d

2. The `File(String dirpath)` constructor _____.
- a. Creates a `File` object with pathname of the file specified by the `String` variable `dirpath` into an abstract pathname.
 - b. Creates a `File` object with pathname of the file specified by the `String` variables `parent` and `child`.
 - c. Creates a new `File` instance from another `File` object specified by the variable `fileObj`.
 - d. Converts the given file URI into an abstract pathname and creates a new `File` instance.

(A)	a	(C)	c
(B)	b	(D)	d

3. Which of the following classes create a `FileInputStream` or `FileOutputStream` to contain it?
- a. `System`
 - b. `File`
 - c. `FileDescriptor`
 - d. `FilenameFilter`

(A)	a	(C)	c
(B)	b	(D)	d

4. The `sync` method _____.
- a. Creates an (invalid) `FileDescriptor` object
 - b. Checks whether the file descriptor is valid
 - c. Reads bytes from a binary stream
 - d. Clears the system buffers and writes the content that they contain to the actual hardware

(A)	a	(C)	c
(B)	b	(D)	d

5. What is the process of saving data to some permanent storage called?

- a. Serialization
- b. Persistence
- c. Deserialization
- d. Append mode

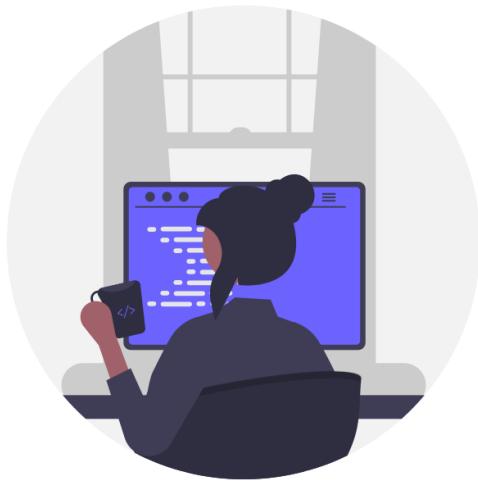
(A)	a	(C)	c
(B)	b	(D)	d

3.18.1 Answers

1.	A
2.	A
3.	C
4.	D
5.	B

Try It Yourself

1. Create a Java program that reads a text file and counts the number of words in it. Display the total word count on the console.
2. Write a Java program that reads a CSV file, parses its data, and displays it in a structured format.
3. Create a Java application that reads a list of student objects from a file, employs the Stream API to sort students based on their GPA in descending order, and then, generates a report highlighting the top-performing students.
4. Design a Java program that reads a list of products from a file, utilizes the Stream API to categorize products by price range (for example, low, medium, high), and then, generates a user-friendly interface on the console to browse and select products from each category.



Session 4

Threading

Welcome to the Session, **Threading**.

This session introduces you to the concept of threads in Java and explains how to create a `Thread` object. It further explains different states of the `Thread` object and methods of the `Thread` class. Finally, the session provides an overview of daemon threads.

In this Session, you will learn to:

- Explain the `Thread` class
- Elaborate on the process of creating threads
- Identify `Thread` states
- List methods of `Thread` class
- Explain managing threads
- Describe daemon threads
- Explain Scoped values in JDK 20
- Describe how to create Virtual Threads
- Explain the difference between Virtual threads and OS threads

4.1 *Introduction*

A process is a program that is executed. Each process has its own runtime resources, such as their own data, variables, and memory space. These runtime resources constitute an execution environment for the processes inside a program. So, every process has a self-contained execution environment to run independently. Each process executes several tasks at a time and each task is carried out by separate thread. A thread is nothing but the basic unit to which the operating system allocates processor time. A thread is the entity within a process that can be scheduled for execution.

A process is started with a single thread, often called the primary, default or main thread. So, a process in turn contains one or more threads.

A thread has following characteristics:

- A thread has its own complete set of basic runtime resources to run it independently.
- A thread is the smallest unit of executable code in an application that performs a particular job or task.
- Several threads can be executed at a time, facilitating execution of several tasks of a single application simultaneously.

4.1.1 Comparing Processes and Threads

Multiple threads allow access to the same part of memory whereas, processes cannot directly access memory of another process.

Some of the similarities and differences of processes and threads are as follows:

Similarities	Differences
<p>Threads share a central processing unit and only one thread is active (running) at a time.</p> <p>Threads within processes execute sequentially.</p> <p>A thread can create child threads or sub threads.</p> <p>If one thread is blocked, another thread can run.</p>	<p>Unlike processes, threads are not independent of one another.</p> <p>Unlike processes, all threads can access every address in the task.</p> <p>Unlike processes, threads are designed to assist one other.</p>

4.1.2 Application and Uses of Threads

In Java, a Thread facilitates parallel processing of multiple tasks. Some of the applications of threads are as follows:

- i. Playing sound and displaying images simultaneously.
- ii. Displaying multiple images on the screen.
- iii. Displaying scrolling text patterns or images on the screen.

4.2 Creating Threads

An easy way of creating a new thread is to derive a class from `java.lang.Thread` class. This class consists of methods and constructors, which helps in realizing concurrent programming concepts in Java. This is used to create applications which can execute multiple tasks at a given point of time.

The step-by-step procedure to create a new thread by extending the `Thread` class is discussed here.

1) Step 1: Creating a Subclass

Declare a class that is a subclass of the `Thread` class defined in the `java.lang` package.

Code Snippet 1 shows the creation of the subclass. `MyThread` will inherit the capabilities of `Thread` class and allow us to perform threading tasks.

Code Snippet 1:

```
class MyThread extends Thread {  
//Extending Thread class  
//class definition  
...  
}
```

2) Step 2: Overriding the run() method

Inside the subclass, override the `run()` method defined in the `Thread` class. The code in the `run()` method defines the functionality required for the thread to execute. The `run()` method in a thread is analogous to the `main()` method in an application.

Code Snippet 2 displays the implementation of the `run()` method.

Code Snippet 2:

```
class MyThread extends Thread {  
//Extending Thread class  
  
// class definition  
public void run() //overriding the run() method  
{  
// implementation  
}  
...  
}
```

3) Step 3: Starting the Thread

The `main()` method creates an object of the class that extends the `Thread` class. Next, the `start()` method is invoked on the object to start the Thread. The `start()` method will place the `Thread` object in a runnable state. The `start()` method of a thread invokes the `run()` method which allocates the resources required to run the thread.

Code Snippet 3 displays the implementation of the `start()` method. It makes use of the `MyThread` class defined in Code Snippet 2.

Code Snippet 3:

```
...  
public class TestThread {  
    public static void main(String args[]) {  
        MyThread t=new MyThread(); //creating Thread object  
        t.start(); //Starting the thread  
    }  
}
```

Since we have not included any code in the `run()` method (in Code Snippet 2) as of now, Code Snippet 3 when executed will not display anything.

4.2.1 Constructors and Methods of Thread Class

Constructors of the Thread class are listed in Table 4.1. The ThreadGroup class represents a group of threads and is often used in constructors of the Thread class.

Constructor	Description
Thread()	Default constructor
Thread(Runnable objRun)	Creates a new Thread object, where objRun is the object whose run() method is called
Thread(Runnable objRun, String threadName)	Creates a new named Thread object, where objRun is the object whose run() method is called and threadName is the name of the thread that will be created
Thread(String threadName)	Creates a new Thread object where threadName is the name of the thread that will be created
Thread(ThreadGroup group, Runnable objRun)	Creates a new Thread object, where group is the thread group and objRun is the object whose run() method is called
Thread(ThreadGroup group, Runnable objRun, String threadName)	Creates a new Thread object so that it has objRun as its run object, has the specified threadName as its name, and belongs to ThreadGroup referred to by group
Thread(ThreadGroup group, Runnable objRun, String threadName, long stackSize)	Creates a new Thread object so that it has objRun as its run object, has the specified threadName as its name, belongs to the thread group referred to by group, and has the specified stack size
Thread(ThreadGroup group, String threadName)	Creates a new Thread object with group as the thread group and threadName as the name of the thread that will be created

Table 4.1: Constructors of Thread Class

The Thread class provides several methods to work with threaded programs. Some methods of the Thread class are listed in Table 4.2.

Method	Description
static int activeCount()	Returns the number of active threads among the current threads in the program
static Thread currentThread()	Returns a reference to the currently executing thread object
ThreadGroup getThreadGroup()	Returns the thread group to which this thread belongs
static boolean interrupted()	Tests whether the current thread has been interrupted
boolean isAlive()	Tests if this thread is alive
boolean isInterrupted()	Tests whether this thread has been interrupted
void join()	Waits for this thread to die

Method	Description
void setName(String name)	Changes the name of this thread to be equal to the argument name

Table 4.2: Methods of the Thread Class

Code Snippet 4 demonstrates the creation of a new thread by extending the Thread class and using some of the methods of Thread class.

Code Snippet 4:

```
/*
Creating threads using Thread class and using methods of the class
*/
package demo;
/**
NamedThread is created as a subclass of the class Thread
*/
public class NamedThread extends Thread {
    String name;
    /**
     * This method of Thread class is overridden to specify the action
     * that will be done when the thread begins execution
    */
    public void run() {
        //Will store the number of threads
        int count = 0;
        while(count<=3) {
            //Display the number of threads
            System.out.println(Thread.activeCount());
            //Display the name of the currently running thread
            name = Thread.currentThread().getName();
            count++;
            System.out.println(name);
            if (name.equals ("Thread1"))
                System.out.println("Marimba");
            else
                System.out.println("Jini");
        }
    }
    public static void main(String args[]) {
        NamedThread objNamedThread = new NamedThread();
        objNamedThread.setName("Thread1");

        //Display the status of the thread, whether alive or not
        System.out.println(Thread.currentThread().isAlive());
        System.out.println(objNamedThread.isAlive());
        /*invokes the start method which in turn will call
         * run and begin thread execution
        */
        objNamedThread.start();
    }
}
```

```
System.out.println(Thread.currentThread().isAlive());  
System.out.println(objNamedThread.isAlive());  
}  
}
```

In this example, **NamedThread** is declared as a derived class of **Thread**. In the **main()** method of this class, a thread object **objNamedThread** is created by instantiating **NamedThread** and its name is set to **Thread1**. The code then checks to see if the current thread is alive by invoking the **isAlive()** method and displays the return value of the method. This will result in **true** being printed because the main (default) thread has begun execution and is currently alive.

The code also checks if **objNamedThread** is alive by invoking the same method on it. However, at this point of time, **objNamedThread** has not yet begun execution so the output will be **false**.

Next, the **start()** method is invoked on **objNamedThread** which will cause the thread to invoke the **run()** method which has been overridden. The **run()** method prints the total number of threads running, which are by now, 2. The method then checks the name of the thread running and prints **Marimba** if the currently running thread's name is **Thread1**. The method performs this checking three times. Thus, the final output of the code will be:

```
true  
true  
Thread1  
Marimba  
Thread1  
Marimba  
Thread1  
Marimba  
Thread1  
Marimba
```

Note: Concurrent Programming

Concurrent programming is a process of running several tasks at a time. In Java, it is possible to execute simultaneously an invoked method and the statements following the method call, without waiting for the invoked method to terminate. The invoked method runs independently and concurrently with the invoking program, and can share variables, data, and so on with it.

4.2.2 Runnable Interface

The **Runnable** interface is designed to provide a common set of rules for objects that wish to execute a code while a thread is active.

Another way of creating a new thread is by implementing the **Runnable** interface. This approach can be used because Java does not allow multiple class inheritance. Therefore, depending upon the required and requirement, either of the approaches can be used to create Java threads. The step-by-step procedure for creating and running a new **Thread** by implementing the **Runnable** interface has been discussed here.

1) Step 1: Implementing the Runnable interface

Declare a class that implements the Runnable interface. Code Snippet 5 implements the Runnable interface.

Code Snippet 5:

```
// Declaring a class that implements Runnable interface  
class MyRunnable implements Runnable {  
    ...  
}
```

This code is still incomplete so it will not compile.

2) Step 2: Implementing the run() method

The Runnable interface defines a method, `run()`, to contain the code that will be executed by the thread object. The class implementing the Runnable interface should override the `run()` method. Code Snippet 6 implements the `run()` method.

Code Snippet 6:

```
// Declaring a class that implements Runnable interface  
class MyRunnable implements Runnable {  
    public void run() { // Overriding run()  
        ... // implementation  
    }  
}
```

3) Step 3: Starting the Thread

In the `main()` method, create an object of the class that implements the Runnable interface. Next, pass this object to the constructor of a `Thread` class to create an object of `Thread` class. Finally, invoke the `start()` method on the thread object to start the thread.

Code Snippet 7 implements the `start()` method. **MyRunnable** was created in Code Snippet 6.

Code Snippet 7:

```
class ThreadTest{  
public static void main(String args[]) {  
    Runnable r = new MyRunnable();  
    Thread thObj=new Thread(r);  
    thObj.start(); //Starting a thread  
}
```

Code Snippet 8 demonstrates a complete example of how a thread can be created using the interface, Runnable.

Code Snippet 8:

```
/*
*Creating threads using Thread
*class and using methods of the class
*/
package test;
/**
 * NamedThread is created to implement the interface Runnable
 */
class NamedThread implements Runnable {
/* this will store name of the thread */
String name;
/**
 * This method of Runnable is implemented to specify the action
 * that will be done when the thread begins execution.
 */
public void run() {
int count = 0; //will store the number of threads
while(count < 3) {
name = Thread.currentThread().getName();
System.out.println(name);
count++;
}
}
}

public class MainTest {
public static void main(String args[]) {
NamedThread objNewThread= new NamedThread();
Thread objThread = new Thread(objNewThread);
objThread.start();
}
}
```

In this example, the **NamedThread** class implements Runnable interface. The **NamedThread** class implements Runnable, therefore its instance can be passed as an argument to the constructor of Thread class. The output will be:

```
Thread-0
Thread-0
Thread-0
```

4.3 Thread States

A thread can exist in several states, such as new, runnable, blocked, waiting, and terminated, according to its various phases in a program. When a thread is newly created, it is a **new** Thread and is not alive.

In this state, it is an empty Thread object with no system resources allocated. So, the thread is left in a 'new thread' state till the `start()` method is invoked on it. When a thread is in this state, you can only start the thread or stop it. Calling any other method before starting a thread raises an `IllegalThreadStateException`.

Figure 4.1 displays the **new** state of a thread.

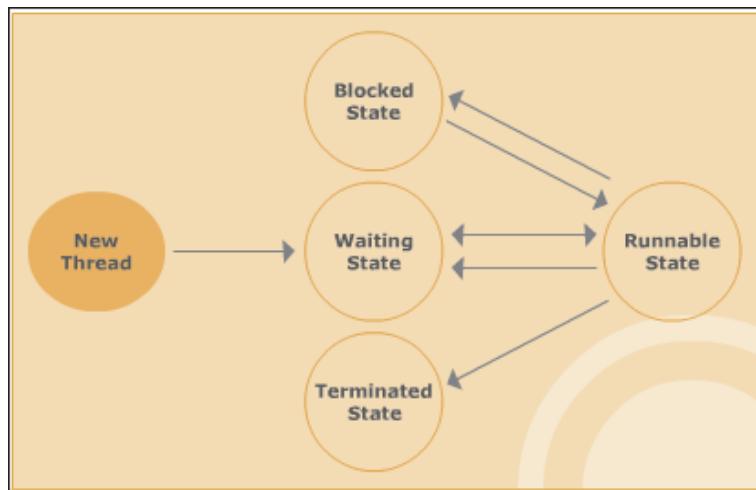


Figure 4.1: New State

Code Snippet 9 displays the creation of a Thread object.

Code Snippet 9:

```
...  
Thread thObj = new Thread();  
...
```

An instance of `Thread` class named `thObj` is created and will be in a **new** state.

4.3.1 Runnable State

A new thread can be in a runnable state when the `start()` method is invoked on it. A thread in this state is alive. A thread can enter this state from running or blocked state.

Threads are prioritized because in a single processor system all runnable threads cannot be executed at a time. In the runnable state, a thread is eligible to run, but may not be running as it depends on the priority of the thread. The runnable thread, when it becomes eligible for running, executes the instructions in its `run()` method. Figure 4.2 displays the runnable state.

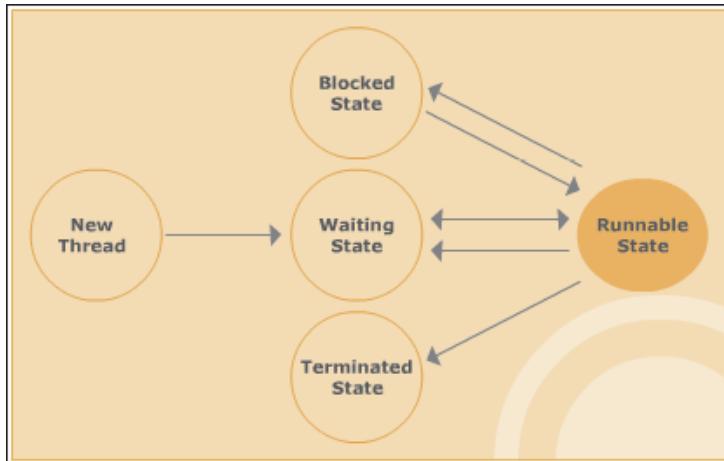


Figure 4.2: Runnable State

Code Snippet 10 demonstrates the runnable state of a thread.

Code Snippet 10:

```

...
MyThreadClass myThread = new MyThreadClass () ;
myThread.start () ;
...
  
```

An instance of thread is created and is in a runnable state.

Note: This state is called runnable because although the thread might not be running, the thread is allocated all the resources to run.

Scheduler is a component in Java that assigns priority to the threads so that their execution is inline with the requirements.

4.3.2 Blocked State

Blocked state is one of the states in which a thread:

- Is alive but currently not eligible to run as it is blocked for some other operation
- Is not runnable but can go back to the runnable state after getting the monitor or lock

A thread in the blocked state waits to operate on the resource or object which at the same time is being processed by another thread. A running thread goes to blocked state when `sleep()`, `wait()` or `suspend()` method is invoked on it. Figure 4.3 displays the blocked state.

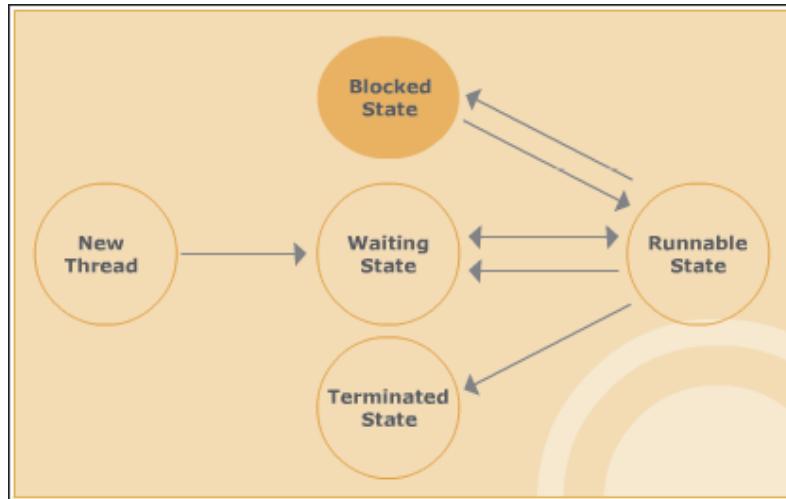


Figure 4.3: Blocked State

Note: Lock or monitor is an imaginary box containing an object. This imaginary box will only allow a single thread to enter it so that it can operate on the object. Any thread which wants to share the resource or object must get the lock.

4.3.3 Waiting State

A thread is in this state when it is waiting for another thread to release resources for it. When two or more threads run concurrently and only one thread takes hold of the resources all the time, other threads ultimately wait for this thread to release the resource for them. In this state, a thread is alive but not running.

A call to the `wait()` method puts a thread in this state. Invoking the `notify()` or `notifyAll()` method brings the thread from the waiting state to the runnable state. Figure 4.4 displays the waiting state of a thread.

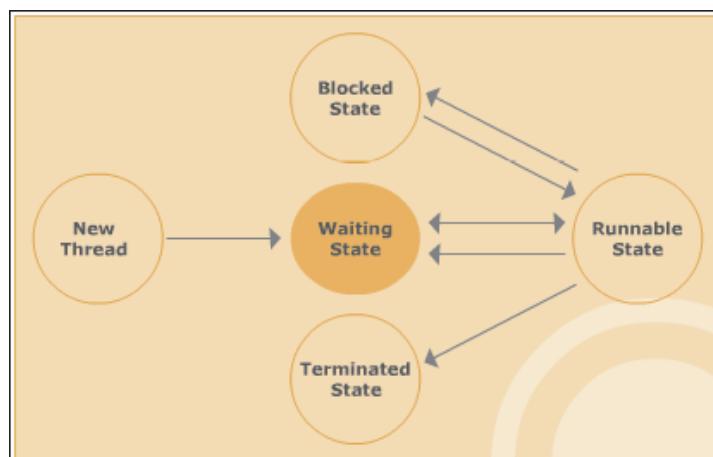


Figure 4.4: Waiting State

4.3.4 Terminated State

A thread, after executing its `run()` method dies and is said to be in a terminated state. This is the way a thread can be stopped naturally. Once a thread is terminated, it cannot be brought back to runnable state. Methods such as `stop()` and `destroy()` can force a thread to be terminated, but JDK 1.5 onwards, these methods are deprecated.

Figure 4.5 displays the terminated state of a thread.

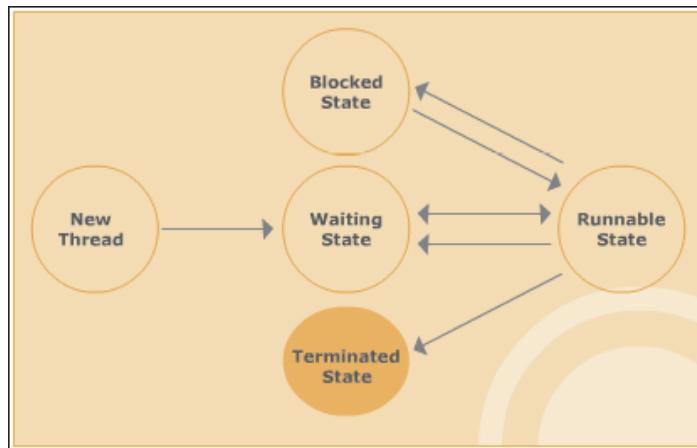


Figure 4.5: Terminated State

Note: If `start()` method is invoked on a terminated state it will throw a runtime exception.

4.4 Methods of Thread Class

The `Thread` class consists of several methods that can be used to manipulate threads.

All threads have a name associated with it. At times, it is required to retrieve the name of a particular thread. The `getName()` method helps to retrieve the name of the current thread.

Code Snippet 11 demonstrates the use of the `getName()` method.

Code Snippet 11:

```
public void run() {  
    for(int i = 0; i < 5; i++) {  
        Thread t = Thread.currentThread();  
        System.out.println("Name = " + t.getName());  
        ...  
    }  
}
```

The code in Code Snippet 11 demonstrates the use of `getName()` method to obtain the name of the thread object.

Note: The `setName()` method assigns a name to a `Thread` instance. The name is passed as an argument to the method.

```
public final void setName(String name)  
where, name is a String argument passed to the setName() method.
```

4.4.1 *start ()* Method

A newly created thread remains idle until the `start ()` method is invoked. The `start ()` method allocates the system resources necessary to run the thread and executes the `run ()` method of its target object.

At the time of calling the `start ()` method, following actions happen:

- A new thread execution starts
- The thread moves from the new thread to runnable state Figure 4.6 displays the use of the `start ()` method in the code.

```
public static void main(String args[]) {  
    nameRunnable r=new nameRunnable();  
    Thread one=new Thread(r);  
    one.setName("James"); // setName() method named the thread as James.  
    Thread two=new Thread(r);  
    two.setName("Rita"); // setName() method named the thread as Rita.  
    Thread three=new Thread(r);  
    three.setName("Jack"); // setName() method named the thread as Jack.  
    one.start(); // start() method changes the thread state to runnable state.  
    two.start(); // start() method changes the thread state to runnable state.  
    three.start(); // start() method changes the thread state to runnable state.  
}
```

Figure 4.6: *start()* Method

Code Snippet 12 demonstrates the use of `start ()` method.

Code Snippet 12:

```
...  
NewThread thObj = new NewThread();  
thObj.start();  
...
```

The thread object is in a runnable state after `start ()` is called.

Note: Do not start a thread more than once. A thread may not be restarted once it has completed execution.

4.4.2 *run ()* Method

The life of a thread starts when the `run ()` method is invoked. The characteristics of the `run ()` method are as follows:

- 1) It is public
- 2) Accepts no argument
- 3) Does not return any value
- 4) Does not throw any exceptions

The `run ()` method contains instructions, which are executed once the `start ()` method is invoked. Figure 4.7 displays the use of `run ()` method in the code.

The screenshot shows a Java code editor with a file named 'nameRunnable.java'. The code implements the Runnable interface. The 'run()' method is highlighted with a red box. The code inside the run() method prints the name of the currently running thread and sleeps for 1000 milliseconds.

```
public class nameRunnable implements Runnable{
    /**
     * run() method allows the a thread to go to running state.
     */
    public void run(){
        for(int x=1; x<4; x++) {
            System.out.println("nameRunnable is running by" +
                /** getName() method give the name of the
                 * currently running thread.*/
                Thread.currentThread().getName());
            try{
                Thread.sleep(1000);
                /** sleep() method will force the running thread to
                 * wait for 1000 milli seconds.*/
            }catch (InterruptedException ex) { }
        }
    }
}
```

Figure 4.7: run() Method

Syntax

```
public void run()
```

Code Snippet 13 demonstrates the use of `run()` method.

Code Snippet 13:

```
class MyRunnable implements Runnable {
    ...
    public void run() {
        System.out.println("Inside the run method.");
    }
    ...
}
```

4.4.3 `sleep()` Method

The `sleep()` method has following characteristics:

- 1) It suspends the execution of the current thread for a specified period.
- 2) It makes the processor time available to the other threads of an application or other applications that might be running on the computer system.
- 3) It stops the execution if the active thread for the time specified in milliseconds or nanoseconds.
- 4) It raises `InterruptedException` when it is interrupted using the `interrupt()` method.

Syntax

```
void sleep(long millis)
```

Figure 4.8 displays the use of the `sleep()` method in the code.

```

public class nameRunnable implements Runnable{
    /**
     * run() method allows the a thread to go to running state.
     */
    public void run()
    {
        for(int x=1; x<4; x++) {
            System.out.println("nameRunnable is running by" +
                /** getName() method give the name of the
                 * currently running thread.*/
                Thread.currentThread().getName());
            try{
                Thread.sleep(1000);
                /** sleep() method will force the running thread to
                 * wait for 1000 milli seconds.*/
            }catch (InterruptedException ex) { }
        }
    }
}

```

Figure 4.8: sleep() Method

Code Snippet 14 demonstrates the use of the `sleep()` method.

Code Snippet 14:

```

try{
myThread.sleep (10000);
}
catch (InterruptedException e)
{ }

```

Code Snippet 14 demonstrates the use of `sleep()` method. The thread has been put to sleep for 10000 milliseconds.

4.4.4 *interrupt () Method*

The `interrupt()` method interrupts the thread. The method tells the thread to stop what it was doing even before it has completed the task. The `interrupt()` method has following characteristics:

- 1) An interrupted thread can die, wait for another task, or go to the next step depending on the requirement of the application.
- 2) It does not interrupt or stop a running thread; rather it throws an `InterruptedException` if the thread is blocked, so that it exits the blocked state.
- 3) If the thread is blocked by `wait()`, `join()`, or `sleep()` methods, it receives an `InterruptedException`, thus terminating the blocking method prematurely.

Syntax

```
public void interrupt()
```

Note: A thread can interrupt itself. When a thread is not interrupting itself, the `checkAccess()` method gets invoked. This method inspects whether the current thread has enough permission to modify the thread. In case of insufficient permissions, the security manager throws a `SecurityException` to indicate a security violation.

4.5 Managing Threads

In our day-to-day activities, different degrees of importance must be assigned to different tasks. The decision is taken so that the best possible outcome is achieved. For example, while deciding between the activities going to work and watching television, the activity going to work gets priority over watching television.

This is because going to work is more important for the financial benefit of the person than mere entertainment. Similarly, in Java programming, it is necessary to prioritize the threads according to their importance.

Threads are self-executing entities inside a program. In a single program, several threads can execute independently of each other. However, at times, it may happen that a particular runtime resource must be shared by many threads, running simultaneously. This forces other running threads to enter the blocked state. So, in such situations some internal control or management of the threads is required so that the threads are executed simultaneously. Figure 4.9 displays multiple threads.

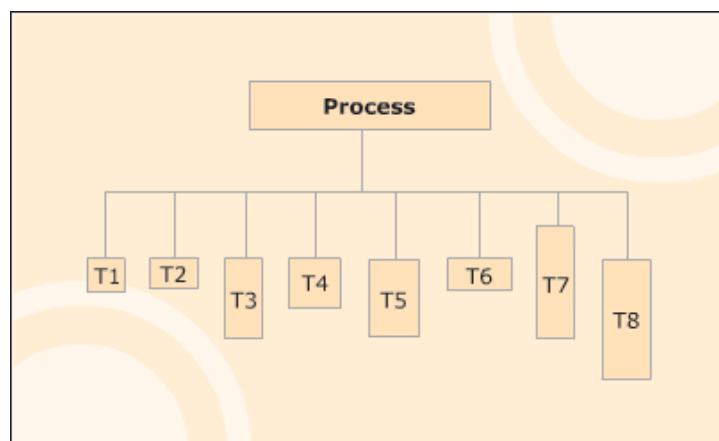


Figure 4.9: Managing Threads

Note: Take an example of a word processor; there are several threads running at a time, such as threads for accepting typed words, auto-saving and checking the spelling. Hence, it is necessary to manage the system resources effectively to run all the tasks without any conflicts. When the application is closed, all the threads should be terminated simultaneously.

4.5.1 Necessity for Thread Priority

While creating multi-threaded applications, situations may come up where a thread is already running and you are required to run another thread of greater importance. This is where thread priorities play an important role. Priorities are used to express the importance of different threads.

Priorities play an important part when there is heavy contention among threads trying to get a chance to execute. This prioritizing process is managed by the scheduler which assigns priority to the respective threads.

Note: Thread priority is like your daily life where you must prioritize your schedules or decisions based on some factors so that every work is carried out according to their priority.

4.5.2 Types of Thread Priority

Thread priority helps the thread scheduler to decide which thread to run. Priority also helps the operating system to decide the number of resources that has to be allocated to each thread. Thread priorities are integers ranging from `MIN_PRIORITY` to `MAX_PRIORITY`.

The higher the integers, the higher are the priorities. Higher priority threads get more CPU time than a low priority thread. Thread priorities in Java are constants defined in the `Thread` class. They are as follows:

➤ **Thread.MAX_PRIORITY**

It has a constant value of 10. It has got the highest priority.

➤ **Thread.NORM_PRIORITY**

It has a constant value of 5. It is the default priority.

➤ **Thread.MIN_PRIORITY**

It has a constant value of 1. It has the lowest priority.

Note: Every thread in Java has a priority. By default, the priority is `NORM_PRIORITY` or 5.

4.5.3 setPriority() Method

A newly created thread inherits the priority from the thread that created it. To change the priority of a thread, the `setPriority()` method is used. The `setPriority()` method changes the current priority of any thread. The `setPriority()` method accepts an integer value ranging from 1 to 10.

Syntax

```
public final void setPriority(int newPriority)
```

Code Snippet 15 demonstrates this.

Code Snippet 15:

```
...
Thread threadA = new Thread("Meeting deadlines");
threadA.setPriority(8);
```

An instance of the `Thread` class is created and its priority has been set to 8.

4.5.4 getPriority() Method

The `getPriority()` method helps to retrieve the current priority value of any thread. A query to know the current priority of the running thread to ensure that the thread is running in the required priority level.

Syntax

```
public final int getPriority()
```

4.6 Daemon Threads

A daemon thread runs continuously to perform a service, without having any connection with the overall state of the program. In general, the threads that run system codes are good examples of daemon threads.

The characteristics of the daemon threads are as follows:

- They work in the background providing service to other threads.
- They are fully dependent on the user threads.
- Java virtual machine stops once a thread dies and only daemon thread is alive.

The Thread class has two methods related to daemon threads. They are as follows:

1) **setDaemon (boolean value)**

The `setDaemon ()` method turns a user thread to a daemon thread. It takes a boolean value as its argument. To set a thread as daemon, the `setDaemon ()` method is invoked with true as its argument. By default, every thread is a user thread unless it is explicitly set as a daemon thread previously.

Syntax

```
void setDaemon (boolean val)
```

2) **isDaemon ()**

The `isDaemon ()` method determines if a thread is a daemon thread or not. It returns true if this thread is a daemon thread, else returns false.

Syntax

```
boolean isDaemon ()
```

Note: For instance, the garbage collector thread and the thread that processes mouse events for a Java program are daemon threads. On the other hand, User threads are the default threads which are important to the execution of the program.

4.6.1 Necessity for Daemon Threads

The task performed by the Daemon threads are as follows:

- 1) Daemon threads are service providers for other threads running in the same process.
- 2) Daemon threads are designed as low-level background threads that perform some tasks such as mouse events for Java programs.

Note: A thread can be set to daemon if the programmer does not want the main program to wait until a thread ends.

4.7 *Scoped Values in JDK 20*

4.7.1 *Overview*

Scoped values refer to a new feature in JDK version 20 that helps Java programmers manage values in a more organized way.

What are Scoped Values?

Think of scoped values as little containers that hold important information within the Java program. These containers have a specific scope or area where they are valid. It is like a secret note that is only useful in a certain classroom.

How Do Scoped Values Work in JDK 20?

In JDK 20, Java has introduced a way to create these scoped values more easily. A new keyword called `scoped` is added.

Code Snippet 16:

```
scoped int myScopedValue = 42;
```

In Code Snippet 16, a scoped value is created called `myScopedValue`, and it is set to 42. This value will only be valid and usable in the part of the code where it is declared.

4.7.2 *Benefits of Scoped Values*

Why Do We Require Scoped Values?

Imagine you are working on a big project with many variables (such as boxes to store things). Sometimes, you only want a variable to be valid and accessible in a specific part of your code (like a box that only exists in one room of your house). Scoped values help you do that - keep things organized and prevent mix-ups.

More benefits are listed as follows:

- 1) **Less Confusion:** Scoped values help prevent confusion because you know exactly where a value is supposed to be used.
- 2) **Improved Security:** It is like locking a door; you can make sure that a value is only accessible where it should be.
- 3) **Cleaner Code:** Scoped values make your code cleaner and easier to read because you know the context of each value.

4.7.3 Example in Real Life

Imagine you have a magic pen, but it only works in your bedroom. If you leave your bedroom, the pen does not work anymore. That is bit like how scoped values work in Java 20 – they are like magic variables that only work in specific parts of your code.

In summary, scoped values in JDK 20 are a cool new feature that help programmers organize their code better by keeping certain values restricted to specific areas. It is like having special keys that only unlock certain doors in your program.

4.7.4 Implementation of Scoped Values

Code Snippet 17 demonstrates use of scoped values.

Code Snippet 17:

```
public class ScopedValuesExample {  
  
    public static void main(String[] args) {  
        // Creating a scoped value  
        scoped int myScopedValue = 42;  
  
        // Using the scoped value within its scope  
        {  
            int result = myScopedValue + 10;  
            System.out.println("Result inside the scope: " + result);  
        }  
  
        // Attempting to use the scoped value outside its scope would result in  
        // an error  
        // int outsideScopeResult = myScopedValue + 20; // This would generate  
        // an error  
    }  
}
```

In Code Snippet 17, a scoped value is created called `myScopedValue` and assigned a value of 42.

Then, a block of code is enclosed in curly braces `{ }`. This defines the scope where `myScopedValue` is valid. Inside this scope, `myScopedValue` can be used to perform operations. In this example, we add 10 to this variable and print the result, which is 52. If we try to use `myScopedValue` outside of its declared scope (as shown in the commented-out line), it will result in an error because the scoped value is only accessible within the curly braces where it was declared.

4.8 Creating Virtual Threads

Starting from Java 16, the concept of Virtual Threads was introduced as part of Project Loom. Virtual Threads are lightweight, user-mode threads that can be created in Java, enabling programmers to write concurrent programs with a more efficient and scalable approach.

Virtual Threads were previously known as Fibers or Continuations.

Code Snippet 18 demonstrates creating a Java Virtual Thread using the `java.lang.Thread` class.

Code Snippet 18:

```
public class VirtualThreadExample {  
    public static void main(String[] args) {  
        /* Creating a virtual thread using the Thread.startVirtualThread  
        method */  
        Thread virtualThread = Thread.startVirtualThread(() -> {  
            for (int i = 1; i <= 5; i++) {  
                System.out.println("Virtual Thread: " + i);  
                try {  
                    // Simulating some work in the virtual thread  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
  
        /* The main thread continues executing while the virtual thread is  
        running concurrently*/  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("Main Thread: " + i);  
            try {  
                // Simulating some work in the main thread  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
  
        /* Wait for the virtual thread to complete before ending the program */  
        try{  
            virtualThread.join();  
        }catch(Exception e){}  
    }  
}
```

In Code Snippet 18, we create a virtual thread using the `Thread.startVirtualThread(Runnable)` method. The code inside the lambda expression represents the work that the virtual thread will perform. The virtual thread runs concurrently with the main thread, and both threads execute independently.

The `start()` method is used in this example to initiate the creation of a virtual thread, which then immediately begins executing the `Runnable` method that was supplied to it.

Since virtual threads are a preview feature in Java 20, to compile this code, you must use following command:

```
javac --enable-preview --release 20 VirtualThreadExample.java
```

Then, you can execute it as follows:

```
java --enable-preview VirtualThreadExample
```

The outcome of Code Snippet 18 would be:

```
Main Thread: 1
Virtual Thread: 1
Main Thread: 2
Main Thread: 3
Virtual Thread: 2
Main Thread: 4
Main Thread: 5
Virtual Thread: 3
Virtual Thread: 4
Virtual Thread: 5
```

You can use the `unstarted()` function instead of starting the virtual thread instantly if you do not want it to start immediately. Code Snippet 19 is an example of generating a virtual thread that has not yet started.

Code Snippet 19:

```
Thread vThreadUnstarted = Thread.ofVirtual().unstarted(runnable);
```

Simply calling the `start()` method on the virtual thread that has not yet been started will get the process started, as shown here in Code Snippet 20.

Code Snippet 20:

```
vThreadUnstarted.start();
```

4.9 Difference Between Virtual Threads and OS Threads

In Java, the term Virtual Threads refers to lightweight, user-mode threads introduced as part of Project Loom. On the other hand, OS threads refer to the traditional operating system-level threads. Let us explore the differences between the two with examples and code.

Table 4.3 depicts feature-wise differences between virtual threads and OS threads.

Feature	Virtual Threads	OS Threads
Creation and Control	Managed by the language	Managed by the operating system

Feature	Virtual Threads	OS Threads
	runtime or application framework.	system.
Lightweight	Extremely lightweight, often costing a few bytes of memory each.	Relatively heavyweight, with a more significant memory and resource overhead.
Concurrency Control	Uses Cooperative multitasking; threads yield control explicitly.	Uses Preemptive multitasking; the OS schedules threads automatically.
Parallelism	May or may not provide true parallelism depending on the runtime and hardware.	Designed to provide true parallelism, utilizing multiple CPU cores.
Concurrency Model	Typically used for high concurrency scenarios, such as handling thousands of connections simultaneously.	Suitable for managing a limited number of threads running in parallel.
Scalability	Well-suited for highly concurrent workloads and scenarios with a massive number of threads.	May encounter scalability issues when the number of threads becomes very large due to OS limitations.
Synchronization	Requires explicit synchronization mechanisms such as mutexes and semaphores to coordinate between threads.	Often provides built-in synchronization primitives such as mutexes, semaphores, and condition variables.
Fault Isolation	A failure in one virtual thread usually does not impact others, offering some fault isolation.	A crash or error in one OS thread can potentially affect the entire application.
Portability	Highly portable across different platforms and operating systems.	Less portable, as OS thread APIs can vary significantly between platforms.
Programming Complexity	Generally simpler to work with as they are managed at the application level.	Can be more complex due to lower-level OS interactions and thread management.
Examples	Green threads in languages such as Python, Go, and Java (before Java 11's <code>java.lang.Thread</code> changes).	POSIX threads (<code>pthread</code>) in Unix-like systems, Windows Threads (Win32), and others.

Table 4.3: Differences Between Virtual Threads and OS Threads

Consider Code Snippet 21 to understand the Virtual Threads and OS Threads.

Code Snippet 21:

```
import java.util.concurrent.*;

public class ThreadDemo {
    public static void main(String[] args) {
        // Creating an ExecutorService with a fixed thread pool
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        // Using OS threads (ExecutorService)
        executorService.execute(() -> {
            cpuBoundTask();
        });

        // Using CompletableFuture for asynchronous tasks (emulating virtual
        // threads)
        CompletableFuture<Void> ioTask = CompletableFuture.runAsync(() -> {
            ioBoundTask();
        });

        // Waiting for the OS thread to finish
        executorService.shutdown();

        // Waiting for the CompletableFuture to complete
        ioTask.join();

        System.out.println("All tasks completed");
    }

    // Function to simulate a CPU-bound task
    public static void cpuBoundTask() {
        long total = 0;
        for (int i = 0; i < 10_000_000; i++) {
            total += i;
        }
        System.out.println("CPU-bound task completed");
    }

    // Function to simulate an I/O-bound task
    public static void ioBoundTask() {
        try {
            Thread.sleep(2000); /* Sleep for 2 seconds to simulate I/O */
            System.out.println("I/O-bound task completed");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The code begins by importing necessary Java libraries for concurrent programming (`java.util.concurrent`).

Inside the main method:

- An `ExecutorService` named `executorService` is created with a fixed thread pool size of 2. This means it can execute upto two tasks concurrently.
- The `executorService` is used to execute a CPU-bound task using the `execute` method. It runs the `cpuBoundTask` method in a separate OS thread.
- A `CompletableFuture` named `ioTask` is created using `CompletableFuture.runAsync`. This emulates asynchronous tasks, often referred to as ‘virtual threads’ and runs the `ioBoundTask` method concurrently.
- The `executorService` is shut down, which means no more tasks can be submitted to it, but it will finish executing previously submitted tasks.
- `ioTask.join()` is used to wait for the `ioTask` `CompletableFuture` to complete. This ensures that the I/O-bound task finishes before moving on.

Finally, a message, ‘All tasks complete’ is printed to indicate that all tasks have finished.

The `cpuBoundTask` method simulates a CPU-bound task by performing a large loop computation. It calculates the sum of numbers from 0 to 10,000,000.

The `ioBoundTask` method simulates an I/O-bound task by causing the current thread to sleep for 2 seconds. This sleep represents waiting for I/O operations to complete.

4.9.1 Creation and Resource Overhead

OS Threads: Creating and managing OS threads involves significant overhead, as the operating system must allocate resources such as stack memory and manage thread synchronization and context switching.

Virtual Threads: Virtual threads have lower resource overhead compared to OS threads. They are managed within the JVM and do not require direct OS support for creation and management.

An example of using OS Threads to manage resource overload is shown in Code Snippet 22.

Code Snippet 22:

```
public class OSThreadsExample {  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(() -> {  
            for (int i = 1; i <= 5; i++) {  
                System.out.println("Thread 1: " + i);  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                }  
            }  
        });  
        thread1.start();  
    }  
}
```

```

        e.printStackTrace();
    }
}
});

Thread thread2 = new Thread(() -> {
    for (int i = 1; i <= 5; i++) {
        System.out.println("Thread 2: " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

thread1.start();
thread2.start();

try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Main Thread: Done!");
}
}

```

4.9.2 Concurrency Limit

OS Threads: The number of OS threads that can be created is typically limited by the operating system and hardware resources. Creating too many OS threads can lead to resource exhaustion and decreased performance.

Virtual Threads: It is possible to create millions of virtual threads without exhausting system resources. They offer a more scalable approach to concurrency.

Example using Virtual Threads to manage concurrency limit is shown in Code Snippet 23.

Code Snippet 23:

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class VirtualThreadsExample {
    public static void main(String[] args) {

```

```

ExecutorService executor = Executors.newVirtualThreadExecutor();

executor.submit(() -> {
    for (int i = 1; i <= 5; i++) {
        System.out.println("Virtual Thread 1: " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

executor.submit(() -> {
    for (int i = 1; i <= 5; i++) {
        System.out.println("Virtual Thread 2: " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

executor.shutdown();
try {
    executor.awaitTermination(Long.MAX_VALUE,
                               TimeUnit.NANOSECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Main Thread: Done!");
}
}

```

4.9.3 Synchronization

OS Threads: Synchronization between OS threads typically involves more overhead, as it requires OS-level locking mechanisms.

Virtual Threads: Virtual threads can avoid some of the synchronization overhead by using cooperative multitasking and lighter synchronization mechanisms suitable for user-mode threads.

In the Virtual Threads example, we use ExecutorService with Executors.newVirtualThreadExecutor() to create and manage virtual threads. The code runs concurrently on virtual threads, and the executor takes care of the underlying details. This approach allows efficient execution of tasks without the necessity for multiple OS threads.

4.10 Summary

- A thread is the smallest executable code in a Java program, which contributes to making it possible to run multiple tasks at a time.
- There are two ways of creating a Thread object. A thread object can be created by extending the Thread class that defines the `run()` method. It can also be created by declaring a class that implements the Runnable interface and passing the object to the Thread constructor.
- A newly created thread can be in following states: new, runnable, waiting, blocked, and terminated.
- A thread is said to be in blocked state while waiting for the lock of an object. When the `run()` method has been completely executed a thread terminates.
- The `getName()` method returns the name of the thread while `start()` allows a thread to be in runnable state.
- The `run()` method allows a thread to start executing whereas `sleep()` forces a running thread to wait for another thread.
- The `interrupt()` method redirects a thread to perform another task keeping aside what it was doing before.
- The `setPriority()` and `getPriority()` methods are used to assign and retrieve the priority of any thread respectively. While working with multiple threads, it is inevitable to prioritize the threads so that they will be executed in a sequence without interfering with each other.
- The daemon thread runs independently of the default user thread in a program, providing background services such as the mouse event, garbage collection, and so on. The `isDaemon()` method helps to find whether a thread is a daemon or not, whereas `setDaemon()` method changes a user thread to a daemon thread.
- Using scoped values, we are provided with a convenient construction that allows us to supply a thread (and, if necessary, a group of child threads) with a value that is thread-specific and can only be read. This value is read-only.
- Virtual Threads can be created using the `Thread.startVirtualThread(Runnable)` method, providing a more efficient and scalable approach to concurrency in Java.

4.11 Check Your Progress

1. Which of these statements related to characteristics and uses of processes and threads are true?

(A)	A process has a self-contained execution environment.
(B)	A thread is the smallest unit of executable code in an application that performs a particular task.
(C)	A thread has no definite starting point, execution steps and terminating point.
(D)	A process contains one or more threads inside it.
(E)	Threads can be used for playing sound and displaying images simultaneously.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

2. Which of the following code snippets creates a thread object and makes it runnable?

(A)	<pre>class MyRunnable implements Runnable { public void run() { ... //Implementation } } class TestThread { public static void main(String args[]) { Runnable mrObj = new MyRunnable(); Thread thObj=new Thread(mrObj); thObj.start(); }</pre>
(B)	<pre>class MyRunnable implements Runnable { public void run() { ... //Implementation } } public static void main(String args[]) { MyRunnable mrObj = new MyRunnable(); MyThread thObj=new MyThread(mrObj); }</pre>

(C)	<pre>class MyThread extends Thread public void run() { //Implementation } class TestThread { public static void main(String args[]) { MyThread myThread = new MyThread(); myThread.start(); } }</pre>
(D)	<pre>class MyThread extends Thread { public void run() { //Implementation } }</pre>

3. Which of these statements regarding different states of a thread are true?

(A)	A thread in the waiting state is alive but not running.
(B)	A thread is considered terminated when its <code>run()</code> method starts.
(C)	A thread is in a blocked state when it is waiting for the lock of another object.
(D)	The thread comes out of the runnable state when the <code>start()</code> method is invoked on it.
(E)	A new thread is an empty thread object with no system resources allocated.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

4. Which of the following method declarations of the `Thread` class are true?

(A)	<code>public void interrupt()</code>
(B)	<code>static void sleep(long millis)</code>
(C)	<code>public void run()</code>
(D)	<code>thread.start()</code>
(E)	<code>public String getName()</code>

(A)	A, C, D	(C)	A, C, E
(B)	B, C, D	(D)	A, B, D

5. Which of these statements regarding the priority of threads are true?

(A)	The <code>getPriority()</code> method retrieves the current priority of any thread.
(B)	The <code>setPriority()</code> method changes the current priority of any thread.
(C)	Thread priorities are numbers and range from <code>Thread.MIN_PRIORITY</code> to <code>Thread.MAX_PRIORITY</code> .
(D)	A newly created thread inherits the priority of its parent thread.
(E)	The higher the integers, the lower are the priorities.

(A)	A, C, E	(C)	A, C, D, E
(B)	A, B, C, D	(D)	A, B, D

6. Which of these statements about daemon threads are true?

(A)	Daemon threads are service providers for other threads running in different process.
(B)	Daemon threads are designed as low-level background threads that perform some tasks such as mouse events.
(C)	The threads that run system code are user threads.
(D)	The <code>setDaemon()</code> method with the argument as true will convert the user thread to a daemon thread.
(E)	The <code>isDaemon()</code> method returns true if the thread is a user thread.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

4.11.1 Answers

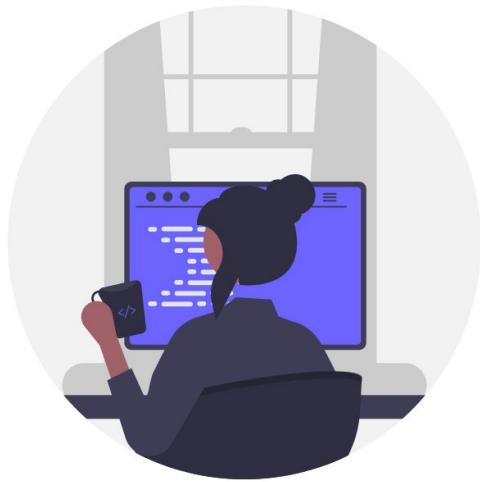
1.	D
2.	A
3.	A
4.	C
5.	B
6.	D

Try It Yourself

1. You are tasked with creating a Java program that simulates a simple ticket booking system for a movie theater. The theater has multiple screens and customers can book seats for different movies. To handle concurrent bookings, you must use concept of threading.

Write a Java program that accomplishes following tasks:

- i. Create a MovieTheater class that represents the theater. This class should have methods for booking seats, checking seat availability, and displaying available movies and their showtimes.
- ii. Implement a BookingAgent class that represents a customer booking agent. Each booking agent should have a unique name and be capable of making seat reservations for customers.
- iii. Ensure that the MovieTheater class is thread-safe to handle multiple agents booking seats simultaneously. Use appropriate synchronization mechanisms to avoid double-booking seats.
- iv. In the main method, create an instance of the MovieTheater and several BookingAgent instances. Start the booking agent threads and simulate customers booking seats for different movies.



Session 5

Multithreading and Concurrency

Welcome to the Session, **Multithreading and Concurrency**.

This session describes multithreading, which allows two parts of the program to run simultaneously. The session also explains the concept of synchronization. The mechanism for inter-process communication will also be discussed. The session describes how to deal with deadlocks which cause two processes to be waiting for a resource and also describes `java.util.concurrent` collections.

In this Session, you will learn to:

- Define multithreading
- Differentiate between multithreading and multitasking
- Explain the use of `isAlive()` and `join()` method
- Explain race conditions and ways to overcome them
- Elaborate on intrinsic lock and synchronization
- Identify atomic access
- Identify the use of `wait()` and `notify()` methods
- Define deadlock and the ways to overcome deadlock
- Explain `java.util.concurrent` collections
- Explain about Structured Concurrency
- Elaborate on the process to build radix sort program with Foreign API
- Outline the use of off-heap and on-heap memory areas

5.1 Multithreading in Java

Multithreading is a specialized form of multitasking. Differences between multithreading and multitasking have been provided in Table 5.1.

Multithreading	Multitasking
In a multithreaded program, two or more threads can run concurrently.	In a multitasking environment, two or more processes run concurrently.
Multithreading requires less overhead. In case of multithreading, threads are lightweight processes. Threads can share same address space and inter-thread communication is less expensive than inter-process communication.	Multitasking requires more overhead. Processes are heavyweight tasks that require their own address space. Inter-process communication is very expensive and the context switching from one process to another is costly.

Table 5.1: Differences Between Multithreading and Multitasking

Necessity for Multithreading

Multithreading is required for following reasons:

- Multithreading increases performance of single-processor systems, as it reduces the CPU idle time.
- Multithreading encourages faster execution of a program when compared to an application with multiple processes, as threads share the same data whereas processes have their own sets of data.
- Multithreading introduces the concept of parallel processing of multiple threads in an application which services a huge number of users.

The Java programming language provides ample support for multithreading by means of the `Thread` class and `Runnable` interface. Code Snippet 1 creates multiple threads, displays the count of the threads, and displays the name of each running child thread within the `run()` method.

Code Snippet 1:

```
/**
Creating multiple threads using a class derived from Thread class
*/
package test;
/**
MultipleThreads is created as a subclass of the class Thread
*/
public class MultipleThreads extends Thread {
// Variable to store the name of the thread
String name;
/**
This method of Thread class is overridden to specify the action
that will be done when the thread begins execution.
*/
public void run() {
while(true) {
name = Thread.currentThread().getName();
System.out.println(name);
try {
Thread.sleep(500);
}
catch(InterruptedException e) {
e.printStackTrace();
}
}
}
}
```

```
        } catch(InterruptedException e) {
            break;
        }
    } // End of while loop
}
/***
 * This is the entry point for the MultipleThreads class.
 */
public static void main(String args[]) {
    MultipleThreads t1 = new MultipleThreads();
    MultipleThreads t2 = new MultipleThreads();
    t1.setName("Thread2");
    t2.setName("Thread3");
    t1.start();
    t2.start();
    System.out.println("Number of threads running: " +
        Thread.activeCount());
}
```

In the code, the `main()` method creates two child threads by instantiating the `MultipleThreads` class which has been derived from the `Thread` class. The names of the child threads are set to `Thread2` and `Thread3` respectively. When the `start()` method is invoked on the child thread objects, the control is transferred to the `run()` method which will begin thread execution.

Just as the child threads begin to execute, the number of active threads is printed in the `main()` method.

Figure 5.1 shows the output of the code.

```
run:  
Thread2  
Number of threads running: 3  
Thread3  
Thread2  
Thread3  
Thread2  
Thread3  
Thread3  
Thread2  
Thread3  
Thread2  
Thread2  
Thread2
```

Figure 5.1: Output

The code executes until the user presses **Ctrl + C** to stop execution of the program.

Thread execution stops once it finishes the execution of `run()` method. Once stopped, thread execution cannot restart by using the `start()` method. Also, the `start()` method cannot be invoked on an already running thread. This will also throw an exception of type `IllegalThreadStateException`.

Threads eat up a lot memory (RAM) therefore, it is always advisable to set the references to null when a thread has finished executing. If a Thread object is created but fails to call the `start()` method, it will not be eligible for garbage collection even if the underlying application has removed all references to the thread.

5.2 More Methods of Thread Class

The `Thread` class supports methods to check the live status of a thread and to make the current thread to wait until its calling thread terminates respectively.

5.2.1 `isAlive()` Method

The thread that is used to start the application should be the last thread to terminate. This indicates that the application has terminated. This can be ensured by stopping the execution of the main thread for a longer duration within the main thread itself. Also, it should be ensured that all the child threads terminate before the main thread. However, how does one ensure that the main thread is aware of the status of the other threads? There are ways to find out if a thread has terminated. First, by using the `isAlive()` method.

A thread is considered to be alive when it is running. The `Thread` class includes a method named `isAlive()`. This method is used to find out whether a specific thread is running or not. If the thread is alive, then the boolean value `true` is returned. If the `isAlive()` method returns `false`, it is understood that the thread is either in new state or in terminated state.

Syntax

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

Code Snippet 2 displays the use of `isAlive()` method.

Code Snippet 2:

```
public class ThreadDemo extends Thread {  
    public static void main(String args[]) {  
        ThreadDemo Obj = new ThreadDemo ();  
        Thread t = new Thread(Obj);  
        System.out.println("The thread is alive: " + t.isAlive());  
    }  
}
```

The code demonstrates the use of `isAlive()` method. The method returns a boolean value of true or false depending whether the thread is running or terminated. The code here will return `false` since the thread is in new state and is not running.

5.2.2 `join()` Method

The `join()` method causes the current thread to wait until the thread on which it is called terminates.

The `join()` method performs following operations:

- This method allows specifying the maximum amount of time that the program should wait for the particular thread to terminate.
- It throws `InterruptedException` if another thread interrupts it.
- The calling thread waits until the specified thread terminates.

Syntax

```
public final void join()
```

Code Snippet 3 displays the use of `join()` method. Assume that `objTh` is an instance of a class derived from `Thread` class.

Code Snippet 3:

```
public class ThreadDemo extends Thread {  
    public static void main(String args[]) {  
        ThreadDemo objTh = new ThreadDemo();  
        Thread t = new Thread(objTh);  
        try {  
            t.start();  
            System.out.println("Number of threads running: " +  
                Thread.activeCount());  
            System.out.println("I am in the main and waiting for the thread to  
                finish");  
            objTh.join(); // objTh is a Thread object  
        }  
        catch (InterruptedException e) {  
            System.out.println("Main thread is interrupted");  
        }  
    }  
}
```

The code illustrates the use of the `join()` method. In this snippet, the current thread waits until the thread, `objTh` terminates.

The `join()` method of the `Thread` class has two other overloaded versions:

- **void join(long timeout)**

In this type of `join()` method, an argument of type `long` is passed. The amount of timeout is in milliseconds. This forces the thread to wait for the completion of the specified thread until the given number of milliseconds elapses.

- **void join(long timeout, int nanoseconds)**

In this type of `join()` method, arguments of type `long` and `integer` are passed. The amount of timeout is given in milliseconds in addition to a specified amount of nanoseconds. This forces the thread to wait for the completion of the specified thread until the given timeout elapses.

Code Snippet 4 displays the use of different methods of the Thread class.

Code Snippet 4:

```
/*
 * Using the isAlive and join methods
 */
package test;
/** ThreadDemo inherits from Runnable interface */
class ThreadDemo implements Runnable {
String name;
Thread objTh;
/* Constructor of the class */
ThreadDemo(String str) {
name = str;
objTh = new Thread(this, name);
System.out.println("New Threads are starting :" + objTh);
objTh.start();
}
public void run() {
try {
for (int count = 0; count < 2; count++) {
System.out.println(name + ":" + count);
objTh.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted");
}
System.out.println(name + " exiting");
}
public static void main(String [] args) {
ThreadDemo objNew1 = new ThreadDemo("one");
ThreadDemo objNew2 = new ThreadDemo ("two");
ThreadDemo objNew3 = new ThreadDemo ("three");
System.out.println("First thread is alive: " +
    objNew1.objTh.isAlive());
System.out.println("Second thread is alive: " +
    objNew2.objTh.isAlive());
System.out.println("Third thread is alive: " +
    objNew3.objTh.isAlive());
try {
System.out.println("In the main method, waiting for the threads to
    finish");
objNew1.objTh.join();
objNew2.objTh.join();
objNew3.objTh.join();
} catch (InterruptedException e) {
System.out.println("Main thread is interrupted");
System.out.println("First thread is alive :" +
    objNew1.objTh.isAlive());
System.out.println("Second thread is alive :" +
```

```

        objNew2.objTh.isAlive());
System.out.println("Third thread is alive :" +
        objNew3.objTh.isAlive());
System.out.println("Main thread is over and exiting");
}
}
}

```

In the code, three thread objects are created in the `main()` method. The `isAlive()` method is invoked by the three thread objects to test whether they are alive or dead. Then, the `join()` method is invoked by each of the thread objects. The `join()` method ensures that the main thread is the last one to terminate.

Finally, the `isAlive()` method is invoked again to check whether the threads are still alive or dead. These statements are enclosed inside a `try-catch` block.

Figure 5.2 shows the output of the code.

```

run:
New Threads are starting : Thread[one,5,main]
New Threads are starting : Thread[two,5,main]
New Threads are starting : Thread[three,5,main]
First thread is alive :true
Second thread is alive :true
Third thread is alive :true
In the main method, waiting for the threads to finish
two : 0
one : 0
three : 0
one : 1
three : 1
two : 1
three exiting
two exiting
one exiting
BUILD SUCCESSFUL (total time: 3 seconds)

```

Figure 5.2: Output

5.3 Thread Synchronization

In multithreaded programs, several threads may simultaneously try to update the same resource, such as a file. This leaves the resource in an undefined or inconsistent state. This is called race condition.

5.3.1 Race Conditions

In general, race conditions in a program occur when:

- Two or more threads share the same data between them.
- Two or more threads try to read and write the shared data simultaneously.

The race conditions can be avoided by using synchronized blocks. This is a block of code qualified by the `synchronized` keyword.

5.3.2 Synchronized Blocks and Methods

Consider a situation where people are standing in a queue outside a telephone booth. They wish to make a call and are waiting for their turn. This is similar to a synchronized way of accessing data where each thread wanting to access data waits its turn. However, going back to the analogy described earlier, if there was no queue and people were permitted to go in randomly, two or more persons would try to enter the booth at the same time, resulting in confusion and chaos. This is similar to a race condition that can take place with threads.

When two threads attempt to access and manipulate the same object and leave the object in an undefined state, a race condition occurs. Java provides the `synchronized` keyword to help avoid such situations. The core concept in synchronization with Java threads is something called a monitor. A monitor is a piece of code that is guarded by a mutual-exclusion program called a mutex. A real-life analogy for a monitor can be the telephone booth described earlier, but this time with a lock. Only one person can be inside the telephone booth at a time and while the person is inside, the booth will be locked, thus preventing the others from entering.

The telephone inside the booth here is the real-life equivalent of an object, the booth is the monitor and the lock is the mutex. A Java object has only one monitor and mutex associated with it.

Thus, synchronized blocks are used to prevent the race conditions in Java applications. The synchronized block contains code qualified by the `synchronized` keyword. A lock is assigned to the object qualified by `synchronized` keyword. When a thread encounters the `synchronized` keyword, it locks all the doors on that object, preventing other threads from accessing it. A lock allows only one thread at a time to access the code. When a thread starts to execute a synchronized block, it grabs the lock on it. Any other thread will not be able to execute the code until the first thread has finished and released the lock. The lock is based on the object and not on the method.

The syntax to create the synchronized block is as follows:

Syntax

```
synchronized(object) {  
// statements to be synchronized  
}
```

where,

`object` is the reference of the object being synchronized.

Code Snippet 5 demonstrates the synchronized block. Assume relevant imports have been added.

Code Snippet 5:

```
class Account {  
double balance = 200.0;  
public void deposit(double amount) {  
balance = balance + amount;  
}  
public void displayBalance() {  
System.out.println("Balance is: "+balance);  
}
```

```

}
}

class Transaction implements Runnable {
double amount;
Account account;
Thread t;
public Transaction(Account acc, double amt) {
account = acc;
amount = amt;
t = new Thread(this);
t.start();

}

// Synchronized block calls deposit method
public void run() {
synchronized (account) {
// Synchronized block
account.deposit(amount);
account.displayBalance();
}
}
}

public class DepositAmount {
public static void main(String[] args) {
Account accObj = new Account();
Transaction t1 = new Transaction(accObj, 500.00);
Transaction t2 = new Transaction(accObj, 200.00);
}
}

```

The code creates an **Account** class with methods namely, **deposit()** and **displayBalance()** which will add the amount to the existing balance and display it. The **Transaction** class creates a **Thread** object and calls the **run()** method on it. The **run()** method contains a synchronized block. The synchronized block takes the account object which will act as a monitor object. This allows only one thread to be executed inside the synchronized block on the same monitor object.

Figure 5.3 displays the output of the code.

```

run:
Balance is:700.0
Balance is:900.0
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 5.3: Output - Synchronized Block

5.3.3 Synchronized Methods

The synchronized method obtains a lock on the class object. This means that at a single point of time only one thread obtains a lock on the method, while all other threads require to wait to invoke the

synchronized method.

Consider a scenario where two threads require to perform the read and write operation on a single file stored on the system. If both threads attempt to manipulate the file data for read or write operations simultaneously, it may leave the file in inconsistent state. To prevent this, a synchronized method can be defined. Each thread will acquire a lock on the method to perform the respective operation. Thus, both the thread cannot invoke the method at the same time, as it will be locked by the other thread.

The syntax to declare a synchronized method is as follows:

Syntax

```
synchronized method(....)
{
// body of method
}
```

Note: Constructors cannot be synchronized.

Code Snippet 6 shows how to use a synchronized method.

Code Snippet 6:

```
/**
 * Demonstrating synchronized methods.
 */
package test;
class One {
    // This method is synchronized to use the thread safely
    synchronized void display(int num) {
        System.out.print(num);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println(" done");
    }
}
class Two extends Thread {
    int number;
    One objOne;
    public Two(One one_num, int num) {
        objOne = one_num;
        number = num;
    }
    public void run() {
        // Invoke the synchronized method
        objOne.display(number);
    }
}
```

```

}
public class SynchMethod {
public static void main(String args[]) {
One objOne = new One();
int digit = 10;
// Create three thread objects
Two objSynch1 = new Two(objOne, digit++);
Two objSynch2 = new Two(objOne, digit++);
Two objSynch3 = new Two(objOne, digit++);
objSynch1.start();
objSynch2.start();
objSynch3.start();
}
}

```

Here, the class **One** has a method **display()** that takes an **int** parameter. This number is displayed with a suffix "done". The **Thread.sleep(1000)** method pauses the current thread after the method **display()** is called.

The constructor of the class **Two** takes a reference to an object **t** of the class **One** and an integer variable. Here, a new thread is also created. This thread calls the method **run()** of the object **t**. The main class **SynchDemo** instantiates the class **One** as an object **objOne** and creates three objects of the class **Two**. The same object **objOne** is passed to each **Two** object. The method **join()** makes the caller thread wait till the calling thread terminates.

The output of the Code Snippet 6 could be as follows depending upon how the CPU schedules the threads:

```

10 done
11 done
12 done

```

It is not always possible to achieve synchronization by creating synchronized methods within classes. The reason for this is as follows:

Consider a case where the programmer wants to synchronize access to objects of a class, which does not use synchronized methods. Also assume that the source code is unavailable because either a third party created it or the class was imported from the built-in library. In such a case, the keyword **synchronized** cannot be added to the appropriate methods within the class.

Therefore, the problem here would be how to make the access to an object of this class synchronized. This could be achieved by putting all calls to the methods defined by this class inside a synchronized block.

5.3.4 Intrinsic Locks and Synchronization

Synchronization is built around the concept of an in-built monitor which is also referred to as intrinsic lock or monitor lock. The monitor lock enforces exclusive access to the thread objects, thus creating a relationship between the thread action and any further access of the same lock.

This helps to make thread relationship visible.

Every object is connected to an intrinsic lock. Typically, a thread acquires the object's intrinsic lock before accessing its fields, and then, releases the intrinsic lock. In this span of acquiring and releasing the intrinsic lock, the thread owns the intrinsic lock. No other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a 'happens-before' relationship is established between that action and any subsequent acquisition of the same lock.

When a thread invokes a synchronized method, the following occurs:

- It automatically acquires the intrinsic lock for that method's object.
- It releases it when the method returns.

The lock is released even if the return is caused by an uncaught exception.

If a static method is associated with a class, the thread gets the intrinsic lock for the Class object associated with the class. Therefore, the lock that controls the access to the class's static fields is different from the lock for any instance of the class.

Synchronized code can also be created with synchronized statements. These statements should specify the object that provides the intrinsic lock.

Synchronized statements also help improve concurrency with fine-grained synchronization.

5.4 wait-notify Mechanism

Java also provides a wait and notify mechanism to work in conjunction with synchronization. The wait-notify mechanism acts as the traffic signal system in the program. It allows the specific thread to wait for some time for other running thread and wakes it up when it is required to do so. For these operations, it uses the `wait()`, `notify()` and `notifyAll()` methods.

In other words, the wait-notify mechanism is a process used to manipulate the `wait()` and `notify()` methods. This mechanism ensures that there is a smooth transition of a particular resource between two competitive threads.

It also oversees the condition in a program where one thread is:

- Allowed to wait for the lock of a synchronized block of resource currently used by another thread.
- Notified to end its waiting state and get the lock of that synchronized block of resource.

5.4.1 `wait()` Method

The `wait()` method causes a thread to wait for some other thread to release a resource. It forces the currently running thread to release the lock or monitor, which it is holding on an object. Once the resource is released, another thread can get the lock and start running. The `wait()`

method can only be invoked only from within the synchronized code.

Following points should be remembered while using the `wait()` method:

- The calling thread gives up the CPU and lock.
- The calling thread goes into the waiting state of monitor.

Syntax

```
public final void wait()
```

The 'dining philosophers' problem is a popular example used in Java programming world to demonstrate synchronization and concurrency control. According to this example, there are five philosophers and their way of life is to think and eat alternately. They share a round table and sit on five chairs. There is a bowl of rice for each of the philosophers, but only five chopsticks, not ten.

Each philosopher requires both their right and left chopstick to eat. A hungry philosopher is able to only eat if there are both chopsticks available. Otherwise, a philosopher has to wait for his/her turn until both chopsticks and begins thinking again.

Code Snippet 7 makes use of this scenario to demonstrate how `wait()` method works.

Code Snippet 7:

```
import java.io.*;
import java.util.*;
class Chopstick{
boolean available;
Chopstick(){
this.available=true;
}
//Pick up the chopsticks
public synchronized void takeUp () {
// As long as someone is already using and chopstick is not available
while (!available) {
try{
System.out.println("Waiting to eat...");
```

//Enter the waiting queue

```
wait();
```

} catch (InterruptedException e) { }

```
}
// Received the chopstick so mark it as unavailable for others
available=false;
}
// Put down the chopsticks
public synchronized void putDown () {
// Finished eating then, mark it as available so that other people can use
available=true;
}
}
// Philosophers
```

```
class Philosopher extends Thread{
Chopstick left;
Chopstick right;
int ID;
// Parameterized Constructor
public Philosopher(Chopstick left, Chopstick right, int ID) {
this.left = left;
this.right = right;
this.ID = ID+1;
}
//Dining
public void eat() {
left.takeUp();
right.takeUp();
System.out.println(ID+" : The Philosopher is Dining");
}
//Thinking
public void think() {
left.putDown();
right.putDown();
System.out.println(ID+" : The Philosopher is Thinking");
}
public void run() {
while (true) {
eat();
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
think();
try {
Thread.sleep(1000);
} catch (InterruptedException ee) {
}
}
}
}
// Class to test the functionality
public class DiningDemo {
public static void main(String[] args) throws IOException {
int i;
// 5 chopsticks, 5 philosophers
Philosopher[] philosopher = new Philosopher[5];
Chopstick[] chopstick = new Chopstick[5];
//Instantiate
for(i=0;i<5;i++) {
chopstick[i] = new Chopstick();
}
//Instantiate
```

```

for(i=0;i<5;i++) {
philosopher[i] = new
    Philosopher(chopstick[i],chopstick[(i+1)%5],i);
}
//Start the process
for(i=0;i<5;i++) {
philosopher[i].start();
}
}
}

```

Code Snippet 7 demonstrates the use of `wait()` method. The `wait()` method forces the currently running thread to pause so that another thread can get hold of the resource.

5.4.2 *notify()* Method

The `notify()` method alerts the thread that is waiting for a monitor of an object. This method can be invoked only within a synchronized block. If several threads are waiting for a specific object, one of them is selected to get the object. The scheduler decides this based on the requirement of the program.

The `notify()` method functions in following way:

- The waiting thread moves out of the waiting space of the monitor and into the ready state.
- The thread that was notified is now eligible to get back the monitor's lock before it can continue.

Syntax

```
public final void notify()
```

Code Snippet 8 displays the use of the `notify()` method that can be added in the `putDown()` method of Code Snippet 7.

Code Snippet 8:

```

public synchronized void putDown() {
available=true;
// Inform others of availability
notify();
}

```

This code illustrates the use of the `notify()` method. This method will notify the waiting thread to resume its execution.

5.4.3 *Thread.onSpinWait()*

All threads follow the order of execution. Low priority threads are executed first, followed by high priority threads. Creating thread is a daemon thread if and only if thread is daemon and is low priority.

`Thread.onSpinWait()` is a method first introduced in Java 9. It informs CPU that the current thread running is burning more CPU cycles and is currently unable to progress. Then, CPU allocates more resources to other threads at the cost of loading the OS scheduler and dequeuing the current thread method.

Usage of `onSpinWait()`:

- It is suitable when threads are waiting for a long time for an external event to occur.
- It allows events that have occurred to finish very quickly, thus avoiding a long wait time.

In case of `wait()` and `notify()` pattern methods, the other thread waits for the waiting or sleeping thread to wake up. Sometimes, the other thread is unaware of the fact, that all other waiting threads are yet to be notified, if the developer fails to control the other thread. There is no way to get notified. In such a case, `onSpinWait()` is used.

Code Snippet 9 shows an example for verifying whether a product is received or not by the customer in the online shopping cart by checking a condition in a `while` loop. The code prints the status on the screen and the customer is later notified using `sendNotification()` method. Here, `while` loop execution is very frequent and it consumes too much electricity/CPU power.

Code Snippet 9:

```
...
while(true) {
    while(!isProductReceived) {
        System.out.println("Product received");
    }
    sendNotification();
}
...
```

The output prints following text until the condition becomes false:

Product received Product received Product received

.

.

The `while` loop in the code executes continuously until the cart is empty. Since this execution is continuous, it consumes more electricity.

In versions of Java prior to 9, the `sleep()` method was executed in order to reduce electricity. Refer to Code Snippet 10.

Code Snippet 10:

```
...
while(true) {
    while(!isProductReceived) {
        try{
            Thread.sleep(5000);
        }
```

```
System.out.println("Product received");
} catch(InterruptedException e) {}
}
sendNotification();
}
...
.
```

The output prints following text until the condition becomes false:

```
Product received
Product received
Product received
.
.
```

In order to replace short term sleep calls and to avoid consuming 100% CPU usage, `onSpinWait()` method was released in Java 9. This makes applications more responsive. Code Snippet 11 shows an example of executing `onSpinWait()` method.

Code Snippet 11:

```
...
while(true) {
while(!isProductReceived) {
Thread.onSpinWait();
System.out.println("Product received");
}
sendNotification();
}
...
.
```

In this case, electricity/CPU power consumption is greatly reduced. This way, the CPU can allocate electricity/power to other resources as well.

5.5 Deadlocks

Deadlock describes a situation where two or more threads are blocked forever, waiting for the others to release a resource. At times, it happens that two threads are locked to their respective resources, waiting for the corresponding locks to interchange the resources between them. In that situation, the waiting state continues forever as both are in a state of confusion as to which one will leave the lock and which one will get into it. This is the deadlock situation in a thread-based Java program. The deadlock situation brings the execution of the program to a halt.

Figure 5.4 displays a deadlock condition.

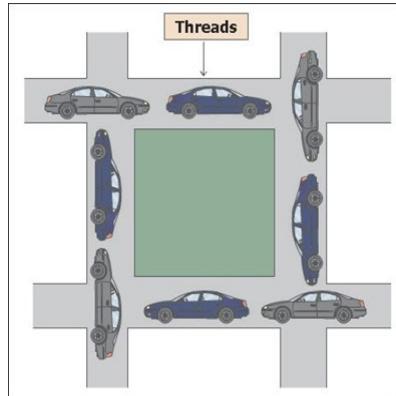


Figure 5.4: Deadlock

It is difficult to debug a deadlock since it occurs very rarely. Code Snippet 12 demonstrates a deadlock condition.

Code Snippet 12:

```
/*
Demonstrating Deadlock.
DeadlockDemo implements the Runnable interface.
*/
public class DeadlockDemo implements Runnable {
public static void main(String args[]) {
DeadlockDemo objDead1 = new DeadlockDemo();
DeadlockDemo objDead2 = new DeadlockDemo();
Thread objTh1 = new Thread (objDead1);
Thread objTh2 = new Thread (objDead2);
objDead1.grabIt = objDead2;
objDead2.grabIt = objDead1;
objTh1.start();
objTh2.start();
System.out.println("Started");
try {
objTh1.join();
objTh2.join();
} catch(InterruptedException e) {
System.out.println("error occurred");
}
System.exit(0);
}
DeadlockDemo grabIt;
public synchronized void run() {
try {
Thread.sleep(500);
} catch(InterruptedException e) {
System.out.println("error occurred");
}
grabIt.syncIt();
}
public synchronized void syncIt() {
```

```

try {
    Thread.sleep(500);
    System.out.println("Sync");
} catch(InterruptedException e) {
    System.out.println("error occurred");
}
System.out.println("In the syncIt() method");
}
} // end class

```

The program creates two child threads. Each thread calls the synchronized `run()` method. When thread `objTh1` wakes up, it calls the method `syncIt()` of the `DeadlockDemo` object `objDead1`. Since the thread `objTh2` owns the monitor of `objDead2`, thread `objTh1` begins waiting for the monitor.

When thread `objTh2` wakes up, it tries to call the method `syncIt()` of the `DeadlockDemo` object `objDead2`. At this point, `objTh2` also is forced to wait since `objTh1` owns the monitor of `objDead1`. Since both threads are waiting for each other, neither will wake up. This is a deadlock condition. The program is blocked and does not proceed further.

Figure 5.5 shows the output of the code.



Figure 5.5: Output – Deadlock Condition

Overcoming the Deadlock

You can plan for the prevention of deadlock while writing the codes. You can ensure any of the following in a program to avoid deadlock situations in it:

- Avoid acquiring more than one lock at a time.
- Ensure that in a Java program, you acquire multiple locks in a consistent and defined order.

Java does not provide any mechanism for detection or control of potential deadlock situations. The programmer is responsible for avoiding them.

5.6 Concurrency Utilities

Java platform has added a rich concurrency library for large applications executed on multiple processors environment. The concurrency library is the new addition in the `java.util` package.

It also provides new concurrent data structures in the Collections Framework.

5.6.1 *java.util.concurrent Collections*

Following are some of the collections that are categorized by the Collection interfaces:

- **BlockingQueue**: This defines a FIFO data structure that blocks or times out when data is added to a full queue or retrieved from an empty queue.
- **ConcurrentMap**: This is a sub interface of `java.util.Map` that defines useful atomic operations. These operations add a key-value pair only if the key is absent. They can also remove or replace a key-value pair only if the key is present. Such operations can be made atomic to avoid synchronization.
- **ConcurrentNavigableMap**: This is a sub interface of `ConcurrentMap` that supports approximate matches.

5.6.2 *Atomic Variables*

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables. All classes include get and set methods that work similar to reads and writes on volatile variables. Therefore, a set has a happens-before relationship with any successive get on the same variable.

Code Snippet 13 displays the use of `AtomicVariableApplication` class.

Code Snippet 13:

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicVariableApplication {
    private final AtomicInteger value = new AtomicInteger(0);
    public int getValue() {
        return value.get();
    }
    public int getNextValue() {
        return value.incrementAndGet();
    }
    public int getPreviousValue() {
        return value.decrementAndGet();
    }
    public static void main(String[] args) {
        AtomicVariableApplication obj = new AtomicVariableApplication();
        System.out.println(obj.getValue());
        System.out.println(obj.getNextValue());
        System.out.println(obj.getPreviousValue());
    }
}
```

Figure 5.6 displays the output.

```

run:
0
1
0
BUILD SUCCESSFUL (total time: 3 seconds)

```

Figure 5.6: AtomicVariableApplication Class – Output

5.6.3 Executors and Executor Interface

Objects that separate thread management and creates them from the rest of the application are called executors.

The `java.util.concurrent` package defines following three executor interfaces:

- **Executor**: This helps launch new tasks. The `Executor` interface includes a single method, `execute`. It is designed to be a drop-in replacement for a common thread-creation idiom. If `r` is a `Runnable` object and `e` is an `Executor` object, `(new Thread(r)).start();` can be replaced with `e.execute(r);`

The executor implementations in `java.util.concurrent` use advanced `ExecutorService` and `ScheduledExecutorService` interfaces.

The low-level idiom creates a new thread and launches it immediately. Depending on the `Executor` implementation, `execute()` uses an existing worker thread to run `r`. It can also place `r` in a queue to wait for a worker thread to become available.

- **ExecutorService**: This is a sub interface of `Executor` and helps manage the development of the executor tasks and individual tasks. The interface provides `execute()` with a resourceful `submit()` method that accepts `Runnable` objects and `Callable` objects. The latter allow the task to return a value. The `submit()` method returns a `Future` object, which retrieves the `Callable` return value. The object also manages the status of `Callable` and `Runnable` tasks.

`ExecutorService` provides methods to submit large collections of `Callable` objects. It also includes various methods to manage the shutdown of the executor.

- **ScheduledExecutorService**: This is a subinterface of `ExecutorService` and helps periodic execution of tasks.

The interface provides a schedule for the methods of its parent `ExecutorService`.

The `schedule` executes a `Runnable` or `Callable` task after a specified delay.

The interface also defines `scheduleAtFixedRate()` and `scheduleWithFixedDelay()`. At defined intervals, these execute specified tasks repeatedly.

5.6.4 ThreadPools

Thread pools have worker threads that help create threads and thus, minimize the overhead. Certain executor implementations in `java.util.concurrent` use thread pools. Thread pools are often

used to execute multiple tasks. Allocating and deallocating multiple thread objects creates a considerable memory management overhead in a large-scale application.

Fixed thread pool is a common type of thread pool that includes following features:

- There are a specified number of threads running.
- When in use if a thread is terminated, it is automatically replaced with a new thread.
- Applications using fixed thread pool services HTTP requests as quickly as the system sustains.

This class provides following factory methods:

newCachedThreadPool method:

This static factory method creates an executor with an expandable thread pool. This is suitable for applications that launch many short-lived tasks.

newSingleThreadExecutor method:

This static factory method creates an executor that executes one task at a time.

5.7 Fork/Join Framework

This is an implementation of the `ExecutorService` interface. The framework helps work with several processors to boost the performance of an application. It uses a work-stealing algorithm and is used when work is broken into smaller pieces recursively.

The Fork/Join framework allocates tasks to worker threads in a thread pool.

There is the `ForkJoinPool` class in the fork/join framework. The class is an extension of the `AbstractExecutorService` class. The `ForkJoinPool` class implements the main work-stealing algorithm and executes `ForkJoinTask` processes.

Following describes the basic steps to use the fork/join framework:

1. Write the code that performs a segment of the work. The code should resemble following pseudocode:

```
if (my portion of the work is small
    enough) do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```
2. Wrap the code in a `ForkJoinTask` subclass, typically using `RecursiveTask` that returns a result or `RecursiveAction`.
3. Create the object for all the tasks to be done.
4. Pass the object to the `invoke()` method of a `ForkJoinPool` instance. Code Snippet 14 displays the use of Fork/Join functionality.

Code Snippet 14:

```
package threadapplication;
import java.util.Random;
```

```

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class ForkJoinApplication extends RecursiveTask<Integer> {
private static final int SEQUENTIAL_THRESHOLD = 5;
private final int[] data;
private final int startData;
private final int endData;
public ForkJoinApplication(int[] data, int startValue, int endValue) {
    this.data = data;
    this.startData = startValue;
    this.endData = endValue;
}
public ForkJoinApplication(int[] data) {
    this(data, 0, data.length);
}
//recursive method which forks all small work units and then, joins them
@Override
protected Integer compute() {
final int length = endData - startData;
if (length < SEQUENTIAL_THRESHOLD) {
return computeDirectly();
}
final int midValue = length / 2;
final ForkJoinApplication leftValues = new
        ForkJoinApplication(data, startData, startData + midValue);
//forks all the small work units
leftValues.fork();
final ForkJoinApplication rightValues = new
        ForkJoinApplication(data, startData + midValue, endData);
//joins them all again using the join method
return Math.max(rightValues.compute(), leftValues.join());
}
private Integer computeDirectly() {
System.out.println(Thread.currentThread() +" computing: " + startData
+" to " + endData);
int max = Integer.MIN_VALUE;
for (int i = startData; i < endData; i++) {
if (data[i] > max) {
max = data[i];
}
}
return max;
}
public static void main(String[] args) {
// create a random object value
final int[] value = new int[20];
final Random randObj = new Random();
for (int i = 0; i < value.length; i++) {
value[i] = randObj.nextInt(100);
}
}

```

```

// submit the task to the pool
final ForkJoinPool pool = new ForkJoinPool(4);
final ForkJoinApplication maxFindObj = new
    ForkJoinApplication(value);
//invokes the compute method
System.out.println("Maximum value after computing is: "+
    pool.invoke(maxFindObj));
}
}

```

Figure 5.7 displays the output.

```

Output - ThreadApplication (run) % Inspector Terminal
run:
Thread[ForkJoinPool-1-worker-2,5,main] computing: 7 to 10
Thread[ForkJoinPool-1-worker-3,5,main] computing: 2 to 5
Thread[ForkJoinPool-1-worker-4,5,main] computing: 12 to 15
Thread[ForkJoinPool-1-worker-1,5,main] computing: 17 to 20
Thread[ForkJoinPool-1-worker-3,5,main] computing: 0 to 2
Thread[ForkJoinPool-1-worker-4,5,main] computing: 10 to 12
Thread[ForkJoinPool-1-worker-3,5,main] computing: 5 to 7
Thread[ForkJoinPool-1-worker-1,5,main] computing: 15 to 17
Maximum value after computing is: 91
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 5.7: Fork/Join Functionality - Output

5.8 StackWalker API

Before looking into stack walking and traversing stack frames, it is crucial to understand what is a stack frame. Each JVM thread has a private JVM stack that is created when a thread is created. By now you may be aware that a stack is a linear data structure in memory based on Last In First Out (LIFO) principle. Whenever a method is called, a new stack frame is created and pushed to the top of the stack. When the method call completes, the stack frame is destroyed (that is, 'popped' from the stack).

A stack for a particular thread can be termed as runtime stack. Every method call performed by that thread is stored in the corresponding runtime stack. Each stack frame on the stack has its own array of local variables, operand stack, and other data. In a given thread, at any point there is only one stack frame active. Active stack frame is called the current stack frame, which method is called the current method. The method defined in the current class is called the current class.

A stack trace is a representation of a call stack at a specific point in time, with each element representing a method call. This is extremely useful in exception handling to identify the root cause of the failure.

In earlier versions, before Java 9, extracting stack trace information was cumbersome and tedious because the process would often capture a snapshot of the entire stack, which was not necessary.

Following are the disadvantages of using `getStackTrace()` to get stack trace information:

- Performance issues occur when accessing the current thread stack frame out of the recursive call at particular stack depth. There are scenarios where developers are interested in only a few stack elements. In such cases, these methods are inconvenient.
- Developers may end up accessing useless frames that are not necessary for the development.
- Sometimes information is missing because of returning incomplete stack frames in VM implementation.
- Accessing Class information is difficult with the class instance.

Java 9 introduced the StackWalker API which provides capabilities to walk through the stack. StackWalker API avoids capturing the entire stack trace. Hence, it enhances easy filtering as well as frame filtering of the stack. With this API, class information can be accessed entirely because stack trace contains actual references to `class<?>` instances. It is multithreaded as multiple threads are shared among a single `StackWalker` object. The `StackWalker` class is the main component of StackWalker API. It returns the information of the stack frame objects that `StackWalker` determines. Stack Frames offer class names and method names, but not class references.

Features and benefits of this API are as follows:

- It provides developers a mechanism to filter/skip classes which in turn gives flexibility to process specific classes.
- It offers a way to load only certain frames (for example, one can load only five frames). This can improve performance.
- One can directly get an instance of the declaring class without using the method `java.lang.SecurityManager.getClassContext()`.
- It offers developers a way to understand application behavior more easily.
- It is thread-safe and enables multiple threads to share a single `StackWalker` instance for accessing their respective stacks.
- Accessing times are same for the current stack top access of different stack depths (1,10,100,1000).
- It filters certain classes, thereby trimming stack frames by passing frames to `filter()` and `limit()` methods.
- `getInstance()` method is called to access certain frames directly, thereby ensuring more flexible access to limited frames.

The `StackWalker` class has several methods that can be used to capture the information from the stack. These include:

- **forEach:** Performs given action on each element of `StackFrame` stream of the current thread. Can traverse from the top towards bottom of the stack trace of the current thread by applying the method on each frame.

Syntax

```
public void forEach(Customer<? super StackWalker.StackFrame>
action)
```

- **getInstance()** : Returns a StackWalker instance. There are three overloads for this static method.
- **walk()** : Applies given function to the stream of StackFrames for the current thread. The walk() method of StackWalker class easily traverses through the stack to improve frame filtering. Stack elements are arranged from where the stack is generated to the bottom of the stream. The stream is closed when walk() method returns. It creates and returns the sequential stream of the stack frames by applying a function to the stream. Once the method returns, the stream is closed.

Syntax

```
public <T> T walk(Function<? super
Stream<StackWalker.StackFrame>, ? extends T> function)
```

- **StackWalker.getCallerClass()** : Returns the object of the class of the caller. This method filters reflection frames, method handle, and hidden frames.

Syntax

```
public Class<?> getCallerClass()
```

The StackWalker.StackFrame interface is a static nested interface of StackWalker. An object of this interface represents method invocation returned by StackWalker. It has methods for accessing stack frame's information such as getDeclaringClass(), getLineNumber(), and so on. The StackWalker.Option enumeration configures the stack frame information obtained by the stackwalker when an instance is created via StackWalker.getInstance(). It has valueOf(String name) and values() methods.

The basic StackWalker call can be as follows:

```
StackWalker stackWalker = StackWalker.getInstance();
```

Code Snippet 15 demonstrates a simple example using the StackWalker API classes and interfaces.

Code Snippet 15:

```
import java.lang.StackWalker.StackFrame;
import java.util.*;
import java.util.stream.*;
public class StackWalkingDemo {
public static void main(String args[]) {
new StackWalkingDemo().walk();
}
private void walk() { new Walker1().walk(); }
private class Walker1 {
public void walk() {
new Walker2().walk();
}
}
```

```

private class Walker2 {
    public void walk() {
        FirstMethod();
    }
    void FirstMethod() {
        SecondMethod();
    }
    void SecondMethod() {
        StackWalker stackWalker =
            StackWalker.getInstance(Set.of(StackWalker.
                Option.RETAIN_CLASS_REFERENCE,
                StackWalker.Option.SHOW_HIDDEN_FRAMES), 16);
        Stream<StackFrame> stackStream = StackWalker.getInstance().walk(f
-> f);
        List<String> stacks = walkAllStackframes();
        System.out.println("Number of StackFrames: " + stacks.size());
        System.out.println("*Walk through all StackFrames*");
        System.out.println(stacks);
        System.out.println("*Skip some StackFrames*");
        List<String> stacksAfterSkip = walkSomeStackframes(2);
        System.out.println("Number of StackFrames remaining: " +
            stacksAfterSkip.size());
        System.out.println(stacksAfterSkip);
    }

    private List<String> walkAllStackframes() {
        return StackWalker.getInstance().walk(s -> s.map(frame -> "\n" +
            frame.getClassName() + "/" + frame.getMethodName()) .
            collect(Collectors.toList()));
    }
    private List<String> walkSomeStackframes(int numberOfframes) {
        return StackWalker.getInstance().walk(s -> s.map(frame -> "\n" +
            frame.getClassName() + "/" + frame.getMethodName()) .
            skip(numberOfframes).collect(Collectors.toList()));
    }
}
}

```

In the code, there are several method calls within several classes. Each method call is saved on to a stack frame internally by the JVM. In this code, we have printed all stack frames and skipped some stack frames in the display by using StackWalker API.

The method `walkAllStackframes()` returns a list of stack frames which are then stored in a `List` instance named `stacks`. The code then prints the size of this list which represents the number of stack frames currently existing. The code prints the stack frames one by one. To do this, it makes use of `StackWalker.getInstance().walk()` method, along with methods of `StackFrame` such as `getClassName()` and `getMethodName()`.

Further, in the code, we skip two frames and print the rest. We also print the number of frames remaining after skipping a specified number.

The output of Code Snippet 15 is as follows:

```
Number of StackFrame: 7
*Walk through all StackFrames*
[
StackWalkingDemo$Walker2/walkAllStackframes,
StackWalkingDemo$Walker2/SecondMethod,
StackWalkingDemo$Walker2/FirstMethod,
StackWalkingDemo$Walker2/walk, StackWalkingDemo$Walker1/walk,
StackWalkingDemo/walk, StackWalkingDemo/main]
*Skip some StackFrames*
Number of StackFrames remaining: 5
[
StackWalkingDemo$Walker2/FirstMethod,
StackWalkingDemo$Walker2/walk, StackWalkingDemo$Walker1/walk,
StackWalkingDemo/walk, StackWalkingDemo/main]
Walking through the stack frames in this manner can help debugging greatly.
```

5.9 Structured Concurrency for JDK 20

Structured Concurrency is a programming paradigm aimed at making concurrent programming more manageable and less error-prone by providing better tools and abstractions for managing concurrent tasks. It is designed to address some of the challenges and complexities associated with traditional approaches to concurrency, such as using threads or low-level synchronization primitives.

Here are some key concepts of structured concurrency:

Task Scopes:

In structured concurrency, tasks are organized within a hierarchical structure called a task scope. A task scope is a logical grouping of tasks that share a common lifecycle. Tasks within the same scope are created, executed, and terminated together. This helps avoid issues such as orphaned threads or resources due to premature termination. The scope helps to enforce certain rules and guarantees about how tasks are executed and how their lifecycles are managed.

Task Lifecycle:

Structured concurrency introduces a clear lifecycle for tasks. Tasks are typically created, started, and awaited within a well-defined scope. This ensures that tasks are properly managed and that the scope is not exited before all tasks within it have completed.

Cancellation and Error Propagation:

Structured concurrency provides mechanisms for propagating cancellation and errors throughout the task scope. When a task encounters an error or is cancelled, the scope ensures that all related tasks are also appropriately handled. This helps prevent silent failures and resource leaks.

No Shared State by Default:

Structured concurrency encourages the use of communication mechanisms (such as message passing) rather than shared mutable state between tasks. This can help mitigate many of the common issues related to race conditions and data synchronization.

Higher-Level Abstractions:

Instead of dealing directly with threads and low-level synchronization constructs, structured concurrency often provides higher-level abstractions that make it easier to express concurrent logic.

5.10 Radix Sort with Foreign APIs

Implementing radix sort using foreign APIs in Java involves using the Java Native Interface (JNI) to call functions from a native library that provides the radix sort algorithm. Code Snippets 16a and 16b demonstrate high-level overview of the steps involved in.

- 1. Choose a Foreign Language and Library:** Select a foreign programming language and a suitable library that implements radix sort. For this example, let us assume you are using C and the C Standard Library.
- 2. Write the Foreign Implementation:** Implement the radix sort algorithm in C. Create a C source file (example:- radixsort.c) containing the radix sort function.

Code Snippet 16a: C code

```
// radixsort.c
#include <jni.h>
JNIEXPORT void JNICALL Java_com_example_RadixSort_radixSort(JNIEnv *env, jobject obj, jintArray arr, jint size) { // Convert jintArray to C array
    jint *arrData = (*env)->GetIntArrayElements(env, arr, NULL);
    // Radix sort implementation (replace with actual implementation)
    // ...
    // Release C array
    (*env)->ReleaseIntArrayElements(env, arr, arrData, 0);
}
```

- 3. Create a JNI Header File:** Generate a JNI header file for the C code using the javac command with the -h flag. This will generate a header file (for example, com_example_RadixSort.h) that you require to include in your C code. Compile the program in this manner:

```
javac -h . RadixSort.java
```

- 4. Compile the C Code:** Compile the C code along with the JNI header and create a shared library (DLL on Windows).

```
gcc -shared -o libradixsort.so -I"${JAVA_HOME}/include" -I"${JAVA_HOME}/include/linux" radixsort.c
```

- 5. Load the Shared Library in Java:** In your Java code, load the shared library using System.loadLibrary().

Code Snippet 16b: Java code

```
public class RadixSort {
    static {
        System.loadLibrary("radixsort");
    }
    public static native void radixSort(int[] arr, int size);
```

```

public static void main(String[] args) {
    int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
    radixSort(arr, arr.length);
    for (int num : arr) {
        System.out.print(num + " ");
    }
}
}

```

This is a simplified example to illustrate the process of implementing radix sort with foreign APIs using JNI. In practice, you would require to implement the actual radix sort algorithm in C, handle memory management, data conversion between Java and C, and error checking.

Be aware that JNI involves challenges such as memory leaks, potential crashes, and platform dependencies. It is important to have a good understanding of both Java and C programming and carefully manage resources to ensure correct and reliable behavior.

5.11 Off-heap and on-heap

In Java, off-heap and on-heap refer to two different memory areas used for storing objects and data during the execution of a Java program. These terms are relevant when discussing memory management and can have implications for performance, garbage collection, and overall system behavior.

On-Heap Memory

On-heap memory refers to the memory allocated within the Java Virtual Machine (JVM) heap. The JVM heap is the primary area where Java objects are created and reside. When you use constructs new `SomeObject()`, the memory for the object is allocated on the heap.

On-heap memory management is handled by the JVM's garbage collector, which automatically reclaims memory from objects that are no longer reachable, freeing up space for new objects. The JVM is responsible for managing the allocation and deallocation of memory on the heap.

Advantages of on-heap memory:

- Automatic memory management through garbage collection.
- Simplified memory allocation and deallocation.

Off-Heap Memory

Off-heap memory refers to memory that is allocated outside the JVM heap. This memory is managed directly by the operating system or native libraries, and Java objects stored in off-heap memory are not subject to the JVM's automatic garbage collection.

Off-heap memory is often used for specific purposes, such as caching, storing large amounts of data, or when you require more control over memory management. It is particularly useful when you want to avoid potential performance issues or long garbage collection pauses associated with a large heap.

Advantages of off-heap memory:

- Reduced impact on garbage collection, leading to more predictable performance.
- Can store large datasets that might not fit in the JVM heap.
- Greater control over memory allocation and deallocation.

5.12 Summary

- Multithreading is nothing, but running of several threads in a single application.
- The `isAlive()` method tests whether the thread is in runnable, running, or terminated state.
- The `join()` method forces a running thread to wait until another thread completes its task.
- Race condition can be avoided by using synchronized block.
- The `wait()` method sends the running thread out of the lock or monitor to wait.
- The `notify()` method instructs a waiting thread to get in to the lock of the object for which it has been waiting.
- Deadlock describes a situation where two or more threads are blocked forever, waiting for each to release a resource.
- Java 9 introduced the `StackWalker API` which provides capabilities to walk through the stack and helps to trace root causes of exceptions.
- The `StackWalker` class is the main component of `StackWalker API`.

5.13 Check Your Progress

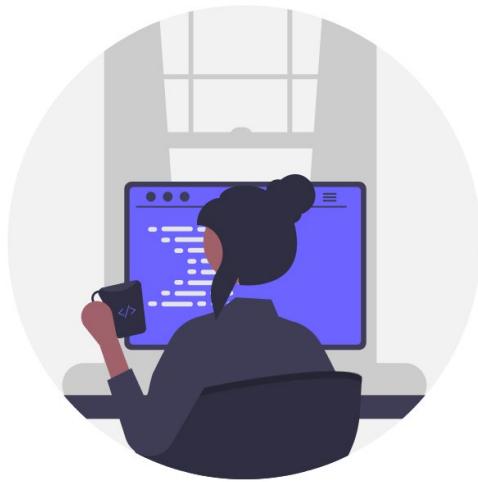
1. Which one of the following can cause race condition?
 - A. When two or more threads share the same data between them
 - B. The use of synchronized blocks
 - C. Releasing the intrinsic lock
 - D. Interleaving the atomic actions
2. Which of the following statements are true for multithreading?
 - A. Multithreading requires less overhead
 - B. In a multithreaded environment, two or more processes run concurrently
 - C. Multithreading is a specialized form of multitasking
 - D. In a multithreaded program, two or more threads can run concurrently
3. Which one of the following acts as the traffic signal system in a program?
 - A. Locks
 - B. Race condition
 - C. The wait-notify mechanism
 - D. Atomic variables
4. Which one of the following brings the execution of the program to a halt?
 - A. Thread pools
 - B. Executors
 - C. Race condition
 - D. Deadlock
5. Which of the following options do the executor implementations in `java.util.concurrent` use?
 - A. ExecutorService interface
 - B. Runnable interface
 - C. Collections interface
 - D. ScheduledExecutorService interface
6. A _____ is a piece of code that is guarded by a mutual-exclusion program.
 - A. Mutex
 - B. Monitor
 - C. The synchronized keyword
 - D. Lock

5.13.1 Answers

1. D
2. A, C, D
3. C
4. D
5. A, D
6. B

Try It Yourself

1. Write a Java program that creates two threads. One thread prints odd numbers and the other thread prints even numbers. Implement synchronization to ensure proper alternation between the two threads.
2. Implement a simple producer-consumer problem using multithreading. Create a shared buffer, where the producer thread adds items, and the consumer thread removes items. Use proper synchronization mechanisms to prevent race conditions.
3. Design a Java program that simulates a bank account management system. Implement two types of transactions: deposits and withdrawals. Ensure thread safety using synchronized methods or blocks to prevent concurrent access issues.



Session 6 JDBC API

Welcome to the Session, **JDBC API**.

This session introduces you to JDBC, a software API for database connectivity, explores JDBC, and describes its architecture. It also discusses how to use JDBC and its drivers to develop a database application. The session also describes how to connect to and retrieve data from an SQL Server database. Finally, the session explains about database meta data information.

In this Session, you will learn to:

- Explain basics of JDBC
- Elaborate on JDBC architecture
- Identify and describe different types of processing models
- Outline JDBC Driver Types
- List the steps involved in JDBC application development
- Explain Database Meta Information
- Elaborate on parameterized queries in JDBC API

6.1 Introduction

A database contains data that is in an organized form. Client/server applications make extensive use of database programming. Typical activities involved in a database application involve opening a connection, communicating with a database, executing Structured Query Language (SQL) statements, and retrieving query results.

Java has established itself as one of the prime backbones of enterprise computing.

The core of Java enterprise applications depends on Database Management Systems (DBMS), which acts as the repository for an enterprise's data. Hence, to build such applications, these databases in the repository required to be accessed. To connect the Java applications with the databases, Application Programming Interface (API) software for database connectivity is used. This software API is a collection of application libraries and database drivers, whose implementation is

independent of programming languages, database systems, and operating systems.

Open DataBase Connectivity (ODBC) and JDBC are two widely used APIs for such activities.

Note: JDBC is not an acronym, though it is often mistaken to be 'Java Database Connectivity'. It is actually the brand name of the product.

Figure 6.1 displays the concept of database connectivity.

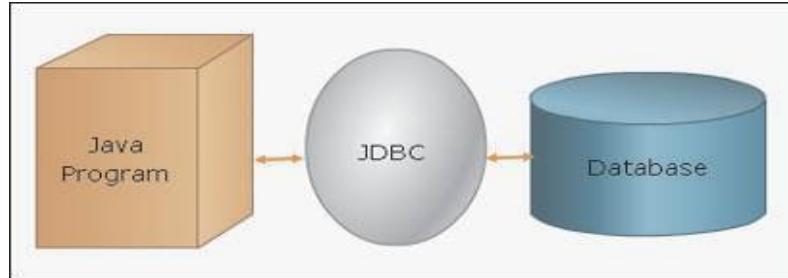


Figure 6.1: Database Connectivity

6.2 ODBC

ODBC is an API provided by Microsoft for accessing the database. It uses SQL as its database language. It provides functions to insert, modify, and delete data and obtain information from the database. Figure 6.2 displays ODBC connection.

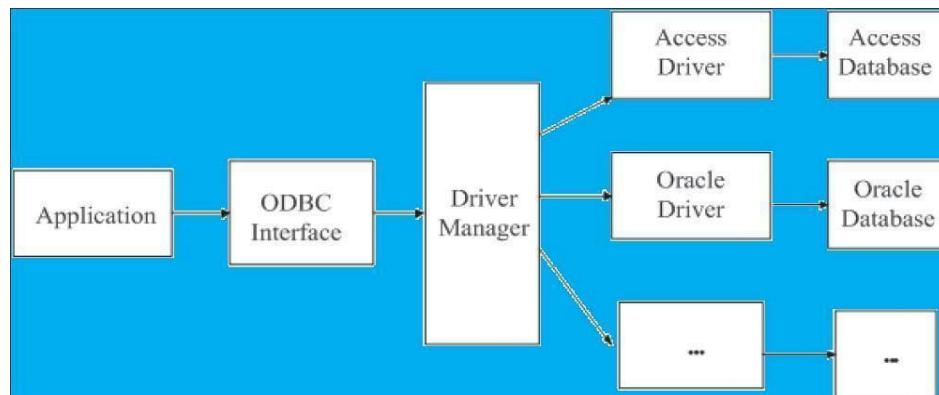


Figure 6.2: ODBC Connection

The application can be a GUI program written in Java, VC++, or any other software. The application makes use of ODBC to connect with the databases. The driver manager is part of Microsoft ODBC API and is used to manage various drivers in the system including loading. In cases where the application uses multiple databases, it is the function of the driver manager to make sure that the function calls get diverted to the correct DBMS. The driver is the actual software component that knows about the database. Drivers are software specific such as Microsoft Access Driver, Oracle Driver, and so on.

6.2.1 Definitions of JDBC

JDBC is the Java-based API, which provides a set of classes and interfaces written in Java to access and manipulate different kinds of databases. JDBC API is a Java API provided by Sun. The JDBC API has a set of classes and interfaces used for accessing tabular data. These classes and interfaces are

written in Java programming language and provides a standard API for database developers. To access data quickly and efficiently from databases Java applications uses JDBC. The advantage of using JDBC API is that an application can access any database and run on any platform having JVM. In other words, a Java application can write a program using JDBC API, and the SQL statement can access any database.

The combination of JDBC API and Java platform offers the advantage of accessing any type of data source and flexibility of running on any platform which supports JVM. For a developer, it is not necessary to write separate programs to access different databases such as SQL Server, Oracle, or IBM DB2. Instead, a single program with the JDBC implementation can send SQL or other statements to the suitable data source or database. The three tasks that can be performed by using JDBC drivers are as follows: establish a connection with the data source, send queries, and update statements to the data source and finally process the results.

➤ Product Components of JDBC

The four product components of JDBC are as follows:

JDBC API	JDBC Driver Manager	JDBC Test Suite	JDBC-ODBC Bridge
JDBC API is a part of Java platform. JDBC 4.0 APIs are included in two packages which are included in Java Standard Edition and Java Enterprise Edition. The two packages are <code>java.sql</code> and <code>javax.sql</code> . JDBC API allows access to the relational database from the Java programming language. Thus, JDBC API allows applications to execute SQL statements, retrieve results, and make changes to the underlying database.	A Java application is connected to a JDBC driver using an object of the <code>DriverManager</code> class. The <code>DriverManager</code> class is the backbone of the JDBC architecture. The tasks performed by the <code>DriverManager</code> class are to first locate the driver for a specific database and then, process the initialization calls for JDBC.	JDBC driver test suite helps to determine that JDBC drivers will run the program.	JDBC access is provided using ODBC drivers.

6.3 JDBC Architecture

JDBC is one of the core parts of Java platform and is included in the distribution of JDK. The main task of JDBC API is to allow the developers to execute the SQL statements in an independent manner irrespective of the underlying database. The classes and interfaces of JDBC API are used to represent objects of database connections, SQL statements, Result sets, Database metadata, Prepared statements, Callable statements, and so on.

To provide connectivity to heterogeneous databases, the driver manager and database specific drivers are used by JDBC API. The driver manager ensures that correct drivers are used to access the database. Multiple drivers connected to access different data source are supported by the driver manager. Figure 6.3 shows the location of the driver manager with respect to JDBC drivers and applications.

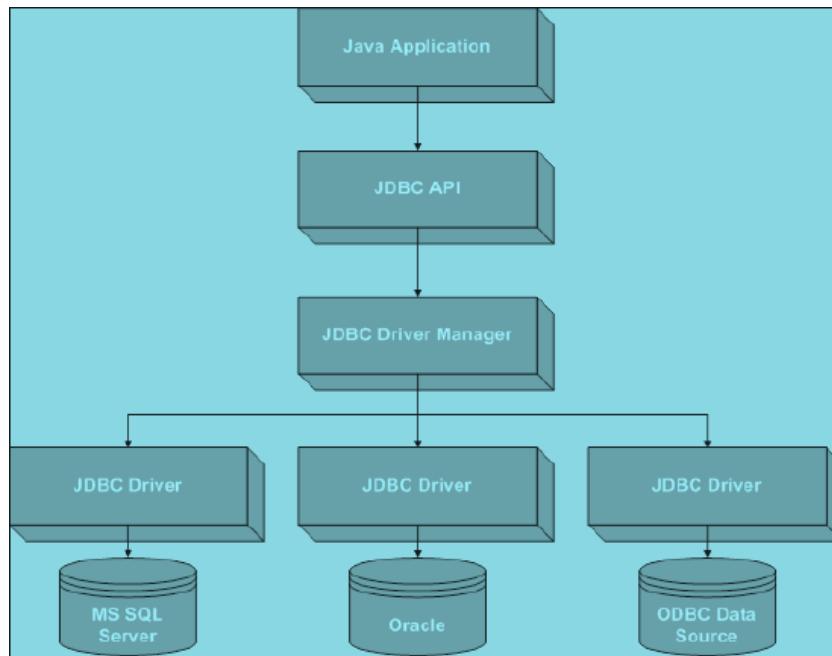


Figure 6.3: JDBC Architecture

6.3.1 Advantages of JDBC

Some of the advantages of JDBC are as follows:

Continued usage of existing data

JDBC enables enterprise applications to continue using existing data even if the data is stored on different database management systems.

Vendor independent

The combination of the Java API and the JDBC API makes the databases transferable from one vendor to another without modifications in the application code.

Platform independent

JDBC is usually used to connect a user application to a 'behind the scenes' database, no matter of what database management software is used to control the database. In this fashion, JDBC is cross-platform or platform independent.

Ease of use

With JDBC, the complexity of connecting a user program to a 'behind the scenes' database is hidden, and makes it easy to deploy and economical to maintain.

6.3.2 Two-Tier Data Processing Model

The JDBC API supports two-tier as well as three-tier data processing models for accessing database. In a two-tier client/server system, the client communicates directly to the database server without the help of any middle-ware technologies or another server. In a two-tier JDBC environment, the Java application is the client and DBMS is the database server. The typical implementation of a two-tier model involves the use of JDBC API to translate and send the client request to the database.

The database may be located on the same or another machine in the network. The results are delivered back to the client again through the JDBC API. Figure 6.4 displays the two-tier data processing model.

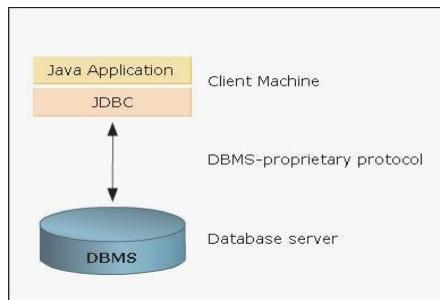


Figure 6.4: Two-Tier Data Processing Model

6.3.3 Three-Tier Data Processing Model

In a three-tier model, a 'middle tier' of services, a third server is employed to send the client request to the database server. This middle-tier helps in separating the database server from the Web server. The involvement of this third server or proxy server enhances the security by passing all the requests to the database server through the proxy server. The database server processes the requests and sends back the results to the middle tier (proxy server), which again sends it to the client. Figure 6.5 displays the three-tier data processing model.

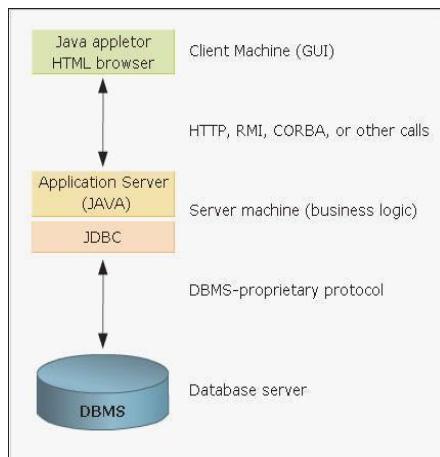


Figure 6.5: Three-Tier Data Processing Model

6.3.4 JDBC API

JDBC API is a collection of specifications that defines the way how database and the applications communicate with each other. The core of JDBC API is based on Java, so, it is used as the common platform for building the middle-tier of three-tier architecture.

Hence, JDBC API being a middle-tier, it defines how a connection is opened between an application and database; how requests are communicated to a database, the SQL queries are executed, and the query results retrieved. JDBC achieves these targets through a set of Java interfaces, implemented separately by a set of classes for a specific database engine known as JDBC driver.

Figure 6.6 displays the JDBC-API.

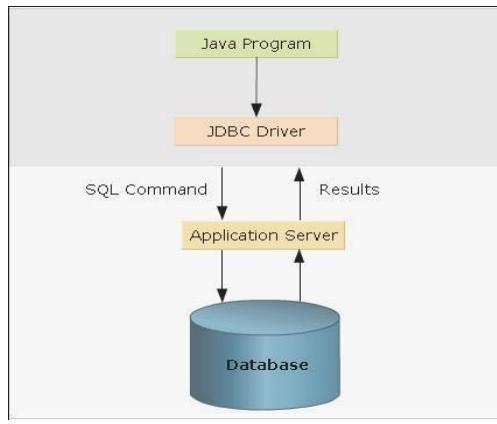


Figure 6.6: JDBC-API

6.4 JDBC Driver Types

The JDBC driver is the base of JDBC API and is responsible for ensuring that an application has a consistent and uniform access to any database. The driver converts the client request to a database understandable, native format and then, presents it to the database. The response is also handled by the JDBC driver, and gets converted to the Java format and presented to the client.

Four types of drivers and a brief description of their basic properties are listed in Table 6.1.

Driver Type	Driver Name	Description
Type I	ODBC-JDBC Bridge	Translates JDBC calls into ODBC calls
Type II	Native API-Java/ Partly Java	Translates JDBC calls into database-specific calls or native calls
Type III	JDBC Network-All Java	Maps JDBC calls to the underlying 'network' protocol, which in turn calls native methods on the server
Type IV	Native Protocol-All Java	Directly calls RDBMS from the client machine

Table 6.1: Types Of Drivers

6.4.1 JDBC Type 1 Driver

The Type 1 driver is a Java software bridge product, also known as JDBC-ODBC bridge plus ODBC driver. Figure 6.7 displays the JDBC type 1 driver.

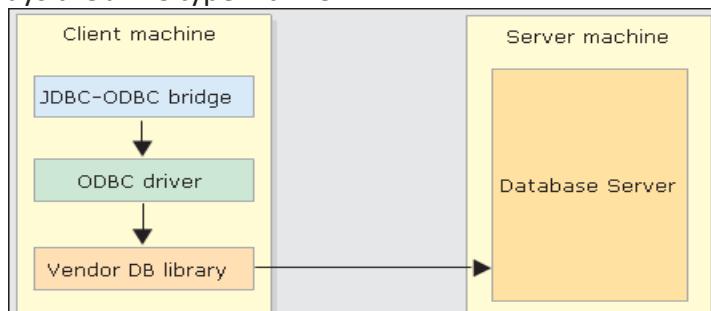


Figure 6.7: JDBC Type 1 Driver

Features	The Type 1 drivers use a bridging technology that provides JDBC access via ODBC drivers. This establishes a link between JDBC API and ODBC API. The ODBC API is in turn implemented to get the actual access to the database via the standard ODBC drivers. Client machine must install native ODBC libraries, drivers, and required support files, and in most of the cases, the database client codes. This kind of driver is appropriate for an enterprise network, where installing client is not a problem.
Advantages	The Type 1 drivers are written to allow access to various databases through pre-existing ODBC drivers. In some cases, they are the only option to databases such as MS-Access or Microsoft SQL Server for having ODBC native call interface.
Disadvantages	The Type 1 driver does not hold good for applications that do not support software installations on client machines. The native ODBC libraries and the database client codes must reside on the server, which in turn reduces the performance.

6.4.2 JDBC Type 2 Driver

The Type 2 driver is also known as Native-API partly Java driver. Figure 6.8 displays the JDBC type 2 driver.

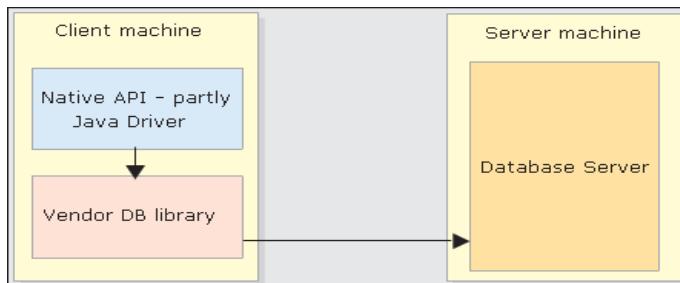


Figure 6.8: JDBC Type 2 Driver

➤ Features

The Type 2 driver comprises the Java code that converts JDBC calls into calls on a local database API for Oracle, Sybase, DB2, or any other type of DBMS. This implies that the driver calls the native methods of the individual database vendors to get the database access.

This kind of driver basically comes with the database vendors to interpret JDBC calls to the database-specific native call interface, for example, Oracle provides OCI driver. The Type 2 driver also required native database-specific client libraries to be installed and configured on the client machine such as that of Type 1 drivers.

➤ Advantages

The Type 2 driver yields better performance than that of Type 1 driver. Type 2 drivers are generally faster than Type 1 drivers as the calls get converted to database-specific calls.

➤ **Disadvantages**

The Type 2 driver does not support applications that do not allow software installations on client machines as it requires native database codes to be configured on client machines. These database specific native code libraries must reside on the server, in turn reducing the performance.

6.4.3 JDBC Type 3 Driver

The Type 3 drivers are also known as JDBC-Net pure Java driver. Figure 6.9 displays the JDBC type 3 driver.

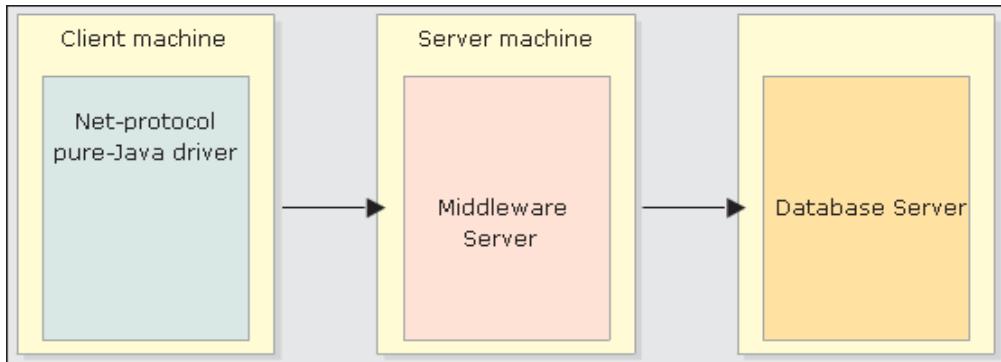


Figure 6.9: JDBC Type 3 Driver

➤ **Features**

The Type 3 driver is a pure Java driver that converts the JDBC calls into a DBMS-independent network protocol, which is again translated to database-specific calls by a middle-tier server. This driver does not require any database-specific native libraries to be installed on the client machines. The Web-based applications should preferably implement Type 3 drivers as this driver can be deployed over the Internet without installing a client.

➤ **Advantages**

The Type 3 driver is the most flexible type as it does not require any software or native services to be installed on client machine. It provides a high degree of adaptability to change and control underlying database without modifying the client-side driver.

➤ **Disadvantages**

Database-specific code must be executed in the middle-tier server. As it supports Web-based applications, it must implement additional security such as access through firewalls.

6.4.4 JDBC Type 4 Driver

The Type 4 drivers are also known as Native-protocol pure Java driver or Java to Database Protocol.

Figure 6.10 displays the JDBC type 4 driver.

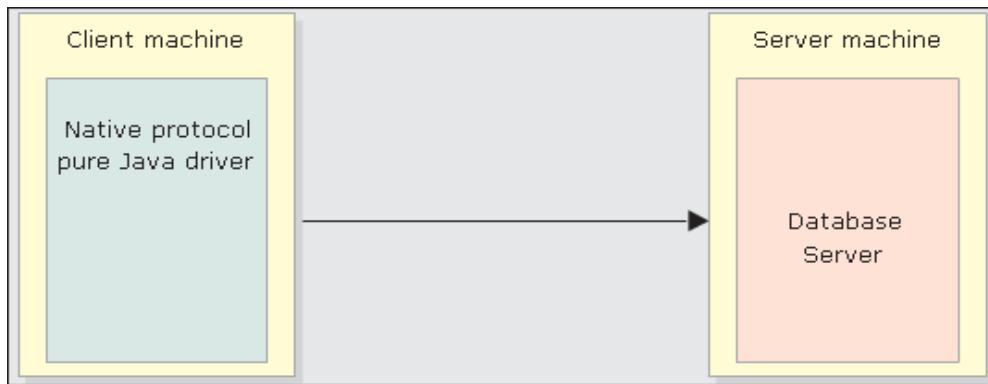


Figure 6.10: JDBC Type 4 Driver

➤ Features

Type 4 drivers are pure Java drivers that convert JDBC calls into the network protocol that communicates directly with the database. This links the client call directly with the DBMS server and provides a practical solution for accessing Intranet. In most cases, the drivers are provided by the database vendors. These drivers also do not require any database-specific native libraries to be configured on client machine and can be deployed on the Web without installing a client, as required for Type 3 drivers.

➤ Advantages

Type 4 drivers communicate directly with the database engine using Java sockets, rather than through middleware or a native library. This is the reason that these drivers are the fastest JDBC drivers available. No additional software such as native library is required for installation on clients.

➤ Disadvantages

The only drawback in Type 4 drivers is that they are database-specific. Hence, if in case, the back-end database changes, the application developer may require to purchase and deploy a new Type 4 driver specific to the new database.

Type 1 driver is helpful for prototyping and not for production. Type 3 driver adds security, caching, and connection control. Type 3 and Type 4 driver required not be pre-installed and are portable.

6.5 `java.sql` Package

JDBC API defines a set of interfaces and classes to communicate with the database. They are contained in the `java.sql` package. The API has different framework where the drivers can be installed dynamically to access different data sources. The JDBC API not only passes SQL statements to a database but it also provides facility for reading and writing data from any data source with a tabular format. Some of the interfaces in this package have been summarized in Table 6.2.

Interface	Description
Connection	This is used to maintain and monitor database sessions. Data access can also be controlled using the transaction locking mechanism.

Interface	Description
DatabaseMetaData	This interface provides database information such as the version number, names of the tables, and functions supported. It also has methods that help in discovering database information such as the current connection, tables available, schemas, and catalogues.
Driver	This interface is used to create Connection objects.
PreparedStatement	This is used to execute pre-compiled SQL statements.
ResultSet	This interface provides methods for retrieving data returned by a SQL statement.
ResultSetMetaData	This interface is used to collect the meta data information associated with the last ResultSet object.
Statement	It is used to execute SQL statements and retrieve data into the ResultSet.

Table 6.2: Interfaces in java.sql Package

Some of the classes included in the `java.sql` package are shown in Table 6.3.

Class Name	Description
Date	This class contains methods for performing conversion of SQL date formats to Java Date formats.
DriverManager	This class is used to handle loading and unloading of drivers and establish connection with the database.
DriverPropertyInfo	The methods in this class are used to retrieve or insert driver properties.
Time	This class provides formatting and parsing operations for time values.

Table 6.3: Classes in java.sql Package

The exceptions defined by the `java.sql` package are listed in Table 6.4.

Exception	Description
DataTruncation	This exception is raised when a data value is truncated.
SQLException	This exception provides information on database access errors or other errors.
SQLWarning	This exception provides information on database access warnings.
BatchUpdateException	This exception is raised when an error occurs during batch update operation.

Table 6.4: Exceptions in java.sql Package

6.5.1 Steps to Develop a JDBC Application

The process of accessing a database and processing queries via JDBC involves these basic steps:

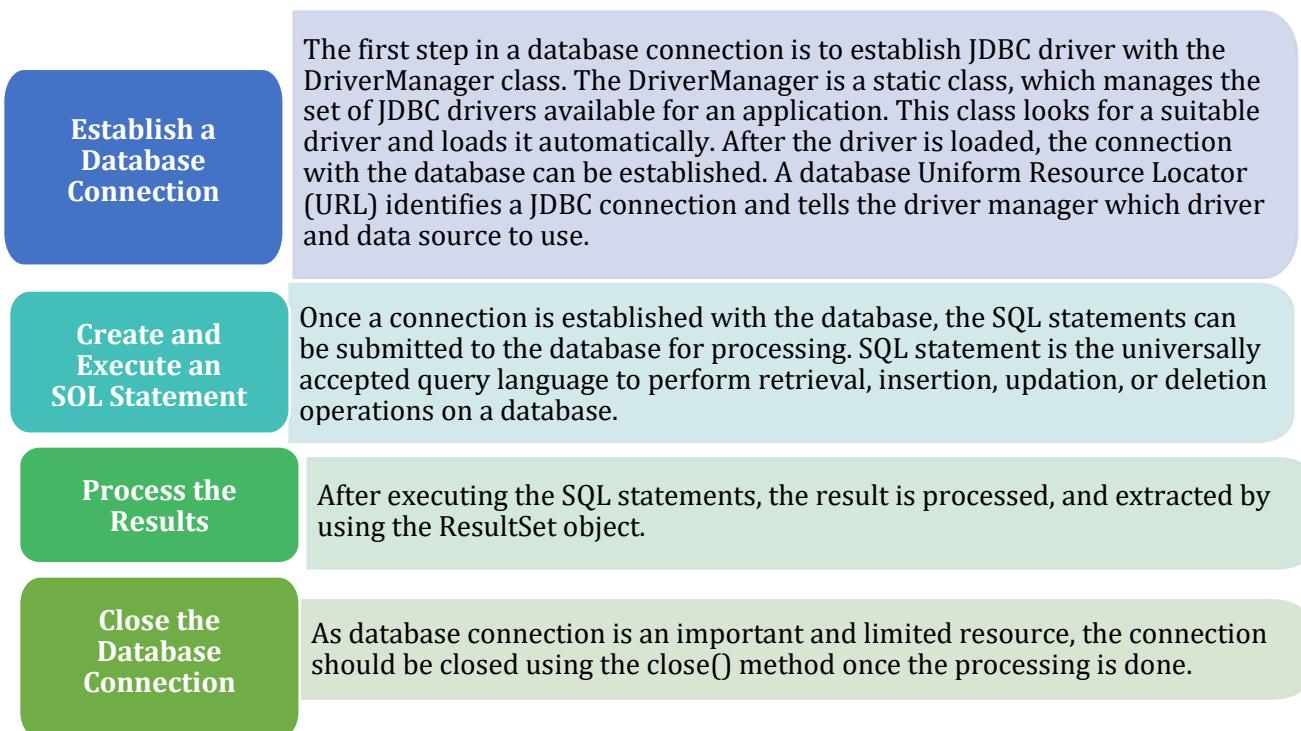


Figure 6.11 displays the steps to develop a JDBC application.

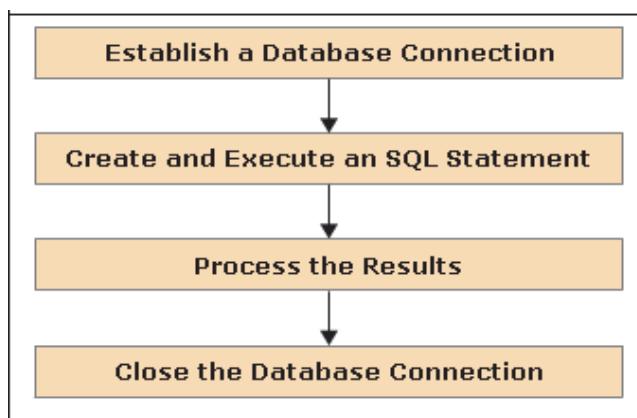


Figure 6.11: Steps to Develop a JDBC Application

6.5.2 Establishing a Connection

For Java 8 onwards, the `DriverManager` class is used to establish the database connection. This is done by using the `DriverManager.getConnection()` method, which checks all available drivers for the eligibility to make a connection. This also checks for a driver that recognizes the URL, sent by the client. Establishing a connection is achieved by the use of following elements:

➤ **Connection URL**

The connection URL specifies necessary information that will be required for connecting to a

particular database.

Syntax

protocol:<subprotocol>:<subname>

The component protocol in the JDBC URL identifies the protocol name for accessing the database.

The second component subprotocol recognizes the underlying database source. The last component subname identifies the database name.

A sample connection URL is as follows:

`jdbc:mysql://localhost:3306/test`

Here, connection is being made to a MySQL database server present on the local system (accessed via localhost) through port 3306 and the database name is **test**.

Another example is as follows:

`jdbc:sqlserver://127.0.0.1:1433`

Here, connection is being made to an SQL Server database server present on the local system (accessed via 127.0.01) through default port 1433. The database name is not specified here so the default database will be used.

Developers must download and add appropriate JAR files from the Oracle Website in order for database handling code to execute properly.

To do this, visit the site (such as <https://dev.mysql.com/downloads/connector/j/?os=26>), download and extract the zip file. Then, in NetBeans, under your project, right-click **Libraries**, **Add JAR/Folder**, and then, add the extracted JAR file. Figure 6.12 shows this step.

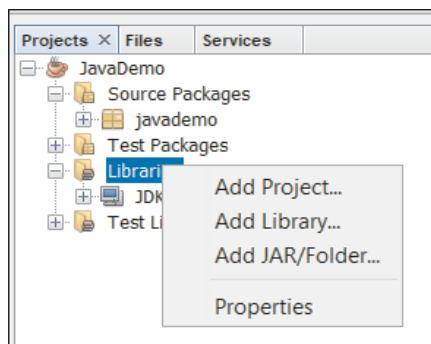


Figure 6.12: Adding MySQL Connector JAR File for Database Connectivity

Similarly, for SQL Server, one can add a relevant jar file, such as **mssql-jdbc-12.4.1.jre11.jar**.

If you create a Java Maven project in Apache NetBeans, there is no Libraries or Add Jar option and there is no necessity to download any jar file. Instead, you have to add a dependency using the **pom.xml** file in your project.

For example, to connect to MySQL database, the dependency to be added in pom.xml is as follows:

```
<dependencies>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-j</artifactId>
<version>8.1.0</version>
</dependency>
</dependencies>
```

Here, 8.1.0 is the version of MySQL. There may be newer versions available too. Developers can visit <https://mvnrepository.com/artifact/com.mysql/mysql-connector-j> to look for the latest version.

Once the `pom.xml` file is configured with appropriate database dependency, Maven will look for those libraries in the central Maven repository. Thus, the developer does not have to explicitly add the libraries. This step is applicable only for Maven projects.

➤ `DriverManager.getConnection()`

The calling of `getConnection()` method involves a two-step process. The first step, matching the URL and Driver, is followed by connecting to the database, which is the second step.

Syntax

```
Connection cn = DriverManager.getConnection(<connection url>,
<username>, <password>);
```

The `getConnection()` method usually accepts two arguments. The first is the connection URL string of the database and the second is the set of login attributes such as username and password.

Code Snippet 1 shows use of `DriverManager.getConnection()`.

Code Snippet 1:

```
Connection cn = DriverManager.
    getConnection("jdbc:mysql://127.0.0.1:3306/test", "root", "");
```

Here, the URL used is `jdbc:mysql://127.0.0.1:3306/test`. The next string is the username `root` and finally, the password is an empty string.

Code Snippet 2 shows the complete code for an application that establishes a connection to a MySQL database named `test`.

Code Snippet 2:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
class BasicDatabaseDemo {
    public static void main(String args[]) {
```

```

try {
Connection cn = DriverManager.
getConnection("jdbc:mysql://127.0.0.1:3306/test", "root", "");
System.out.println("Connection successfully established");
} catch (SQLException ce) {
System.out.println(ce.getMessage());
}
}
}

```

Since the code may raise an exception of type `SQLException`, the statements are enclosed in a `try-catch` block.

6.5.3 Creating Statements and Queries

Once a connection is established with a database, a `Statement` object must be created for query execution. The `Statement` object is the most frequently used object to execute SQL queries that do not require any parameters to be passed.

The `Statement` object is created by using the `Connection.createStatement()` method. A `Statement` object can be classified into three categories based on the type of SQL statements sent to the database. `Statement` and `PreparedStatement` are inherited from `Statement` interface. `CallableStatement` is inherited from the `PreparedStatement` interface and executes a call to stored procedure. A `PreparedStatement` object executes a precompiled SQL statement with or without `IN` parameters.

Syntax

```
public Statement createStatement() throws SQLException  
where,
```

`Statement` is the `Statement` object that is returned by the execution of the `createStatement()` method.

Code Snippet 3 demonstrates creating a `Statement` object. Here, `cn` is assumed to be an existing `Connection` object. The code is assumed to be enclosed in a `try-catch` block.

Code Snippet 3:

```
Statement st = cn.createStatement();
```

Here, `st` is the `Statement` object, which is created and associated with a single connection object `cn`.

6.5.4 Using executeQuery() and ResultSet Objects

A `Statement` object when created has methods to perform various database operations.

The `executeQuery()` method is one of those methods that retrieves information from the database. It accepts a simple SQL SELECT statement as a parameter and returns the database rows in form of a `ResultSet` object. Hence, the database query execution is based on the following:

➤ `executeQuery()`

This method is used to execute any SQL statement with a SELECT clause, that return the result of the query as a result set. It takes the SQL query string as the parameter.

➤ **ResultSet object**

`ResultSet` objects are used to receive and store the data in the same form as it is returned from the SQL queries. The `ResultSet` object is generally created by using the `executeQuery()` method. The `ResultSet` object contains the data returned by the query as well as the methods to retrieve data.

Syntax

```
public ResultSet executeUpdate(String sql) throws SQLException  
where,
```

`sql` is a `String` object containing SQL statement to be sent to the database
`ResultSet` object is created to store the result of SQL statement

Code Snippet 4 demonstrates use of the `executeQuery()` method to select some rows.

Code Snippet 4:

```
ResultSet rs = st.executeQuery("SELECT * FROM Employees");
```

Here, `rs` is the `ResultSet` object created to store the result of the SELECT statement.

Code Snippet 5 shows a complete example with `executeQuery()` method.

Code Snippet 5:

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;  
class BasicDatabaseDemo {  
public static void main(String args[]) {  
try {  
Connection cn = DriverManager.  
getConnection("jdbc:mysql://127.0.0.1:3306/test", "root", "");  
System.out.println("Connection successfully established");  
Statement st = cn.createStatement();  
ResultSet rs = st.executeQuery("SELECT * FROM Employees");  
while(rs.next()){  
System.out.println("Employee Number : "+rs.getInt(1));  
}  
} catch (SQLException ce) {  
System.out.println(ce.getMessage());  
}  
}
```

After successfully establishing a connection, a `Statement` object is created and an SQL query is passed to it to be executed on the database table. The query retrieves all records from the `Employees` table. A `while` loop iterates through all the records using `rs.next()` method and displays employee numbers using `rs.getInt()` method.

6.5.5 Using executeUpdate() and execute() Methods

The other methods of `Statement` interface are as follows:

- `executeUpdate()`

The `executeUpdate()` method is used to execute INSERT, DELETE, UPDATE, and other SQL Data Definition Language (DDL) statements such as CREATE TABLE, DROP TABLE, and so on. The method returns an integer value indicating the row count.

Syntax

```
public int executeUpdate(String sql) throws SQLException
```

where, - `sql` is `String` object created to store the result of SQL statement and `int` is an integer value storing the number of affected rows.

- `execute()`

The `execute()` method is used to execute SQL statements that returns more than one result set. The method returns true if a result set object is generated by the SQL statements.

Syntax

```
public boolean execute (String sql) throws SQLException  
where,
```

`sql` is `String` object created to store the result of SQL statement.

6.5.6 Creating Parameterized Queries

The `PreparedStatement` object implements the execution of dynamic SQL statements with parameters. Hence, to use a parameterized query, the first step should be the creation of `PreparedStatement` object. As done for `Statement` objects, the `PreparedStatement` object also required to be created by a `Connection` object.

➤ Creating Parameterized Query

The `PreparedStatement` object is created by using the `prepareStatement()` method of `Connection` class. This method accepts an SQL statement as an argument, unlike the `Statement` object. As this is supposed to be a pre-compiled statement, the database must know the SQL statement.

For runtime parameters to a dynamic SQL statement, a placeholder "?" is used to indicate that a variable is expected in that place.

Code Snippet 6 shows how to create parameterized query.

Code Snippet 6:

```
import java.sql.PreparedStatement;
...
String sqlStmt = "UPDATE Department SET Dept_ID = ? WHERE Dept_Name
LIKE ?";
PreparedStatement pStmt = cn.prepareStatement(sqlStmt);
```

The `PreparedStatement` object `pStmt` is created and assigned the SQL statement, "UPDATE Employees SET Dept_ID = ? WHERE Dept_Name LIKE ?", which is a pre-compiled query. UPDATE operation requires that the table must have a primary key.

➤ **Passing Parameters**

At the time of compilation, the parameter values are passed and used in the place of "?" placeholder. While compiling, the placeholder becomes a part of the statement and appears as static to the compiler. Hence, the database system does not have to recompile the statement regardless of what value is assigned to the variable. To substitute the "?" placeholder with a supplied value, one of the `setXXX()` method of the required primitive type is used. For example, If an integer value must be set in the placeholder then, the `setInt()` method is used.

Code Snippet 7 shows how to pass parameters.

Code Snippet 7:

```
pStmt.setInt(1, 25);
pStmt.setString(2, "Production");
```

The first line of code sets the first question mark placeholder to a Java `int` with a value of 25: The "1" indicates which question mark ("?") placeholder to be set and "25" is the value for the question mark ("?") placeholder. The second line of code sets the second placeholder parameter to the string "Production".

➤ **Executing Parameterized Query**

The `executeUpdate()` method is used to execute both the Statement and the `PreparedStatement` objects. This is associated with the tasks such as update, insert, and delete. The return type of this method is integer, which specifies the number of rows affected by that operation.

Code Snippet 8 demonstrates the `executeUpdate()` method.

Code Snippet 8:

```
pStmt.executeUpdate();
```

Here, no argument is supplied to `executeUpdate()` method when it is invoked to execute the `PreparedStatement` object `pStmt`. This is because `pStmt` is already assigned the SQL statement to be executed.

6.5.7 Handling Exceptions in JDBC Applications

While working with database applications and JDBC API, occasionally there may be situations that can cause exceptions. Commonly occurring exceptions during database handling with JDBC are as follows:

- **ClassNotFoundException**

While loading a driver using `Class.forName()`, if the class for the specified driver is not present in the package, then the method invocation throws a `ClassNotFoundException`. Hence, the method should be wrapped inside a `try` block, followed by a `catch` block which deals with the thrown exception (if any).

Code Snippet 9 shows use of the `ClassNotFoundException`.

Code Snippet 9:

```
try {  
    Class.forName("SpecifiedDriverClassName");  
} catch(java.lang.ClassNotFoundException e)  
{ System.err.print("ClassNotFoundException: ");  
System.err.println(e.getMessage());  
}
```

- **SQLException**

Every method defined by the JDBC objects such as `Connection`, `Statement`, and `ResultSet` can throw `java.sql.SQLException`. Hence, whenever, these methods are used, they should be wrapped inside a `try...catch` block to handle the exceptions. The vendor-specific error code is inserted inside the `SQLException` object, which is retrieved using the `getErrorCode()` method.

Code Snippet 10 shows use of `SQLException`.

Code Snippet 10:

```
try {  
    // Code that could generate an exception goes here.  
    // If an exception is generated, the catch block will print out  
    // information about it.  
}  
catch(SQLException ex)  
{  
    System.err.println("SQLException: " + ex.getMessage());  
    System.out.println("ErrorCode: " + ex.getErrorCode());  
}
```

6.5.8 Necessity for Processing Queries

Once the database query is executed and `ResultSet` object is created, the next step is to process and retrieve the result from the `ResultSet`. As the data in the `ResultSet` is in a tabular format and the cursor is positioned before the first row, it must traverse the rows using the `next()`

method. The `next()` method allows to traverse forward by moving the cursor one row forward.

It returns a boolean true if the current cursor position is on a valid row and returns a false when the cursor is placed at a position after the last row.

Code Snippet 11 demonstrates use of the `next()` method.

Code Snippet 11:

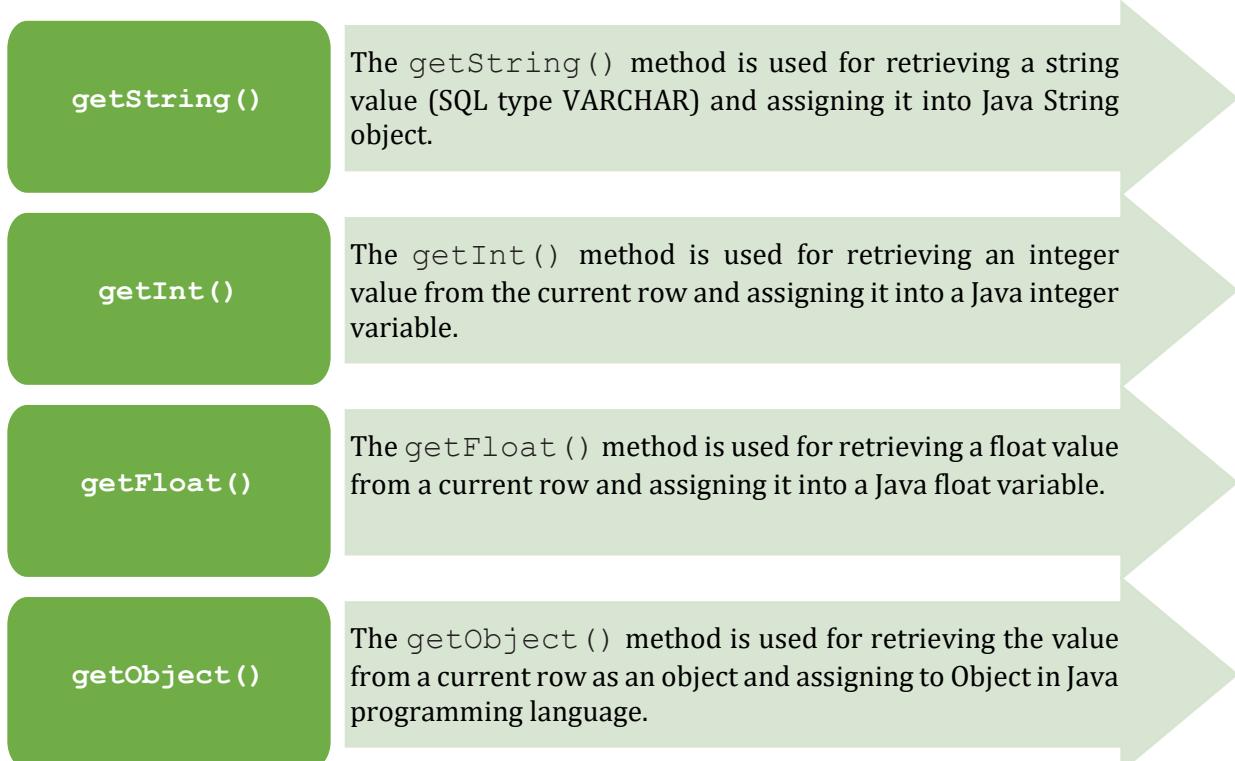
```
ResultSet rs1 = st1.executeQuery("SELECT Employee_Name FROM Employees");
while (rs1.next()) {
String name=rs1.getString("Employee_Name");
System.out.println(name);
}
```

Here, the code retrieves the employee name from the `Employee` table and displays them in a sequence using the `next()` method.

6.5.9 Methods to Process Queries

The getter methods are used to extract the data from the current row of the `ResultSet` object and store it into a Java variable of corresponding data type. These methods are declared by the `ResultSet` interface for retrieving the column values using either the index number or name of the column. If in case there are more than one column having the same name, then the value of the first matching column will be returned.

There are several getter methods to retrieve different Java type values. They are as follows:



Code Snippet 12 shows use of some of the query processing methods.

Code Snippet 12:

```
while (rs1.next()) {  
String name=rs1.getString("Employee_Name");  
int deptId=rs1.getInt("Department_Id");  
float rating=rs1.getFloat("Rating");  
System.out.println(name + " " + deptId+" "+ rating);  
}
```

The name, department, and rating values are extracted from the respective columns and stored in variables belonging to relevant data types of Java programming language. The values in the variables are finally displayed.

6.5.10 Closing the Database Connection

The last but important step in a database application is to close both the connection and any open statements, once the processing is complete. The open connection can trigger security troubles. A simple `close()` method is provided by the `Connection`, `Statement`, and `ResultSet` objects to achieve the purpose. To keep the database-releasing methods tidy and connections always released (even though an exception may be thrown), database connections should be closed within a `finally` block.

Syntax

```
public void close()
```

The method releases the JDBC resources and the database connection immediately. Code Snippet 13 shows use of the `close()` method.

Code Snippet 13:

```
st1.close();  
cn1.close();
```

The `Statement` and the `Connection` objects have been closed using the `close()` method.

6.6 Database Metadata Information

The dictionary meaning of metadata is data about data. In the context of databases it can also be defined as information that defines the structure and properties of the data stored in a database. JDBC support the access of metadata by providing several methods. For example, a table in a database has its defined name, column names, datatypes for its columns and the owner for the table, this description is known as the metadata.

6.6.1 ResultSetMetaData Interface

The information that describes the data contained in a `ResultSet` is known as the `ResultSet` metadata. This metadata information is stored by the objects of `ResultSetMetaData` interface in the `java.sql` package. This object can give the information about attributes such as column names, number of columns and data types for the columns in the result set.

To retrieve all these information, a `ResultSetMetaData` object is created and assigned with the results fetched by the `getMetaData()` method, called by the `ResultSet` object.

Code Snippet 14 shows the creation of a `ResultSetMetaData` object.

Code Snippet 14:

```
ResultSetMetaData rmd = rs.getMetaData();
```

When a Java application gets a valid connection and successfully retrieves the data from the database to a result set, this code creates a `ResultSetMetaData` object.

6.6.2 ResultSetMetaData Methods

The `ResultSetMetaData` interface offers a number of methods to get the information about the rows and columns in the `ResultSet`. These methods can only be called by using the `ResultSet` object rather than using the `Connection` object. Some of the widely used methods provided by the `ResultSetMetaData` interface are listed as follows:

- **`getColumnName()`**

The method retrieves the name of the specified column and returns the same as a `String`.

Syntax

```
public String getColumnName(int column) throws SQLException
```

where,

`String` is the return type of the method containing the column name
`column` is the column number it takes in integer format, such as 1 for the first column, 2 for the second, and so on. The method throws an `SQLException` in case, any database access error occurs.

Code Snippet 15 shows use of `getColumnName()` method.

Code Snippet 15:

```
String colName = rmd1.getColumnName(2);
```

Here, the method will retrieve the name of the second column in the resultset and store the retrieved column name value in the variable `colName`.

- **`getColumnCount()`**

The method retrieves and returns the number of columns as an integer in the current `ResultSet` object.

Syntax

```
public int getColumnCount() throws SQLException
```

where,

`int` is the return type of the method returning the number of columns. This method also throws an `SQLException` in case, any database access error occurs.

Code Snippet 16 shows use of the `getColumnName()`.

Code Snippet 16:

```
int totalCols = rmd1.getColumnName();
```

Here, the method will retrieve the number of columns present in the resultset and store the integer value in the variable `totalCols`.

6.6.3 Connecting to SQL Server Database Using Type 4 Driver

Microsoft SQL Server is a widely used database from Microsoft. It allows the user to manipulate data with a large number of data types, stored procedures, and offers a secure platform for developing enterprise database applications. Java applications can be connected to SQL Server through Type 4 JDBC Driver.

Developers can download it from following link:

<https://learn.microsoft.com/en-us/sql/connect/jdbc/download-microsoft-jdbc-driver-for-sql-server?view=sql-server-ver16>

Then, they can add it as a JAR file to the application.

The syntax to establish a connection to an SQL Server database is as follows:

Syntax

```
jdbc:sqlserver://serverName;instanceName:portNumber;property=value [  
;property  
=value]  
where,
```

`jdbc:sqlserver://` is a required string that stands for the sub-protocol and is constant. `serverName` and `instanceName` are optional and identifies the address of the server and the instance on the server to connect. The `serverName` can be a localhost or IP address of the local computer.

`portNumber` specifies the port to connect to on the `serverName`. If not specified, the default is 1433.

`property` identifies the login information such as username and password.

Code Snippet 17 shows an example of a connection string that can be used to connect to SQL Server database.

Code Snippet 17:

```
jdbc:sqlserver://localhost;user=sa;password=playware;
```

The line of code connects to the default database on the local computer by specifying a username and password.

Code Snippet 18 shows another example of how to connect to SQL Server database.

Code Snippet 18:

```
try {  
String url = "jdbc:sqlserver://127.0.0.1:1433;  
instanceName=FUJI\\SQLEXPRESS;databaseName=BankDB";  
Connection cn = DriverManager.getConnection(url, "sa", "playware");  
...  
...  
} catch (SQLException ce) {  
System.out.println("error: " + ce.getMessage());  
}
```

Here, the code establishes a connection to the BankDB database present at the FUJI/SQLEXPRESS instance, on the local machine, having username sa and password as playware. Such codes and any codes related to database operations that may raise exceptions should be enclosed in appropriate try-catch blocks. In this case, SQLException may be raised, hence, the code has a catch block for this exception. Note that necessary imports may have to be added to the complete code to compile it successfully.

6.7 Summary

- JDBC is a software API to access database connectivity for Java applications. This API provides a collection of application libraries and database drivers, whose implementation is independent of programming languages, database systems, and operating systems.
- JDBC offers four different types of drivers to connect to the databases. Each driver has its own advantages and disadvantages. The drivers are chosen based on the client/server architecture and the requirements.
- The `java.sql` package offers classes that set up a connection with databases, send the SQL queries to the databases and retrieve the computed results.
- The `Statement` object is used to send and execute the SQL queries and the `ResultSet` object is used to retrieve the computed data rows with the help of methods.
- SQL Server is a popular database from Microsoft. To develop a database application using SQL Server as the DBMS, JDBC type 4 driver can be used to establish a connection. To execute parameterized runtime queries, the `PreparedStatement` object is used.
- The information that describes the data in database is known as the metadata. The `java.sql` package defines the `ResultSetMetaData` interface to access different columns in a result set.

6.8 Check Your Progress

1. Which of the following statements about JDBC are true?

A.	The implementation of the software API for database connectivity varies as per the programming languages, underlying database systems, and operating systems.
B.	JDBC is the software API for database connectivity, which provides a set of classes and interfaces written in machine level languages to access any kind of database.
C.	JDBC is independent of the underlying database management software.
D.	In a two-tier model, the JDBC API is used to translate and send the client request to a local or remote database on the network.
E.	In a three-tier model, a 'middle tier' of services, a third server is employed to send the client request to the database server.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

2. Match the properties of drivers with the corresponding descriptions.

	Description		Driver
a.	Converts the JDBC calls into a network protocol which is again translated into database-specific calls by a middle tier	1.	Type-1
b.	Interprets JDBC calls to the database-specific native call interface.	2.	Type-3
c.	Core of JDBC API and converts the client request to a database-understandable, native format.	3.	Type-4
d.	Uses a bridging technology that provides JDBC access via ODBC drivers.	4.	Database drivers
e.	Communicates directly with the database engine using Java sockets, without a middleware or native library.	5.	Type-2

(A)	a-2, b-4, c-5, d-1, e-3	(C)	a-2, b-1, c-5, d-1, e-4
(B)	a-2, b-5, c-1, d-3, e-4	(D)	a-2, b-5, c-4, d-1, e-3

3. Choose the correct code to create a JDBC connection.

(A)	<pre>try { String url = "jdbc:sqlserver://127.0.0.1:1433; instanceName=localhost\\SQLEXPRESS; databaseName=LibraryDB"; Connection cn = DriverManager.getDatabaseConnection(url, "sa", "playware"); Statement st = cn.createStatement(); ResultSet rs = st.executeQuery("SELECT BookName FROM Books"); while(rs.next()){ String name=rs.getString("BookName"); System.out.println(name); } catch (SQLException ce) { System.out.println(ce.getMessage()); }</pre>
(B)	<pre>try { String url = "jdbc:sqlserver://127.0.0.1:1433; instanceName=localhost\\SQLEXPRESS; databaseName=LibraryDB"; Connection cn = DriverManager.getConnection(url, "sa", "playware"); Statement st = cn.createStatement(); ResultSet rs = st.executeQuery("SELECT BookName FROM Books"); while(rs.next()){ String name=rs.getString("BookName"); System.out.println(name); } catch (SQLException ce) { System.out.println(ce.getMessage()); }</pre>
(C)	<pre>try { String url = "jdbc:sqlserver://127.0.0.1:1433; instanceName=localhost\\SQLEXPRESS; databaseName=LibraryDB"; Connection cn = DriverManager.getConnection(url, "sa", "playware"); Statement st = cn.createStatement(); ResultSet rs = st.executeQuery("SELECT BookName FROM Books"); while(rs.next()){ String name=rs.getString("BookName"); System.out.println(name); } catch (SQLException ce) { System.out.println(ce.getMessage()); }</pre>

(D)	<pre> try { String url = "jdbc:sqlserver://127.0.0.1:1433; instanceName=localhost\\SQLEXPRESS; databaseName=LibraryDB"; Connection cn = DriverManager.getConnection(url, "sa", "playware"); Statement st = cn.executeQuery("SELECT BookName FROM Books"); while(rs.next()){ String name=rs.getString("BookName"); System.out.println(name); } catch (SQLException ce) { System.out.println(ce.getMessage()); } </pre>
-----	---

4. Identify the correct syntax of connecting to a SQL Server database using Type 4 driver.

A.	<code>jdbc:sql://serverName; instanceName: portNumber; property=value[;property=value]</code>
B.	<code>jdbc:sqlserver://serverName; instanceName:portNumber; [property=value[;property=value]]</code>
C.	<code>jdbc:sqlserver://instanceName:portNumber; property=value[; property=value]</code>
D.	<code>jdbc:sqlserver://serverName; instanceName; property=value[; property=value]</code>

5. To which file do you require to add a dependency for database libraries in an Apache NetBeans Maven project?

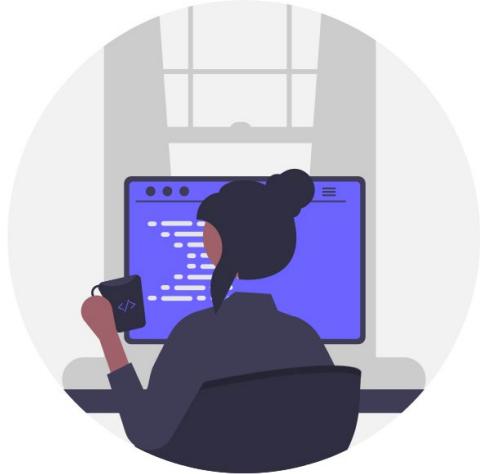
A.	<code>config.xml</code>
B.	<code>database.xml</code>
C.	<code>manifest.xml</code>
D.	<code>pom.xml</code>

6.8.1 Answers

1.	C
2.	D
3.	C
4.	B
5.	D

Try It Yourself

1. Create a Java program that connects to a MySQL database using JDBC. Print a message confirming a successful connection and handle any potential exceptions.
2. Create a program that inserts a new record into a Students table in a database. Prompt the user for student information (for example, name, age, and grade) and insert it into the database.



Session 7

Advanced JDBC

Welcome to the Session, **Advanced JDBC**.

This session describes how to work with scrollable result sets using JDBC. The session provides explanations on stored procedures, batch updates, and transactions along with various ways of implementing them. Finally, the session introduces you to JDBC 4.0 and 4.1 features, such as Rowset and its various types as well as JDBC 4.3 enhancements. The session explains RowSetProvider, RowSetFactory, and RowSet Interfaces. Finally, the session discusses how to use RowSet, JdbcRowSet and CachedRowSet objects for transferring data to and from databases.

In this Session, you will learn to:

- List and describe scrollable result sets
- List different types of ResultSet and row-positioning methods
- Explain stored procedures
- Explain how to call a stored procedure using JDBC API
- Identify the steps to update records
- Explain the steps of implementing transactions using JDBC
- List the enhancements of JDBC 4.3 API
- Explain Rowset and its type
- Describe JDBC 4.1 RowSetProvider and RowSetFactory Interfaces

7.1 *ResultSet*

The `ResultSet` object in JDBC API represents a SQL result set in the JDBC application. An SQL result set is the result table returned by the database in query execution. It includes a set of rows from a database as well as meta-information about the query such as the column names, and the types and sizes of each column. A result set in JDBC API can be thought of as a table of data representing a database result set, which is usually generated by executing a statement that queries the database. A default result set object cannot be updated or scrolled backward and forward. By default, the cursor

moves forward only. So, it can be iterated only once and only from the first row to the last row. In addition to moving forward, one row at a time, through a `ResultSet`, the JDBC Driver also provides the capability to move backward or go directly to a specific row. Additionally, updating and deleting rows of the result is also possible. The `ResultSet` can also be kept open after a `COMMIT` statement.

The characteristics of `ResultSet` are as follows:

Scrollable	Updatable	Holdable
It refers to the ability to move backward as well as forward through a <code>resultset</code> .	It refers to the ability to update data in a result set and then, copy the changes to the database. This includes inserting new rows into the result set or deleting existing rows.	It refers to the ability to check whether the cursor stays open after a <code>COMMIT</code> .

7.1.1 Scrollable `ResultSet`

A scrollable result set allows the cursor to be moved to any row in the result set. This capability is useful for GUI tools for browsing result sets. Since scrollable result sets involve overhead, they should be used only when the application requires scrolling. You can create a scrollable `ResultSet` through the methods of the `Connection` interface.

Table 7.1 lists the methods that can be invoked on the connection instance for returning a scrollable `ResultSet`.

Method	Syntax	Code
<code>createStatement()</code>	<pre>public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException</pre> <p>where, <code>resultSetType</code>: argument that will help to create a scrollable <code>ResultSet</code>.</p> <p>It represents constant values that can be <code>ResultSet.TYPE_FORWARD</code>, <code>ResultSet.TYPE_SCROLL_SENSITIVE</code>, or <code>ResultSet.TYPE_SCROLL_INSENSITIVE</code>. <code>resultSetConcurrency</code>: Represents one of the two <code>ResultSet</code> constants for specifying whether a result set is read-only or updatable.</p>	<pre>Statement st = cn.createStatement(ResultSet.TYPE_ SCROLL_INSENSIT_ IVE, ResultSet.CONCU_ R_READ_ONLY); ResultSet rs = st.executeQuery ("SELECT EMP_ NAME, DEPT FROM EMPLOYEES");</pre>

Method	Syntax	Code
	The constant values for concurrency type can be <code>ResultSet.CONCUR_READ_ONLY</code> or <code>ResultSet.CONCUR_UPDATABLE</code> .	
<code>prepareCall()</code>	<p><code>public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency) throws SQLException</code></p> <p>where,</p> <p><code>sql</code> represents a <code>String</code> object containing the SQL statements to be sent to the database.</p> <p><code>resultSetType</code> has the same attributes as in the <code>createStatement()</code> method.</p> <p><code>resultSetConcurrency</code> represents one of the two <code>ResultSet</code> constants for specifying whether a result set is read-only or updatable.</p>	<code>CallableStatement cs = cn.prepareCall("?" = CALL EMPLOYEE(?, ?, ?)", ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);</code>

Table 7.1: Methods Returning Scrollable ResultSet

7.1.2 Types of ResultSet Values

The `ResultSet` interface in Java provides access to a table of data that represents a database result set. A `ResultSet` object is usually generated by executing a statement that queries the database.

Different static constant values that can be specified for the result set type are as follows:

➔ `TYPE_FORWARD_ONLY`

A cursor that can only be used to process from the beginning of a `ResultSet` to the end of it. The cursor only moves forward. This is the default type. You cannot scroll through this type of result set nor is it positionable and lastly, it is not sensitive to the changes made to the database.

Code Snippet 1 demonstrates the `TYPE_FORWARD_ONLY` cursor. It is assumed that a table **Employees** has been created under a database **BankDB** in SQL Server. This table is assumed to have four fields namely, `EMP_NO`, `NAME`, `SALARY`, and `RATING` of type `int`, `varchar`, `int`, and `float` respectively. A few records must be present with one of the employees having `Emp_No` as 28959.

Code Snippet 1:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Statement;

public class DataDemo {
```

```

public static void main(String args[]) {
try {
Connection cn = DriverManager.
        getConnection("jdbc:sqlserver://127.0.0.1:1433;
instanceName=FUJI\\SQLEXPRESS;databaseName=
BankDB", "sa", "playware");
// ResultSet block
PreparedStatement pst = cn.prepareStatement(
"SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",
ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
pst.setString(1, "28959");
ResultSet rs = pst.executeQuery();
// end of Resultset block
int empnum;
int sal;
Statement st = cn.createStatement();
st.executeUpdate("UPDATE employees set SALARY=2990 WHERE EMP_NO =
28959");

while (rs.next()) {
empnum = rs.getInt("EMP_NO");
sal = rs.getInt("SALARY");
System.out.println("EMP_NO: "+empnum);
System.out.println("SALARY: "+ sal);
}
} catch (SQLException ce) {
System.out.println(ce.getMessage());
}
}
}
}

```

A query statement is constructed using `PreparedStatement` to retrieve details of the employee whose employee number is 28959. The fields `EMP_NO` and `SALARY` are insensitive to changes made to the database while the resultset is open. Using `rs.next()` method, iterate through the resultset and display the values.

While the program is running, if you change the values of `SALARY` as shown here, it has no effect on the output of the program. The old salary before updation will be displayed.

→ `TYPE_SCROLL_INSENSITIVE`

A cursor that can be used to scroll in various ways through a `ResultSet`. This type of cursor is insensitive to changes made to the database while it is open. It contains rows that satisfy the query when the query was processed or when data is fetched.

Code Snippet 2 demonstrates the `TYPE_SCROLL_INSENSITIVE` cursor. Replace the code in Code Snippet 1 marked with comments for ResultSet block with following code. Save under a different name (such as **DataDemo2.java**) and execute it.

Code Snippet 2:

```
PreparedStatement pst = cn.prepareStatement  
("SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",  
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);  
pst.setString(1, "28959");  
ResultSet rs = pst.executeQuery();
```

Here too, the fields **EMP_NO** and **SALARY** are insensitive to changes made to the database while it is open. Similar to the outcome of Code Snippet 1, the old salary before updation will be displayed. The fields **EMP_NO** and **SALARY** are insensitive to changes made to the database while the resultset is open.

→ `TYPE_SCROLL_SENSITIVE`

A cursor that can be used to scroll in various ways through a ResultSet. This type of cursor is sensitive to changes made to the database while it is open. Changes to the database have a direct impact on the ResultSet data.

Code Snippet 3 shows the use of `TYPE_SCROLL_SENSITIVE` cursor. Replace the relevant block in Code Snippet 1 with following code, save under a different name, and execute the code.

Code Snippet 3:

```
PreparedStatement pst = cn.prepareStatement  
("SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",  
ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);  
pst.setString(1, "28959");  
ResultSet rs = pst.executeQuery();
```

This time, any changes made to the fields **EMP_NO** and **SALARY** will have a direct impact on the ResultSet data. Hence, the updated salary will be displayed in the output.

7.1.3 Row Positioning Methods

By default, ResultSet always allows forward movement only, meaning that the only valid cursor-positioning method to call is `next()`. You have to explicitly request for a scrollable ResultSet. Table 7.2 describes the cursor-positioning methods of ResultSet.

Method	Description
<code>next()</code>	This method moves the cursor forward one row in the ResultSet from the current position. The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.

Method	Description
previous ()	The method moves the cursor backward one row in the <code>ResultSet</code> . The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.
first ()	The method moves the cursor to the first row in the <code>ResultSet</code> . The method returns <code>true</code> if the cursor is positioned on the first row and <code>false</code> if the <code>ResultSet</code> is empty.
last ()	The method moves the cursor to the last row in the <code>ResultSet</code> . The method returns <code>true</code> if the cursor is positioned on the last row and <code>false</code> if the <code>ResultSet</code> is empty.
beforeFirst ()	The method moves the cursor immediately before the first row in the <code>ResultSet</code> . There is no return value from this method.
afterLast ()	The method moves the cursor immediately after the last row in the <code>ResultSet</code> . There is no return value from this method.
relative (int rows)	The method moves the cursor relative to its current position. If row value is 0, this method has no effect. If row value is positive, the cursor is moved forward that many rows. If there are fewer rows between the current position and the end of the <code>ResultSet</code> than specified by the input parameters, this method operates like <code>afterLast()</code> method. If row value is negative, the cursor is moved backward that many rows. The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.
absolute (int row)	The method moves the cursor to the row specified by row value. If row value is positive, the cursor is positioned that many rows from the beginning of the <code>ResultSet</code> . The first row is numbered 1, the second is 2, and so on. If row value is negative, the cursor is positioned that many rows from the end of the <code>ResultSet</code> . The last row is numbered -1, the second to last is -2, and so on. If row value is 0, this method operates like <code>beforeFirst()</code> . The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.

Table 7.2: Cursor-positioning Methods of `ResultSet`

Note: Calling `absolute(1)` is equivalent to calling `first()` and calling `absolute(-1)` is equivalent to calling `last()`.

7.1.4 Updating a Row

Rows may be updated in a database table by using an object of the `ResultSet` interface. There are two steps involved in this process. The first step is to change the values for a specific row using various `update<Type>` methods, where `<Type>` is a Java data type. These `update<Type>` methods correspond to the `get<Type>` methods available for retrieving values. The second step is

to apply the changes to the rows of the underlying database. The database itself is not updated until the second step. Updating columns in a `ResultSet` without calling the `updateRow()` method does not make any changes to the database. Once the `updateRow()` method is called, changes to the database are final and cannot be undone.

Consider a scenario where you want to update the `EMP_NO` field of the first row retrieved in the result set.

This operation could be accomplished in several steps.

→ Step 1: Positioning the Cursor

Code Snippet 4 shows how to position the cursor.

Code Snippet 4:

```
// Create an updatable result set
Statement st = cn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
// Retrieve the scrollable and updatable result set
ResultSet rs = st.executeQuery("SELECT NAME, EMP_NO FROM
EMPLOYEES");
// Move to the first row in the result set
rs.first();
```

The `first()` method moves the cursor to the first row in the result set. After invoking this method, any call to update methods affects the values of first row until the cursor is moved to another row.

Similarly, there are other methods such as `last()`, `next()`, `previous()`, `beforeFirst()`, `afterLast()`, `absolute(int)`, and `relative(int)` that can be used to move the cursor to the required location in the result set.

→ Step 2: Updating the Columns

After using the `first()` method to navigate to the first row of the result set, call `updateInt()` to change the value of the `EMP_NO` column in the result set. Code Snippet 5 shows how to update the columns.

Code Snippet 5:

```
// Update the second column in the result set
rs.updateInt(2, 34523);
```

The `updateInt()` method is used because the column to be updated has integer values. The `updateInt()` method has two parameters. The first parameter specifies the column number that is to be updated. The second parameter specifies the new value that will replace with the existing column value for that particular record.

Similarly, there are other update methods such as `updateString()`, `updateFloat()`, and `updateBoolean()` that can be used to update a particular column consisting of a string, float,

and boolean value respectively.

→ Step 3: Committing the Update

After making the change, call `updateRow()` method of the `ResultSet` interface to actually reflect the change in the underlying database as shown in Code Snippet 6.

Code Snippet 6:

```
// Committing the row updation  
rs.updateRow();
```

If the `updateRow()` method is not called before moving to another row in the result set, any changes made will be lost.

To make a number of changes in a single row, make multiple calls to `updateXXX()` methods and then, a single call to `updateRow()`. Be sure to call `updateRow()` before moving on to another row.

7.1.5 Steps for Inserting a Row

The procedure for inserting a row is like that of updating data in an existing row, with a few differences. To insert a row, you first position the cursor, next update one or more columns, and then finally, commit the changes.

→ Step 1: Positioning the Cursor

An updatable result supports a row called the insert row. It is a buffer for holding the values of a new row. The first step is to move to the insert row, using the `moveToInsertRow()` method.

Code Snippet 7 shows the use of `moveToInsertRow()` method.

Code Snippet 7:

```
// Create an updatable result set  
ResultSet rs = st.executeQuery("SELECT NAME, EMP_NO FROM  
EMPLOYEES");  
// Move cursor to insert row  
rs.moveToInsertRow();
```

Here, the insert row is an empty row containing all the fields but no data and is associated with the `ResultSet` object. It can be thought of as a temporary row in which you can compose a new row.

→ Step 2: Updating the Columns

After moving to the insert row, use `updateXXX()` methods to load new data into the insert row. Code Snippet 8 shows the code to update the columns.

Code Snippet 8:

```
// Set values for the new row  
rs.updateString(1, "William Ferris");  
rs.updateInt(2, 35244);
```

The first line of code updates the Employee Name as "William Ferris" in the first column of the result set using the `updateString()` method. The second line of code updates `EMP_NO` as 35244 in the second column using the `updateInt()` method. Ensure that all non-null fields have been specified with values.

→ Step 3: Inserting the Row

After using the `updateXXX()` method, the `insertRow()` method is called to append the new row to the `ResultSet` and the underlying database.

Code Snippet 9 shows the code to insert the row.

Code Snippet 9:

```
// Commit appending of new row to the resultset  
rs.insertRow();
```

After calling the `insertRow()` method, another new row can be created. Using various navigation methods, you can also move back to the `ResultSet`. There is one more navigation method named `moveToCurrentRow()` that takes you back to where you were before you called `moveToInsertRow()`. Note, that it can only be called while you are in the insert row.

7.1.6 Steps for Deleting a Row

Deleting a row from an updatable result set is easy. The two steps to delete the row are to position the cursor and delete the row.

Deleting a row from the result set has been described in steps.

→ Step 1: Positioning the Cursor

The first step is to positioning the cursor by moving the cursor to the desired row that is to be deleted as shown in Code Snippet 10.

Code Snippet 10:

```
// Move the cursor to the last row of the resultset  
rs.last();
```

Here, the cursor is moved to the last record of the result set that is to be deleted. One can move to the desired row by using any of various navigation methods of `ResultSet` interface for deleting a particular row.

→ Step 2: Deleting the Row

The second step is to delete the row. After moving to the last row of the result set, call the `deleteRow()` method to commit the deletion of the row. Code Snippet 11 shows the code to delete the row.

Code Snippet 11:

```
// Deleting the row from the record set  
rs.deleteRow();
```

Calling the `deleteRow()` method of the `ResultSet` interface also deletes the row from the underlying database.

Some JDBC drivers will remove the row and the row will not be visible in the result set. Other JDBC drivers will place a blank row in place of the deleted row. Then, the `absolute()` method can be used with the original row positions to move the cursor because the row numbers in the result set are not changed by deletion.

7.2 *Stored Procedures*

A stored procedure can be defined as a group of SQL statements performing a particular task. Stored procedures are used to group or batch a set of operations or queries to be executed on a database server. Stored procedures having any combination of input, output, or input/output parameters can be compiled and executed. As stored procedures are pre-compiled, they are faster and more efficient than using individual SQL query statements. Stored procedures are supported by database systems such as SQL Server, Oracle, or MySQL. The stored procedure is called from a Java class using a special syntax. When the procedure is called, the name of the procedure and the parameters you specify are sent over the JDBC connection to the DBMS, which executes the procedure and returns the results back over the connection.

7.2.1 *Creating a Stored Procedure Using Statement Object*

Stored procedures can be created using a `Statement` object. Creating a stored procedure with the `Statement` object involves two steps.

→ **Creating stored procedure definition and storing it in a String variable**

Code Snippet 12 shows the code to declare the string variable containing the definition of a stored procedure. Assume that tables `PRODUCTS` and `COFFEES` are existing.

Code Snippet 12:

```
String createProcedure = "Create Procedure DISPLAY_PRODUCTS " + "as" + "select PRODUCTS.PRD_NAME, COFFEES.COF_NAME " + "from PRODUCTS, COFFEES " + "where PRODUCTS.PRD_ID = COFFEES.PRD_ID " + "order by PRD_NAME";
```

The code creates a SQL statement for a stored procedure and stores it in a string variable `createProcedure`.

→ **Using the Statement object**

Code Snippet 13 shows the use of the `Statement` object.

Code Snippet 13:

```
// An active connection cn is used to create a Statement object
Statement st = cn.createStatement();
// Execute the stored procedure
st.executeUpdate(createProcedure);
```

Figure 7.1 displays the sequence of creating stored procedure using `Statement` object.

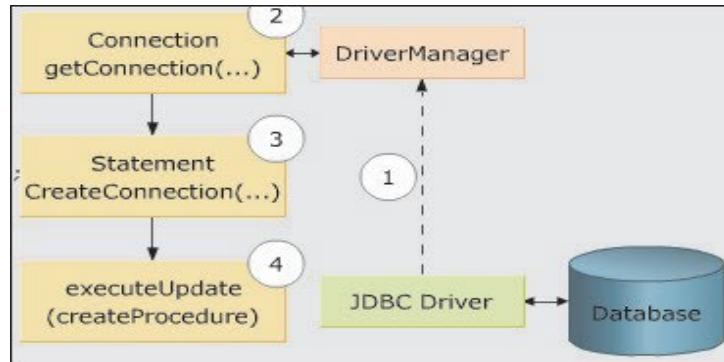


Figure 7.1: Stored Procedure Using Statement Object

In Code Snippet 13, the `Connection` object `cn` is used to create a `Statement` object. The `Statement` object is used to send the SQL statement creating the stored procedure to the database. The procedure `DISPLAY_PRODUCTS` is compiled and stored in the database as a database object and it can be called. On successful completion of `executeUpdate()` method, a procedure named `DISPLAY_PRODUCTS` is created. Assume that this code has been run against an SQL Server connection to a database named Traders. Now, after executing Code Snippet 13, open SQL Server Management, navigate to the database, and look under **Programmability -> Stored Procedures** as shown in Figure 7.2. You will find `DISPLAY_PRODUCTS`.

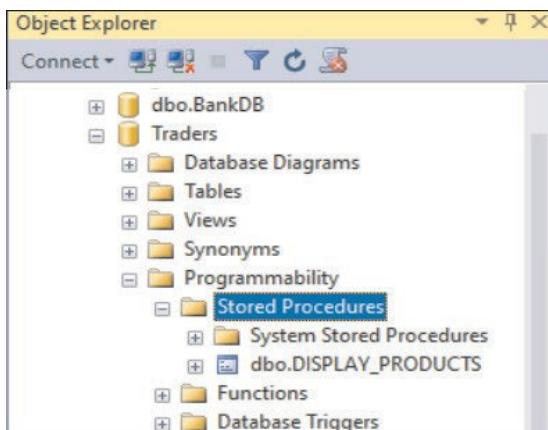


Figure 7.2: Viewing the Stored Procedure in SSMS

7.2.2 Creating a CallableStatement Object

A stored procedure can be called from a Java application with the help of a `CallableStatement` object. A `CallableStatement` object does not contain the stored procedure itself but contains only a call to the stored procedure.

The call to a stored procedure is written in an escape syntax, that is, the call is enclosed within curly braces. The call may take two forms, such as with a result parameter, and without a result parameter. The result parameter is a value returned by a stored procedure, like an `OUT` parameter. Both the forms have a different number of parameters used as input (`IN` parameters), output (`OUT` parameters), or both (`INOUT` parameters). A question mark (?) is used to represent a placeholder.

for a parameter.

Syntax for calling a stored procedure without parameters is as follows:

Syntax

```
{call procedure_name}
```

Syntax for calling a stored procedure in JDBC is as follows:

Syntax

```
{call procedure_name[ (?, ?, ...) ] }
```

Placeholders enclosed in square brackets indicate that they are optional.

Syntax for a procedure that returns a result parameter is as follows:

Syntax

```
{? = call procedure_name[ (?, ?, ...) ] }
```

CallableStatement inherits methods from the Statement and PreparedStatement interfaces. All methods that are defined in the CallableStatement interface deal with OUT parameters. The getXX() methods such as getInt() and getString() in a ResultSet will retrieve values from a result set whereas in a CallableStatement, they will retrieve values from the OUT parameters or return values of a stored procedure.

CallableStatement objects are created using the prepareCall() method of the Connection interface. The section enclosed within the curly braces is the escape syntax for stored procedures. The driver converts the escape syntax into native SQL used by the database.

Syntax

```
CallableStatement cst = cn.prepareCall("{call functionname(?, ?)}");
```

where, cst is the name of the CallableStatement object. functionname is the name of function/procedure to be called.

Consider following statement:

```
CallableStatement cs = cn.prepareCall( "{call sal(?)}");
```

The statement creates an instance of CallableStatement. It contains a call to the stored procedure sal(), which has a single argument and no result parameter. The type of the (?) placeholder parameters whether it is IN, or OUT parameter is totally dependent on the way the stored procedure sal() has been defined.

→ IN Parameters

The set<Type>() methods are used to pass any IN parameter values to a CallableStatement object. These set<Type>() methods are inherited from the PreparedStatement object. The type of the value being passed in determines which set<Type>() method to use. For example, setFloat() is used to pass in a float value, and so on.

→ OUT Parameters

In case the stored procedure returns some values (OUT parameters), the JDBC type of each OUT parameter must be registered before executing the `CallableStatement` object. The `registerOutParameter()` method is used to register the JDBC type. After the registration is done, the statement has to be executed. The `get<Type>()` methods of `CallableStatement` are used to retrieve the OUT parameter value. The `registerOutParameter()` method uses a JDBC type (so that it matches the JDBC type that the database will return), and `get<Type>()` casts this to a Java type.

Some common JDBC types are as follows:

- **Char**: used to represent fixed-length character string.
- **Varchar**: used to represent variable-length string.
- **Bit**: used to represent single bit value that can be zero or one.
- **Integer**: used to represent a 32-bit signed integer value.
- **Double**: used to represent a double-precision floating point number that supports 15 digits of mantissa.
- **Date**: used to represent a date consisting of the day, month, and year.

Code Snippet 14 shows the code to create a stored procedure named `getData` in SQL Server. Run this code in a new query editor in SQL Server Management Studio.

Code Snippet 14:

```
CREATE PROCEDURE getData (
@OutSum int output, @OutMultiply int output
) AS
DECLARE @In int SET @In=9
SELECT @OutSum = @In + @In, @OutMultiply = @In * @In
GO
DECLARE @Out1 int, @Out2 int
EXEC getData @OutSum = @Out1 OUTPUT, @OutMultiply = @Out2 OUTPUT
SELECT @Out1 As Out1, @Out2 As Out2
GO
```

Code Snippet 15 demonstrates how to use the `registerOutParameter()` method to pass output parameters to the procedure.

Code Snippet 15:

```
...
CallableStatement cs = cn.prepareCall("{call getData(?, ?)}");
cs.registerOutParameter(1, java.sql.Types.INTEGER);
cs.registerOutParameter(2, java.sql.Types.INTEGER, 3);
cs.execute();
int x = cs.getInt(1); int n = cs.getInt(2);
System.out.println(x + " " +n);
...
```

The code registers the OUT parameters and executes the stored procedure called by `cs`. Then, it

retrieves the values returned in the OUT parameters. The getInt() methods retrieve integers from the first OUT parameter and second OUT parameter respectively.

CallableStatement does not provide a special mechanism to retrieve large OUT values incrementally, which can be done with the ResultSet object.

Code Snippet 16 demonstrates how to retrieve an OUT parameter returned by a stored procedure.

Code Snippet 16:

```
import java.sql.*;
import java.util.*;
public class CallOutProc {
Connection con;
String url, serverName, instanceName, databaseName, userName,
password, sql;
CallOutProc() {
url = "jdbc:sqlserver://"; serverName = "127.0.0.1:1433";
instanceName="FUJI\\SQLEXPRESS"; databaseName="BankDB";
userName = "sa"; password = "playware";
}
private String getConnectionUrl() {
// Constructing the connection string
return url + serverName + ";instanceName = " +instanceName + " ;
DatabaseName = " +databaseName;
}
private java.sql.Connection getConnection() {
try {
// Establishing the connection
con = DriverManager.getConnection(getConnectionUrl(), userName,
password);
if(con != null)
System.out.println("Connection Successful!");
} catch(Exception e) {
e.printStackTrace();
System.out.println("Error Trace in getConnection(): "
+ e.getMessage());
}
return con;
}
public void display(){
try {
con = getConnection();
CallableStatement cstmt = con.prepareCall("{call recalculatetotal
(?,?) }");
cstmt.setInt(1,2500);
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.execute();
int maxSalary = cstmt.getInt(2);
System.out.println(maxSalary);
}
```

```

} catch(SQLException ce) {
System.out.println(ce);
}
}

public static void main(String args[]) {
CallOutProc proObj = new CallOutProc();
proObj.display();
}
}

```

The CallableStatement makes a call to the stored procedure called `recalculatetotal`. The integer variable ‘`a`’ from the stored procedure is initialized to a value of 2500 by passing an argument through the `setInt()` method. The `OUT` parameter is then registered with JDBC as belonging to data type `java.sql.Types.Integer`. The CallableStatement is then executed and this, in turn, executes the stored procedure `recalculatetotal`. The value of the `OUT` parameter is retrieved by invoking the `getInt()` method. This value is the maximum salary from the `Employee` table and stored in an integer variable `maxSalary` and displayed.

A procedure for recalculating the salary of the highest salary earner is shown in Code Snippet 17. This code must be executed before Code Snippet 16. The procedure will accept a value and will store the value in an `OUT` parameter.

Code Snippet 17:

```

create procedure recalculatetotal @a int, @inc_a int OUT
as
select @inc_a = max(salary) from Employee set @inc_a = @a * @inc_a;

```

7.3 Batch Update

Batch update can be defined as a set of multiple update statements that is submitted to the database for processing as a batch. In Java, the `Statement`, `PreparedStatement`, and `CallableStatement` objects can be used to submit batch updates.

The benefits of batch updating is that it allows you to request records, bring them to the client, make changes to the records on the client side, and then, send the updated record back to the data source at some other time. Submitting multiple updates together, instead of individually, can greatly improve performance. Also, batch updating is used when there is no required to maintain a constant connection to the database.

7.3.1 Batch Update Using Statement Interface

The batch update facility allows a `Statement` object to submit multiple update commands together as a single unit, or batch, to the underlying DBMS.

The steps to implement batch update using the `Statement` interface are as follows:

→ **Disable the auto-commit mode**

The auto-commit mode of `Connection` object is set to `false` in order to allow multiple statements to be sent together as a transaction. For handling the errors correctly, the auto-commit mode should be always disabled before beginning a batch update.

Code Snippet 18 shows how to disable auto-commit mode.

Code Snippet 18:

```
// turn off auto-commit  
cn.setAutoCommit(false);
```

To start sending a batch update to a database, first the auto-commit mode of the connection object `con` is set to `false`. Since the connection's auto-commit mode is disabled, the application is free to decide whether or not to commit the transaction if an error occurs or if some of the commands in the batch fail to execute.

→ **Create a Statement instance**

An instance of `Statement` interface is created by calling the `createStatement()` method on the `Connection` object. Initially, the newly created `Statement` object has no list of commands associated with it.

Code Snippet 19 shows use of the `createStatement` method.

Code Snippet 19:

```
//Create an instance of Statement object  
Statement st = cn.createStatement();
```

An instance of `Statement` object `st` is created that has initially an empty list of commands associated with it.

→ **Add SQL commands to the batch**

Commands are added to the list of commands associated with the `Statement` object.

The commands are added to the list with the `addBatch()` method of the `Statement` interface. A `Statement` object can keep track of a list of commands or batch that can be submitted together for execution.

Code Snippet 20 shows how to add SQL commands to the batch.

Code Snippet 20:

```
// Adding the calling statement batch  
st.addBatch("INSERT INTO EMPLOYEES VALUES (1000, 'William John',  
8000)";  
st.addBatch("INSERT INTO EMPLOYEES VALUES (1001, 'Jacky Lawrence',
```

```
9000) ;  
st.addBatch("INSERT INTO EMPLOYEES VALUES (1002, 'Valentina  
Watson', 4000)");
```

When a `Statement` object is created, its associated list of commands is empty. Each of these `st.addBatch()` method adds a command to the calling statement's list of commands. These commands are all `INSERT INTO` statements, each one adding a row consisting of two column values.

→ Execute the batch commands

The `executeBatch()` method is called to submits the list of commands of the `Statement` object to the underlying DBMS for execution. The command gets executed in the order in which it was added to the batch and returns an update count for each command in the batch, also in order.

Code Snippet 21 shows how to execute the batch commands.

Code Snippet 21:

```
// submit a batch of update commands for execution  
int[] updateCounts = st.executeBatch();
```

The commands are executed by DBMS in the order in which they were added to the list of commands. First, it will add the row of values for 1000; next it will add the row of values for 1001, and finally, 1002. The DBMS will return an update count for each command in the order in which it was executed, provided all the three commands are executed successfully. The integer values indicating how many rows were affected by each command are stored in the array `updateCounts`.

If all the three commands in the batch were executed successfully, `updateCounts` will contain three values, all of which are 1 because an insertion affects one row.

→ Commit the changes in the database

The `commit()` method makes the batch of updates to the table permanently. This method must be called explicitly because the auto-commit mode for this connection was disabled earlier.

Code Snippet 22 shows how to commit changes in the database.

Code Snippet 22:

```
// Enabling auto-commit mode  
cn.commit();  
cn.setAutoCommit(true);
```

The first line of code will commit the changes and makes the batch of updates to the table permanent. The second line of code will automatically enable the auto-commit mode of the `Connection` interface. The statement will be committed after it is executed, and an application no longer must invoke the `commit()` method.

→ Remove the commands from the batch

The `clearBatch()` method empties the current list of SQL commands for the Statement object. This `clearBatch()` method is specified by the `clearBatch()` method in the `java.sql.Statement` interface.

Code Snippet 23 shows how to remove commands from a batch.

Code Snippet 23:

```
// Emptying the current list of SQL commands  
st.clearBatch();
```

The method `st.clearBatch()` can be called to reset a batch if the application decides not to submit a batch of commands that has been constructed for a statement.

7.3.2 Batch Update Using `PreparedStatement` Interface

The batch update facility is used with a `PreparedStatement` to associate multiple sets of input parameter values with a single `PreparedStatement` object. The `addBatch()` method of the `Statement` interface is given an SQL update statement as a parameter, and the SQL statement is added to the `Statement` object's list of commands to be executed in the next batch. It is an example of static batch updates.

`PreparedStatement` interface allows creating parameterized batch update. The `setXXX()` methods of the `PreparedStatement` interface are used to create each parameter set, while the `addBatch()` method adds a set of parameters to the current batch. Finally, the `executeBatch()` method of the `PreparedStatement` interface is called to submit the updates to the DBMS, which also clears the statement's associated list of batch elements.

Code Snippet 24 shows how to perform batch updates using `PreparedStatement`.

Code Snippet 24:

```
import java.sql.*;  
public class BatchDemo {  
public static void main(String args[]) {  
Connection cn;  
try {  
// Establishing the connection  
cn = DriverManager.getConnection("jdbc:sqlserver://127.0.0.1:1433;  
instanceName=FUJI\\SQLEXPRESS;databaseName=BankDB", "sa",  
"playware");  
if(cn != null) {  
System.out.println("Connection Successful!");  
// Turn off auto-commit  
cn.setAutoCommit(false);
```

```

// Creating an instance of PreparedStatement
PreparedStatement pst = cn.prepareStatement("INSERT INTO EMPLOYEES
VALUES (?, ?, ?)");
// Adding the calling statement batches
pst.setInt(1, 5000);
pst.setString(2, "Roger Hoody");
pst.setInt(3, 6700);
pst.addBatch();
pst.setInt(1, 6000);
pst.setString(2, "Kelvin Keith");
pst.setInt(3, 5500); pst.addBatch();
// Submit the batch for execution
int[] updateCounts = pst.executeBatch();
// Enable auto-commit mode
cn.commit();
}
}catch(SQLException e) {
e.printStackTrace();
System.out.println("Error Trace in getConnection(): "
+ e.getMessage());
}
}
}
}

```

The `pst.executeBatch()` method is called to submit the updates to the DBMS. Calling `pst.executeBatch()` clears the statement's associated list of batch elements. The array returned by `pst.executeBatch()` contains an element for each set of parameters in the batch, similar to the case for `Statement` interface.

7.3.3 Batch Update using CallableStatement Interface

The functionality of a `CallableStatement` object and a `PreparedStatement` object is same. The batch update facility on a `CallableStatement` object can call only stored procedures that take input parameters or no parameters at all. Also, the stored procedure must return an updated count. The `executeBatch()` method of the `CallableStatement` interface that is inherited from `PreparedStatement` interface will throw a `BatchUpdateException` if the return value of stored procedure is anything other than an update count or takes OUT or IN/OUT parameters.

Code Snippet 25 shows batch update using `CallableStatement` interface. In this case, a different table **ProductDetails** is assumed to exist.

Code Snippet 25:

```

CallableStatement cst = cn.prepareCall("{call
                                         updateProductDetails(?, ?)}");
// Adding the calling statement batches
cst.setString(1, "Cheese");

```

```
cst.setFloat(2, 70.99f);
cst.addBatch();
cst.setString(1, "Almonds");
cst.setFloat(2, 80.99f);
cst.addBatch();
// Submitting the batch for execution
int [] updateCounts = cst.executeBatch();
// Enabling auto-commit mode cn.commit();
```

The `CallableStatement` object, `cst`, contains a call to the stored procedure named `updateProductDetails` with two sets of parameters associated with it. When `cst` is executed, two statements that call the stored procedure will be executed together as a batch. The first record will have the values as "Cheese" and 70.99f respectively, and second record as "Almond" and 80.99f respectively. The letter `f` followed by a number, as in `70.99f`, tells the Java compiler that the value is a float.

7.4 *Transactions*

A transaction is a set of one or more statements that are executed together as a unit. This ensures that either all the statements in the set are executed or none of them is executed. Transactions also help to preserve the integrity of the data in a table. To avoid conflicts during a transaction, a DBMS will use locks, which are mechanisms for blocking access by others to the data that is being accessed by the transaction. Once a lock is set, it will remain in force until the transaction is committed or rolled back.

Consider the scenario of a bank funds transfer that requires withdrawing money from one account to deposit into another. If the process of withdrawal is completed, but the deposit fails then, the customer faces a loss. If the deposit succeeds but the withdrawal fails then, the bank faces a loss.

Since both the steps form a single transaction, a 'transaction' is often defined as an indivisible unit of work.

7.4.1 *Properties of Transactions*

The properties Atomicity, Consistency, Isolation and Durability (ACID) guarantee that database transactions are processed reliably.

Atomicity	It refers to the ability of the database management system to guarantee that either all the tasks of a transaction are performed or none of them are performed.
Consistency	It refers to the database being in a legal state in that a transaction cannot break rules such as integrity constraints.
Isolation	It refers to the ability of the DBMS to ensure that there are no conflicts between concurrent transactions. For example, if two people are updating the same item, then one person's changes should not be clobbered when the second person saves a different set of changes. The two users should be able to work in isolation.
Durability	It refers to the ability of DBMS to recover committed transactions even if the system or storage media fails.

7.4.2 Implementing Transactions Using JDBC

There are four steps to implement transactions using JDBC, such as Start the transaction, perform transactions, Use Savepoint, and Close or End transaction.

→ Step 1: Start the Transaction

The first step is to start the transaction. When a connection is created, by default, it is in the auto-commit mode. Hence, each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. When you disable auto-commit, the start and end of a transaction is defined which lets you determine whether to commit or rollback the entire transaction.

To disable a connection's auto-commit mode, you invoke the `setAutoCommit()` method, which accepts a single boolean parameter, as shown:

```
cn.setAutoCommit(false);
```

To start the transaction, the active connection auto-commit mode is disabled. Once auto-commit mode is disabled, no SQL statements will be committed until the method `commit` is called explicitly. All statements executed after the previous call to the method `commit` will be included in the current transaction and will be committed together as a unit.

→ Step 2: Perform Transactions

The second step is to perform the transaction as shown in Code Snippet 26. In this code, it is assumed that a table named COFFEES has been created and populated with records.

Code Snippet 26:

```
PreparedStatement modifySales = null;
PreparedStatement modifyTotal = null;
String updateString = "update " + ".COFFEES " + "set SALES = ?
```

```

where COF_NAME = ?";
PreparedStatement modifySales = null;
PreparedStatement modifyTotal = null;
String updateString = "update " + ".COFFEES " + "set SALES = ?
where COF_NAME = ?";
String updateStatement = "update " + ".COFFEES " + "set TOTAL =
TOTAL + ? " + "where COF_NAME = ?";
cn.setAutoCommit(false);
modifySales = cn.prepareStatement(updateString);
modifyTotal = cn.prepareStatement(updateStatement);
modifySales.setInt(1, 104);
modifySales.setString(2, "Colombian" );
modifySales.executeUpdate();
modifyTotal.setInt(1, 103);
modifyTotal.setString(2, "French_Roast_Decaf");
modifySales.setInt(1, 100);
modifySales.setString(2, "French_Roast");
modifySales.executeUpdate();
modifySales.setString(2, "Espresso");
modifySales.executeUpdate();
modifyTotal.executeUpdate();
cn.commit();

```

As learnt earlier, once auto-commit mode is disabled, no SQL statements are committed until the `commit()` method is explicitly called. Here, the two `PreparedStatement` instances `modifySales` and `modifyTotal` will be committed together after the call to `commit()` method. Appropriate try-catch blocks should be used to enclose the entire code.

→ Step 3: Use SavePoint

The third step is to use `SavePoint` in the transaction as shown in Code Snippet 27. This time a different table is used to demonstrate `SavePoint` usage.

Code Snippet 27:

```

// Create an instance of Statement object
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO COFFEES (COF_NAME) VALUES
(\\"Modern Espresso\\")");
// Create an instance of Statement object
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO COFFEES (COF_NAME) VALUES
(\\"Modern Espresso\\")");
// Set the SavePoint
Savepoint svpt = cn.setSavepoint("SAVEPOINT_1");
rows = st.executeUpdate("INSERT INTO COFFEES (NAME)
VALUES (\\"Arabica\\")");
cn.rollback(svpt);

```

The code inserts a row into a table, sets the `Savepoint` `svpt`, and then, inserts a second row. When the transaction is later rolled back to `svpt` based on some condition, the second insertion is

undone, but the first insertion remains intact. Thus, when the transaction is committed, only the row containing Modern Espresso will be added to COFFEES.

The method `cn.releaseSavepoint()` takes a `Savepoint` object as a parameter and removes it from the current transaction. Any reference to the savepoint after it is removed causes an `SQLException`.

→ Step 4: Close the Transaction

The last step is to close or end the transaction. A transaction can end either with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction.

Code Snippet 28 demonstrates how to close the transaction.

Code Snippet 28:

```
// End the transaction  
.  
.  
cn.rollback(svpt);  
OR  
.  
.  
cn.commit();
```

7.5 JDBC 4.3 Features

The JDBC 4.3 update was released with Java 9 in 2017. Signaling through `beginRequest()` and `endRequest()` methods on `Connection` is one of the new features in this version. In simple words, these methods or APIs are used by the connection pool to indicate to the driver that a connection has been checked out of the pool (`beginRequest`) or checked back into the pool (`endRequest`).

The requesting driver is an independent unit of work and is at the starting point on this connection. Through these APIs/methods, drivers receive information about request boundaries. The driver can ping the database, replace the connection without another driver for load balancing, and clear out dirty states.

Other new features introduced in JDBC 4.3 are as follows:

→ Added support for Sharding

→ Added interfaces:

- `java.sql.ConnectionBuilder`
- `java.sql.ShardigKey`
- `java.sql.ShardingKeyBuilder`
- `javax.sql.XAConnectionBuilder`
- `javax.sql.PooledConnectionBuilder`

7.6 RowSet

RowSet is an interface in the new standard extension package `javax.sql.rowset` and is derived from the `ResultSet` interface. It typically contains a set of rows from a source of tabular data like a result set. It can be configured to connect to and read/write data from a JDBC data source. A JDBC RowSet object is easier to use than a result set.

7.6.1 Different Types of RowSets

RowSets are classified depending on the duration of their connection to the database. Therefore, a RowSet can be either connected or disconnected. A connected RowSet object uses a JDBC driver to connect to a relational database. This connection is maintained throughout the lifespan of the RowSet object.

A disconnected RowSet object connects to a data source only to read data from a `ResultSet` or write data back to the data source. On completion of the read/write operation the RowSet object disconnects from the data source.

Figure 7.3 displays the connected and disconnected RowSets.

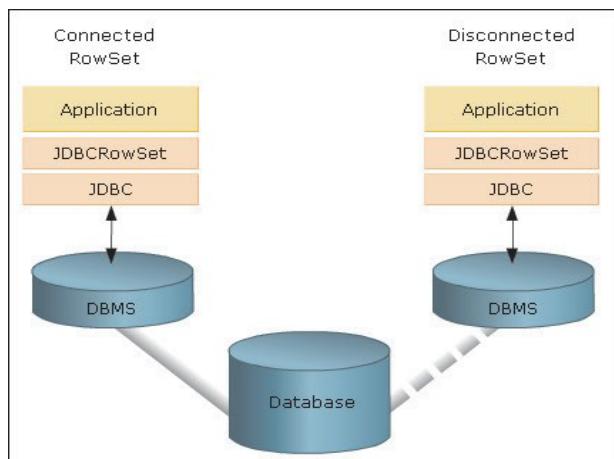


Figure 7.3: Connected and Disconnected RowSets

Some of the interfaces that extend from the `RowSet` interface are as follows:

- `JdbcRowSet`
- `CachedRowSet`
- `WebRowSet`
- `JoinRowSet`
- `FilteredRowSet`

Figure 7.4 shows the RowSet hierarchy.

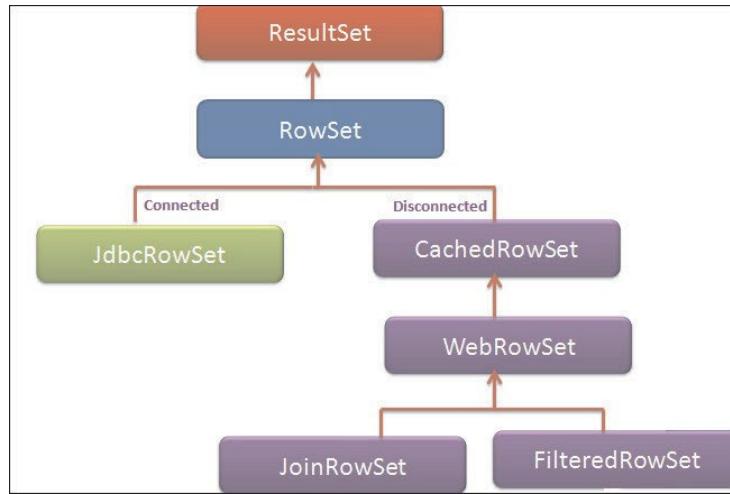


Figure 7.4: RowSet Hierarchy

RowSet 1.1 API includes following classes to construct instances of RowSet:

- ➔ javax.sql.rowset.RowSetProvider
- ➔ javax.sql.rowset.RowSetfactory

The `javax.sql.RowSetProvider` class is used to create a `RowsetFactory` object as shown:

```
RowSetFactory rowsetFactory = RowSetProvider.newFactory();
```

The `RowSetFactory` interface includes following methods to create various RowSet implementations:

- ➔ `createCachedRowSet()`
- ➔ `createFilteredRowSet()`
- ➔ `createJdbcRowSet()`
- ➔ `createJoinRowSet()`
- ➔ `createWebRowSet()`

7.6.2 Implementation of a Disconnected RowSet

A disconnected RowSet stores its data in memory and operates on that data rather than directly operating on the data in the database.

A `CachedRowSet` object is an example of a disconnected RowSet object. These objects store or cache data in memory and operate on this data rather than manipulate data stored in the database. The `CachedRowSet` is the parent interface for all disconnected RowSet objects. A `CachedRowSet` object generally derives its data from a relational database but, it is also capable of retrieving and storing data from a data source which stores data in a tabular form.

A `CachedRowSet` object includes all the capabilities of a `JdbcRowSet` object. In addition, it can perform the following:

- ➔ Connect to a data source and execute a query.

- ➔ Read the data from the resulting ResultSet object and get populated.
- ➔ Manipulate data when disconnected.
- ➔ Reconnect to the data source to write the changes back to it.
- ➔ Resolve conflicts with the data source, if any.

The `CachedRowSet` object can get data from a relational database or from any other data source that stores its data in a tabular format. A key column must be set before data can be saved to the data source.

From Java 9.0 onwards, it is recommended to use `RowSetProvider` to access a `RowSetFactory` to produce an instance of an implementation of `CachedRowSet`.

Code Snippet 29 shows an example of this.

Code Snippet 29:

```
CachedRowSet crs =
RowSetProvider.newFactory().createCachedRowSet();
```

The `username`, `password`, `url`, and `command` properties are set before the `execute()` method is invoked. This helps connect to a data source and select the data as illustrated in Code Snippet 30.

Code Snippet 30:

```
import javax.sql.rowset.RowSetProvider;
import java.sql.SQLException;
import javax.sql.rowset.CachedRowSet;
public class CachedRowSetDemo {
public static void main(String[] args) {
try {
CachedRowSet crs =
RowSetProvider.newFactory().createCachedRowSet();
crs.setUsername("root"); crs.setPassword("");
crs.setUrl("jdbc:mysql://127.0.0.1:3306/test");
crs.setCommand("SELECT * FROM Employees");
crs.execute();
while (crs.next()) {
System.out.println("Emp No: "+crs.getInt(1));
}
} catch (SQLException se) {
System.out.println("error: " + se.getMessage());
}
}
}
```

A CachedRowSet object cannot be populated with data until the `username`, `password`, `url`, and `datasourceName` properties for the object are set. These properties can be set by using the appropriate setter methods of the `CachedRowSet` interface. However, the `RowSet` is still not usable by the application. For the `CachedRowSet` object to retrieve data, the `command` property must be set. This can be done by invoking the `setCommand()` method and passing the SQL query to it as a parameter.

Code Snippet 31 sets the `command` property with a query that produces a `ResultSet` object containing all the data in the table **STUDENT**.

Code Snippet 31:

```
crs.setCommand("select * from STUDENT");
```

If any updates are to be made to the `RowSet` object then, the key columns must be set. The key columns help to uniquely identify a row, as a primary key does in a database. Each `RowSet` object has key columns set for it, similar to a table in a database which has one or more columns set as primary keys for it.

Code Snippet 32 demonstrates how key columns are set for the `CachedRowSet` object.

Code Snippet 32:

```
CachedRowSet crsStudent =
    RowSetProvider.newFactory().createCachedRowSet();
int[] keyColumns = {1, 2};
crsStudent.setKeyColumns(keyColumns);
```

Key columns are used internally, hence, it is of utmost importance to ensure that these columns can uniquely identify a row in the `RowSet` object.

The `execute()` method is invoked to populate the `RowSet` object. The data in the `RowSet` object is obtained by executing the query in the `command` property. The `execute()` method of a disconnected `RowSet` performs many more functions than the `execute()` method of a connected `RowSet` object.

A `CachedRowSet` object has a `SyncProvider` object associated with it. This `SyncProvider` object provides a `RowSetReader` object which on invocation of the `execute()` method, establishes a connection with the database using the `username`, `password`, and `URL`, or `datasourceName` properties set earlier. The reader object then executes the query set in the `command` property and the `RowSet` object is populated with data.

The reader object then closes the connection with the datasource and the `CachedRowSet` object can be used by the application.

Updation, insertion, and deletion of rows is similar to that in a `JdbcRowSet` object. However, the changes made by the application to the disconnected `RowSet` object must be reflected back to the datasource. This is achieved by invoking the `acceptChanges()` method on the `CachedRowSet`

object. Like the `execute()` method, this method also performs its operations behind the scenes. The `SyncProvider` object also has a `RowSetWriter` object which opens a connection to the database, writes the changes back to the database and then finally, closes the connection to the database.

7.6.3 Using CachedRowSet Object

A row of data can be updated, inserted, and deleted in a `CachedRowSet` object. Changes in data are reflected on the database by invoking the `acceptChanges()` method.

→ Update

Updating a record from a `RowSet` object involves navigating to that row, updating data from that row and finally updating the database.

Code Snippet 33 illustrates the updation of row. A table named `EmpLeave` is assumed to be existing for this code. Ensure that this code is enclosed in an appropriate `try-catch` block.

Code Snippet 33:

```
CachedRowSet crs =
    RowSetProvider.newFactory().createCachedRowSet();
crs.setUsername("root");
crs.setPassword("");
crs.setUrl("jdbc:mysql://127.0.0.1:3306/test");
crs.setCommand("SELECT * FROM EmpLeave");
crs.execute();
crs.next();
if (crs.getInt("EMP_NO") == 111) {
    int currentQuantity = crs.getInt("BAL_LEAVE") + 1;
    System.out.println("Updating balance leave to " + currentQuantity);
    crs.updateInt("BAL_LEAVE", currentQuantity + 1);
    crs.updateRow();
    // Synchronizing the row back to the DB
    Connection con = DriverManager.
    getConnection("jdbc:mysql://127.0.0.1:3306/test", "root", "");
    con.setAutoCommit(false);
    crs.acceptChanges(con);
}
```

In Code Snippet 33, the `BAL_LEAVE` column is updated. The `updateInt()` method is used to change the value of the individual cell (column) as shown in the code. The `updateRow()` method is invoked to update the memory. The `acceptChanges()` method saves the changes to the data source.

→ Insert

To insert a record into the `CachedRowSet` object, the `moveToInsertRow()` method is invoked. The current cursor position is remembered, and the cursor is then positioned on an insert row. The insert row is a special buffer row provided by an updatable result set for constructing a new row. When the cursor is in this row only the `update`, `get`, and `insertRow()` methods can be called.

All the columns must be given a value before the `insertRow()` method is invoked.

The `insertRow()` method inserts the newly created row in the result set.

The `moveToCurrentRow()` method moves the cursor to the remembered position.

➔ Delete

Deleting a row from a `CachedRowSet` object is simple. Code Snippet 34 illustrates this.

Code Snippet 34:

```
while (crs.next()) {  
    if (crs.getInt("EMP_ID") == 12345) {  
        crs.deleteRow();  
        break;  
    }  
}
```

Code Snippet 34 deletes the row containing the employee number as 12345 from the row set. The cursor is moved to the appropriate row and the `deleteRow()` method deletes the row from the row set. To ensure the code runs successfully, ensure that autocommit is set to false and the table has a primary key set.

➔ Retrieve

A `CachedRowSet` object is scrollable, which means that the cursor can be moved forward and backward by using the `next()`, `previous()`, `last()`, `absolute()`, and `first()` methods. Once the cursor is on the desired row, the getter methods can be invoked on the `RowSet` to retrieve the desired values from the columns.

7.7 Summary

- A ResultSet object maintains a cursor pointing to its current row of data.
- Updatable ResultSet is the ability to update rows in a result set using methods in the Java programming language rather than SQL commands.
- A stored procedure is a group of SQL statements.
- A batch update is a set of multiple update statements that is submitted to the database for processing as a batch.
- A transaction is a set of one or more statements that are executed together as a unit, so either all the statements are executed, or none of the statements is executed.
- RowSet is an interface that is derived from the ResultSet interface.
- A JdbcRowSet is the only implementation of a connected RowSet.

7.8 Check Your Progress

1. Which of these statements about Scrollable ResultSet are true?

(A)	In Scrollable ResultSet, the cursor is positioned on the first row.
(B)	A default ResultSet object is not updatable and has a cursor that moves forward only.
(C)	The ResultSet should be compulsorily closed after a COMMIT statement.
(D)	Holdability refers to the ability to check whether the cursor stays open after a COMMIT.
(E)	The createStatement method has two arguments namely, resultSetType and resultSetConcurrency.

- | | |
|-----------------|-----------------|
| (A) A, B, and C | (C) B, C, and E |
| (B) B, D, and E | (D) A, D, and E |

2. Which of these statements about ResultSet constant values are true?

(A)	TYPE_SCROLL_INSENSITIVE is the default type of result set.
(B)	In TYPE_FORWARD_ONLY result set, the cursor can only be used to process from the beginning of a ResultSet to the end of it.
(C)	In TYPE_SCROLL_INSENSITIVE result set, the SQL queries applied on a particular field will have a direct impact on the ResultSet.
(D)	The TYPE_SCROLL_SENSITIVE type of cursor is sensitive to changes made to the database while it is open.
(E)	The TYPE_FORWARD_ONLY type of result set is not scrollable, not positionable, and not sensitive.

3. Which of the following statements about RowSet are true?

(A)	A RowSet contains a set of rows of data.
(B)	A RowSet has to be made scrollable and updatable at the time of creation.
(C)	A connected RowSet can read data from a non-relational database source also.
(D)	Scrollability and Updatability of a RowSet is independent of the JDBC driver.

4. Which of these statements about Updatable ResultSet are true?

(A)	Updatable ResultSet is the ability to update rows in a result set using SQL commands, rather than using methods in the Java programming language.
(B)	The UpdateXXX() method of ResultSet is used to change the data in an existing row.
(C)	Updatability in a result set is associated with concurrency in database access because you cannot have multiple write locks concurrently.
(D)	In CONCURRENCY.READ_ONLY type of result set, the updates, inserts, and deletes can be performed on the result set but it cannot be copied to the database.
(E)	The concurrency type of a result set determines whether it is updatable or not.

(A) A, C
(B) B

(C) C, E
(D) D, E

7.8.1 Answers

1. A
2. B
3. B
4. D
5. C

Try It Yourself

1. Create a Java program that uses a scrollable result set to navigate through a database table. Demonstrate how to move both forward and backward through the result set.

Hint: A scrollable result set allows you to move both forward and backward through query results. You can achieve this by setting the `ResultSet` to be scrollable using `stmt.executeQuery("SELECT * FROM yourtable", ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);` and then, using methods such as `next()`, `previous()`, `absolute()`, or `relative()` to navigate.

2. Write a Java program to call a stored procedure using JDBC. Include the necessary steps for registering the stored procedure and passing parameters.

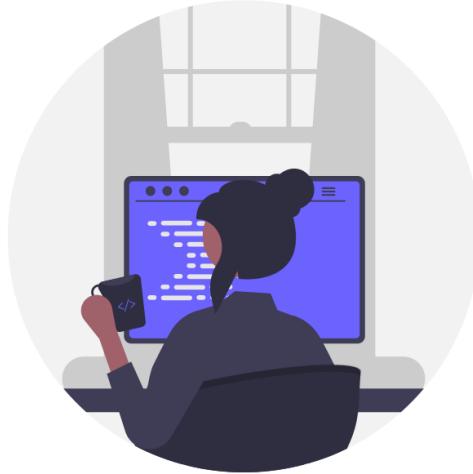
Hint: To call a stored procedure using JDBC, you first register it with `CallableStatement` and then, use methods such as `setXXX()` to set input parameters and `execute()` to invoke the procedure.

3. Describe the steps involved in updating records in a database using JDBC. Include handling both single-record updates and batch updates.

Hint: Updating records involves creating a `PreparedStatement`, setting parameters, executing updates using `executeUpdate()` and handling exceptions. For batch updates, use `addBatch()` and `executeBatch()` methods.

4. Explain the steps for implementing transactions in JDBC. Include the concepts of commit, rollback, and SavePoint.

Hint: Implementing transactions in JDBC involves setting auto-commit to false, creating `SavePoint`, and using `commit()` and `rollback()` methods to control the transaction's behavior.



Session 8

Design Patterns and Other Advanced Features

Welcome to the Session, **Design Patterns and Other Advanced Features**.

In simplest terms, a design pattern can be thought of as a description of a solution to a frequently observed problem. It can be considered as a template or a best practice suggested by expert programmers for commonly occurring problems. The session explains design patterns in detail. This session describes the internationalization and localization process that makes an application serve users in multiple different languages and are suitable for global market. It describes various methods that Java application can use to handle different languages, number formats, and so on. Internationalized applications require meticulous planning, failing which re-engineering of the application can be costly. This session also discusses advanced concurrency and parallelism features provided by Java.

In this Session, you will learn to:

- Explain design patterns
- Describe delegation, composition, and aggregation
- Describe internationalization and localization
- Explain the enhancements of `java.util.concurrent` package
- Describe atomic operations with the new set of classes of the `java.util.concurrent.atomic` package
- Explain the new features of `ForkJoinPool`

8.1 Design Patterns

A design pattern is a clearly defined solution to problems that occur frequently. It can be considered as a template or a best practice suggested by expert programmers. So, if an experienced developer educates another developer about a particular factory pattern being used to solve a problem, the other developer can precisely understand how to deal with a similar problem. A design pattern is a great help to inexperienced developers. They can study patterns and related problems and learn good details about software design. This reduces the learning curve. The proper use of design patterns

results in increased code maintainability.

Design pattern are based on the fundamental principles of object-oriented design. A design pattern is not an implementation, nor is it a framework. It cannot be installed using code. As of today, there are certain standard and popularly used design patterns that have been developed after long periods of research and trial and error by software developers.

8.2 Types of Patterns

Table 8.1 lists different types of design patterns.

Pattern Category	Description	Types
Creational Patterns	They offer ways to create objects while hiding the logic of creation logic, instead of instantiating objects using the new operator.	Singleton Pattern Factory Pattern Abstract Factory Pattern Builder Pattern Prototype Pattern
Structural Patterns	They are related to class and object composition. Interfaces are composed using the concept of inheritance and new ways are defined to compose objects to get different functionalities.	Adapter Pattern Composite Pattern Proxy Pattern Flyweight Pattern Facade Pattern Bridge Pattern Decorator Pattern
Behavioral Patterns	They are related to communication between various objects.	Template Method Pattern Mediator Pattern Chain of Responsibility Pattern Observer Pattern Strategy Pattern Command Pattern State Pattern Visitor Pattern Iterator Pattern Memento Pattern

Table 8.1: Types of Design Patterns

8.2.1 Singleton Pattern

Certain class implementations can be instantiated only once. The singleton design pattern provides complete information on such class implementations. To implement this, usually, a static field is created representing the class. The object that the static field references can be created at the time when the class is initialized or the first time the `getInstance()` method is invoked.

The constructor of a class using the singleton pattern is declared as private to prevent the class from being instantiated.

It is recommended to use singleton classes to concentrate access to resources into a single class instance.

To implement a singleton design pattern, perform following steps:

1. Use a static reference to point to the single instance.
2. Then, add a single private constructor to the singleton class.
3. Next, a public factory method is declared static to access the static field declared in Step 1. A factory method is a method that instantiates objects. Similar to the concept of a factory that manufactures products, the job of the factory method is to manufacture objects.

Note: A public factory method returns a copy of the singleton reference.

4. Use the static `getInstance()` method to get the instance of the singleton.

Consider following points when implementing the singleton design pattern:

- The reference is finalized so that it does not reference a different instance.
- The private modifier allows only same class access and restricts attempts to instantiate the singleton class.
- The factory method provides greater flexibility. It is commonly used in singleton implementations.
- The singleton class usually includes a private constructor that prevents a constructor to instantiate the singleton class.
- To avoid using the factory method, a public variable can be used at the time of using a static reference.

Code Snippet 1 illustrates the implementation of the singleton design pattern.

Code Snippet 1:

```
class SingletonExample {  
    private static SingletonExample singletonExample = null;  
    private SingletonExample() {  
    }  
    public static SingletonExample getInstance() {  
        if (singletonExample == null) {  
            singletonExample = new SingletonExample();  
        }  
        return singletonExample;  
    }  
    public void display() {  
        System.out.println("Welcome to Singleton Design Pattern");  
    }  
}
```

In Code Snippet 1, note the following:

- The `SingletonExample` class contains a private static `SingletonExample`

field.

- There is a `private` constructor. Therefore, the class cannot be instantiated by outside classes.
- The public static `getInstance()` method returns the only `SingletonExample` instance. If the instance does not exist, the `getInstance()` method creates it.
- There is a public `sayHello()` method that can test the singleton.

Note: `SingletonExample` class is an example of a typical singleton class.

Code Snippet 2 includes the `SingletonTest` class that calls the static `SingletonExample.getInstance()` method to get the `SingletonExample` singleton class.

Code Snippet 2:

```
public class SingletonTest {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        SingletonExample singletonExample = SingletonExample.getInstance();  
        singletonExample.display();  
    }  
}
```

In Code Snippet 2, note the following:

- The `display()` method is called on the singleton class.
- The output of the program is "Welcome to Singleton Design Pattern".

8.3 Interfaces in Design Patterns

Consider a scenario where following programs are created to automate certain tasks of an automobile:

- A program that stops the vehicle on red light
- A program that accelerates the vehicle
- A program that turns the vehicle in different directions

There can be such many other programs for automobile automation.

Now, all these programs required not be made by a single individual. Each program can be owned by different programmers or organizations.

To integrate all these programs from different sources on an automobile, there has to be a medium that describes how a software interacts. This medium is the interface. So, when the programmers decide to write a code for a similar target (such as automation of automobile), they comply to this interface.

In Java, interfaces include constant fields. They can be used as reference types. In addition, they are important components of many design patterns.

Java uses interfaces to define type abstraction. Outlining abstract types is a powerful feature of Java.

Following are benefits of abstraction:

- **Vendor-specific Implementation:** Developers define the methods for the `java.sql` package. The communication between the database and the `java.sql` package occurs using the methods. However, the implementation is vendor-specific.
- **Tandem Development:** Based on the business API, the application's UI and the business logic can be developed simultaneously.
- **Easy Maintenance:** Improved classes can replace the classes with logic errors anytime.

Note: Java also uses abstract classes to define type abstraction.

Code Snippet 3 shows an interface declaration.

Code Snippet 3:

```
public interface IAircraft {  
    public int passengerCapacity = 400;  
    // method signatures  
    void fly();  
    ....  
    // more method signatures  
}  
dateFormatter = DateFormat.getDateInstance(DateFormat.MEDIUM,  
locale);  
}  
}
```

Only constant fields are allowed in an interface. A field is implicitly `public`, `static`, and `final` when an interface is declared. As a good design practice, it is recommended to distribute constant values of an application across many classes and interface.

Defining a new interface defines a new reference data type. Consider following points regarding reference types:

- Interface names can be used anywhere. In addition, any other data type name can be used.
- The `instanceOf` operator can be used with interfaces.
- If a reference variable is defined whose type is an interface, then any object assigned to it must be an instance of a class that implements the interface.
- Interfaces implicitly include all the methods from `java.lang.Object`.
- If an object includes all the methods outlined in the interface but does not implement the interface, then the interface cannot be used as a reference type for that object.

8.3.1 Difference between Class Inheritance and Interface Inheritance

A class can extend a single parent class whereas, it can implement multiple interfaces. When a class extends a parent class, only certain functionalities can be overridden in the inherited class. On the other hand, when a class inherits an interface, it implements all functionalities of the interfaces.

A class is extended because certain classes require detailed implementation based on the superclass. However, all the classes require some of the methods and properties of the superclass. On the other hand, an interface is used when there are multiple implementations of the same functionality.

Table 8.2 provides the comparison between an interface and an abstract class.

Interface	Abstract Class
Inheritance of several interfaces by a class is supported.	Inheritance of only one abstract class by a class is supported.
Requires more time to find the actual method in the corresponding child classes.	Requires less time to find the actual method in the corresponding child classes.
Best used when various implementations only share method signatures.	Best used when various implementations use common behavior or status.

Table 8.2: Comparison between an Interface and an Abstract Class

8.3.2 Extending Interfaces

It is recommended to specify all uses of the interface right from the beginning. However, this is not always possible. In such an event, more interfaces can be created later. This way, interfaces can be used to extend interfaces.

Code Snippet 4 creates an interface called **IVehicle**.

Code Snippet 4:

```
public interface IVehicle {  
    int getMileage(String s);  
    . . .  
}
```

Code Snippet 5 shows how a new interface can be created that extends **IVehicle**.

Code Snippet 5:

```
public interface IAutomobile extends IVehicle {  
    boolean accelerate(int i, double x, String s);  
}
```

In Code Snippet 5, an **IAutomobile** interface is created that extends **IVehicle**. Users can now either use the old interface or upgrade to the new interface. If a class implements the new interface, it must override all the methods from **IVehicle** and **IAutomobile**.

8.3.3 Implementation of IS-A and a HAS-A Relationships

It is a concept based on class inheritance or interface implementation. An IS-A relationship displays class hierarchy in case of class inheritance. For example, if the class Ferrari extends the class **Car**, the statement Ferrari IS-A **Car** is true. Figure 8.1 illustrates an IS-A relationship.

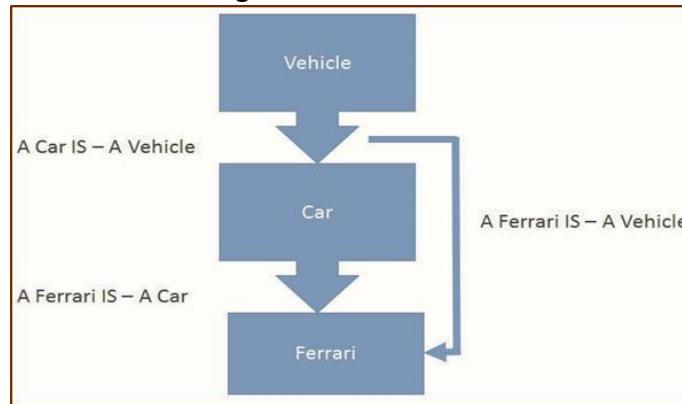


Figure 8.1: IS-A Relationship

The IS-A relationship is also used for interface implementation. This is done using keyword implements or extends. An HAS-A relationship or a composition relationship uses instance variables that are references to other objects, such as a Ferrari includes an Engine.

Figure 8.2 illustrates the example.

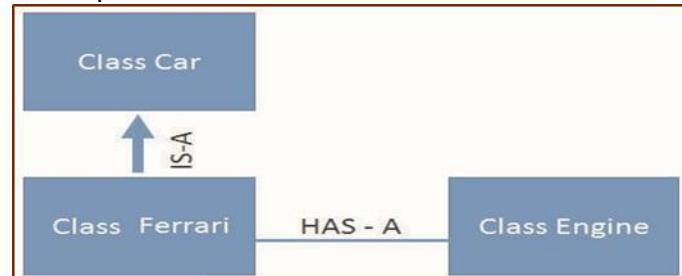


Figure 8.2: HAS-A Relationship Example

8.4 Data Access Object (DAO) Design Pattern

The DAO pattern is used when an application is created that must persist its data. The DAO pattern involves a technique for separating the business logic from persistence logic, making it easier to implement and maintain an application. The advantage of the DAO approach is that it makes it flexible to change the persistence mechanism of the application without changing the entire application, as the data access layer would be separate and unaffected.

The DAO pattern uses the following:

- **DAO Interface:** This defines the standard operations for a model object. In other words it defines the methods used for persistence.
- **DAO Concrete Class:** This implements the DAO interface and retrieves data from a data source, such as a database.
- **Model Object or Value Object:** This includes the get/set methods that store data retrieved by the DAO class.

As the name suggests, DAOs can be used with any data objects, not necessarily databases. Thus, if your data access layer comprises data stores of XML files, DAOs can still be useful there.

Some of various types of data objects that DAOs can support are as follows:

- **Memory based DAOs:** These represent temporary solutions.
- **File based DAOs:** These may be required for an initial release.
- **JDBC based DAOs:** These support database persistence.
- **Java persistence API based DAOs:** These also supports database persistence.

Figure 8.3 shows the structure of a DAO design pattern that will be created.

Following points are to be noted in Figure 8.3:

- **Book** object will act as a Model or Value Object
- **BookDao** is the DAO Interface
- **BookDaoImpl** is the concrete class that implements the DAO interface
- **DAOPatternApplication** is the main class. It will use **BookDao** to display the use of the DAO pattern

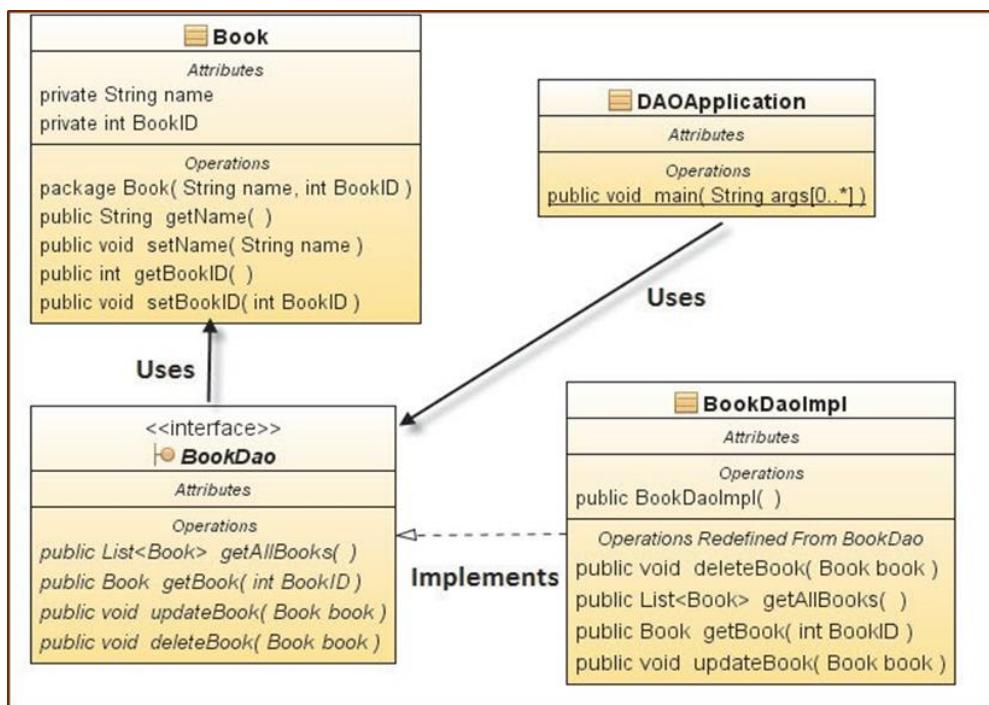


Figure 8.3: Structure of DAO Design Pattern

Based on Figure 8.3, following code snippets will create a DAO design pattern. First, the Model or Value object is created which will store the book information. Code Snippet 6 illustrates this.

Code Snippet 6:

```

class Book {
private String name;
private int BookID;
Book(String name, int BookID) {
this.name = name;
this.BookID = BookID;
}
  
```

```
}

public String getName() {
    return name;
}

public void setName(String name) {
this.name = name;
}

public int getBookID() {
    return BookID;
}

public void setBookID(int BookID) {
this.BookID = BookID;
}

}
```

Code Snippet 6 defines a **Book** class having a constructor and get/set methods.

Code Snippet 7 creates a DAO interface that makes use of this class.

Code Snippet 7:

```
interface BookDao {
public java.util.List<Book> getAllBooks();
public Book getBook(int BookID);
public void updateBook(Book book);
public void deleteBook(Book book);
}
```

Code Snippet 8 creates a class that implements the interface.

Code Snippet 8:

```
class BookDaoImpl implements BookDao {
// list is working as a database
java.util.List<Book> booksList;
public BookDaoImpl(){
booksList = new java.util.ArrayList<Book>();
Book bookObj1 = new Book("Anna",1);
Book bookObj2 = new Book("John",2);
booksList.add(bookObj1);
booksList.add(bookObj2);
}
@Override
public void deleteBook(Book book) {
booksList.remove(book.getBookID());
System.out.println("Book: Book ID " + book.getBookID()
+", deleted from database");
}
// retrieve list of booksList from the database
@Override
public java.util.List<Book> getAllBooks() {
return booksList;
```

```

}
@Override
public Book getBook(int BookID) {
    return booksList.get(BookID);
}
@Override
public void updateBook(Book book) {
    booksList.get(book.getBookID()).setName(book.getName());
    System.out.println("Book: Book ID " + book.getBookID() + ", updated
in the database");
}
}

```

Code Snippet 9 displays the use of the DAO pattern.

Code Snippet 9:

```

public class DAOPatternApplication {
    public static void main(String[] args) {
        BookDao bookDao = new BookDaoImpl();
        System.out.println("Book List:");
        //print all books
        for (Book book : bookDao.getAllBooks()) {
            System.out.println("\nBookID : " + book.getBookID() + ", Name :
" + book.getName() + " ");
        }
        //update book
        Book book = bookDao.getAllBooks().get(0);
        book.setName("Harry Long");
        bookDao.updateBook(book);
        //get the book
        bookDao.getBook(0);
        System.out.println("Book: [BookID : " + book.getBookID() + ", Name
:" + book.getName() + " ]");
    }
}

```

Figure 8.4 displays the output of the DAO design pattern.

```

Book List:

BookID : 1, Name : Anna

BookID : 2, Name : John
Book: Book ID 1, updated in the database
Book: [BookID : 1, Name : Harry Long ]
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 8.4: The DAO Design Pattern – Output

8.5 Factory Design Pattern

Factory pattern is one of the commonly used design patterns in Java. It belongs to the creational

pattern category and provides many ways to create an object. This pattern does not perform direct constructor calls when invoking a method. It prevents the application from being tightly coupled to a specific DAO implementation.

In factory pattern, note the following:

- The client is not aware of the logic that helps create object.
- It uses a common interface to refer to the newly created object.

Consider a scenario where certain automobile classes required to be designed. A general interface **Vehicle** with a common method **move()** will be created. Classes named **Car** and **Truck** will implement this interface respectively. A **VehicleFactory** class is created to get a **Vehicle** type and based on which the respective objects will be returned to the calling program.

Code Snippet 10 creates a common interface for implementing the Factory pattern.

Code Snippet 10:

```
interface Vehicle {  
void move();  
}
```

Code Snippet 11 creates a class that implements the interface.

Code Snippet 11:

```
class Car implements Vehicle {  
@Override  
public void move() {  
System.out.println("Inside Car::move() method.");  
}  
}
```

Code Snippet 12 creates another class that implements the interface.

Code Snippet 12:

```
class Truck implements Vehicle {  
@Override  
public void move() {  
System.out.println("Inside Truck::move() method.");  
}  
}
```

Code Snippet 13 creates a factory to create object of concrete class based on the information provided.

Code Snippet 13:

```
class VehicleFactory {  
//use getVehicle method to get object of type Vehicle  
public Vehicle getVehicle(String vehicleType) {  
if(vehicleType == null) {
```

```

return null;
}
if(vehicleType.equalsIgnoreCase("Car")) {
return new Car();
} else if(vehicleType.equalsIgnoreCase("Truck")) {
return new Truck();
}
return null;
}
}

```

Code Snippet 14 uses the factory pattern to get objects of concrete classes by passing the **Vehicle** type information.

Code Snippet 14:

```

public class FactoryPatternExample {
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
VehicleFactory vehicleFactory = new VehicleFactory();
//get an object of Car and call its move method.
Vehicle carObj = vehicleFactory.getVehicle("Car");
//call move method of Car
carObj.move();
//get an object of Truck and call its move method.
Vehicle truckObj = vehicleFactory.getVehicle("Truck");
//call move method of truck truckObj.move();
}
}

```

In the code, the correct object is created and based on the type of object, the appropriate **move ()** method is invoked.

Figure 8.5 depicts the factory pattern diagram.

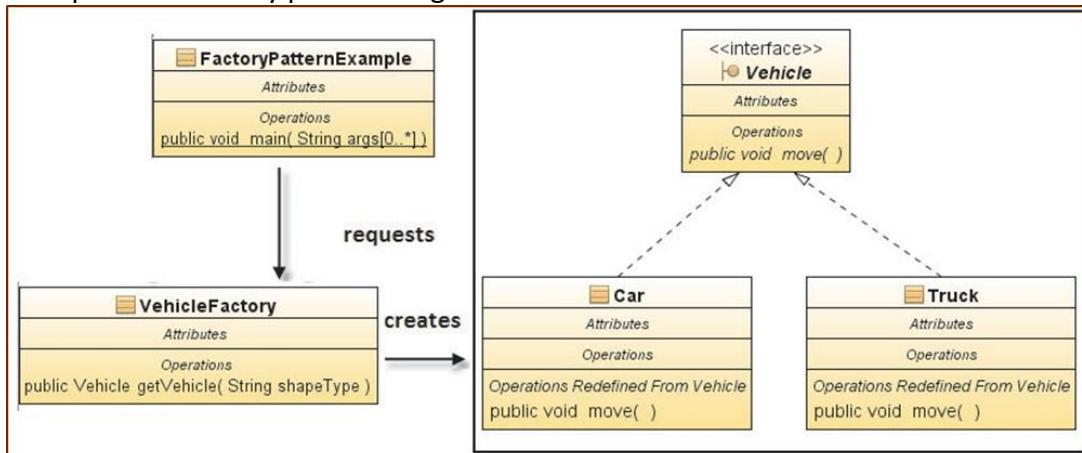


Figure 8.5: Factory Design Pattern

Figure 8.6 displays the output of Code Snippet 14.

```
run:  
Inside Car::move() method.  
Inside Truck::move() method.  
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 8.6: Factory Design Pattern - Output

8.6 Delegation

Besides the standard design patterns, one can also use the delegation design pattern. In Java, delegation means using an object of another class as an instance variable, and forwarding messages to the instance. Therefore, delegation is a relationship between objects. Here, one object forwards method calls to another object, which is called its delegate.

Delegation is different from inheritance. Unlike inheritance, delegation does not create a superclass. In this case, the instance is that of a known class. In addition, delegation does not force to accept all the methods of the superclass. Delegation supports code reusability and provides runtime flexibility. Note- Runtime flexibility means that the delegate can be easily changed at runtime.

Code Snippet 15 displays the use of delegation using a real-world scenario.

Code Snippet 15:

```
interface Employee {  
    public Result sendMail();  
}  
  
public class Secretary implements Employee {  
    public Result sendMail() {  
        Result myResult = new Result();  
        return myResult;  
    }  
}  
  
public class Manager implements Employee {  
    private Secretary secretary;  
    public Result sendMail() {  
        return secretary.sendMail();  
    }  
}
```

In Code Snippet 15, the **Manager** instance forwards the task of sending mail to the **Secretary** instance who in turn forwards the request to the Employee. On a casual observation, it may seem that the manager is sending the mail but in reality, it is the employee doing the task. Thus, the task request has been delegated.

8.7 Composition and Aggregation

Composition refers to the process of composing a class from references to other objects. This way, references to the main objects are fields of the containing object. Composition forms the building blocks for data structures. Programmers can use object composition to create more complex objects.

Aggregation is a similar concept with some differences. In aggregation, one class owns another class. In composition, when the owning object is destroyed, so are the objects within it but in aggregation, this is not true.

For example, an organization comprises various departments and each department has a number of managers. If the organization shuts down, these departments will no longer exist, but the managers will continue to exist. Therefore, an organization can be seen as a composition of departments, whereas departments have an aggregation of managers. In addition, a manager could work in more than one department (if he/she is handling multiple responsibilities), but one department cannot exist in more than one organization.

Composition and aggregation are design concepts and not actual patterns. Code Snippet 16 demonstrates a minimal example of composition.

Code Snippet 16:

```
// Composition class House
{
    // House has door.
    // Door is built when House is built,
    // it is destroyed when House is destroyed.
    private Door dr;
}
```

To implement object composition, perform following steps:

1. Create a class with reference to other classes.
2. Add the same signature methods that forward to the referenced object.
Consider an example of a student attending a course. The student 'has a' course. The composition for the **Student** and **Course** classes is depicted in Code Snippets 17 and 18.

Code Snippet 17:

```
package composition;
public class Course {
    private String title;
    private long score;
    private int id;
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

```

public long getScore() {
    return score;
}
public void setScore(long score) {
    this.score = score;
}
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}

```

The **Course** class is then used in the **Student** class as shown in Code Snippet 18.

Code Snippet 18:

```

package composition;
public class Student {
    //composition has-a relationship
    private Course course;
    public Student(){
        this.course = new Course();
        course.setScore(1000);
    }
    public long getScore() {
        return course.getScore();
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)  {
        Student p = new Student();
        System.out.println(p.getScore());
    }
}

```

8.8 Internationalization and Localization

In the last few decades, with the popularity of Internet, globalization of software products has become an imminent requirement.

The main problems faced in globalization of software products are as follows:

- Not all countries across the world speak or understand English language.
- Symbols for currency vary across countries.
- Date and Time are represented differently in some countries.
- Spelling also varies amongst some countries.

Two possible options for solving the problems faced are as follows:

- **Develop the entire product in the desired language** - This option will mean repetition of coding work. It is a time-consuming process and not an acceptable solution. Development cost will be much higher than the one time cost.
- **Translate the entire product in the desired language** - Successful translation of the source files is very difficult. The menus, labels, and messages of most GUI components are hard-coded in the source code. It is not likely that a developer or a translator will have linguistic as well as coding skills.
Thus, when the input and output operations of an application is made specific to different locations and user preferences, users around the world can use it with ease. This can be achieved using the processes called internationalization and localization. The adaptation is done with extreme ease because there are no coding changes required.

8.8.1 Internationalization

To make an application accessible to the international market, it should be ensured that the input and output operations are specific to different locations and user preferences. The process of designing such an application is called internationalization. Note that the process occurs without any engineering changes.

Java includes built-in support to internationalize applications, called as Java internationalization.

8.8.2 Localization

While internationalization deals with different locations and user preferences, localization deals with a specific region or language. In localization, an application is adapted to a specific region or language.

Locale-specific components are added and text is translated in the localization process. A locale represents a particular language and country.

Primarily, in localization, the user interface elements and documentation are translated. Changes related to dates, currency, and so on are also taken care of. In addition, culturally sensitive data, such as images, are localized. If an application is internationalized in an efficient manner, then it will be easier to localize it for a particular language and character encoding scheme.

8.8.3 ISO Codes and Unicode

In the internationalization and localization process, a language is represented using the alpha-2 or alpha-3 ISO 639 code, such as `es` that represents Spanish. The code is always represented in lower case letters.

A country is represented using the ISO 3166 alpha-2 code or UN M.49 numeric area code. It is always represented in upper case. For example, `ES` represents Spain. If an application is well internationalized, it is easy to localize it for a character encoding scheme.

Code Snippet 19 illustrates the use of Japanese language for displaying a message.

Code Snippet 19:

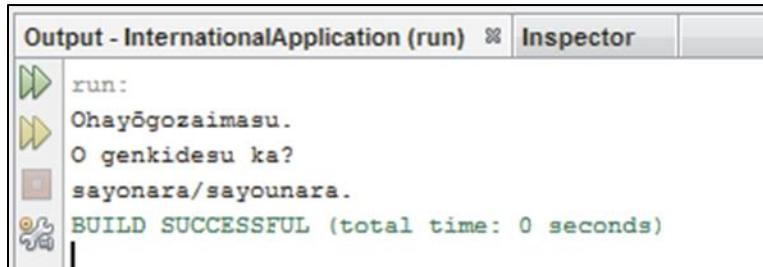
```
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
// TODO code application logic here
String language;
String country;
if (args.length != 2) {
language = new String("en");
country = new String("US");
}
else {
language = new String(args[0]);
country = new String(args[1]);
}
Locale currentLocale;
ResourceBundle messages;
currentLocale = new Locale(language, country);
messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
System.out.println(messages.getString("greetings"));
System.out.println(messages.getString("inquiry"));
System.out.println(messages.getString("farewell"));
}
}
```

In the code, two arguments are accepted to represent country and language. Depending on the arguments passed during execution of the program, the message corresponding to that country and language is displayed. For this, five properties files have been created. You can create a .properties file in NetBeans by using the **File → New → Other** option. If creating a Maven based Java project, the .properties files should be placed under **src\main\resources** path.

The content of the five properties files are as follows:

- **MessagesBundle.properties**
greetings = Hello. farewell = Goodbye. inquiry = How are you?
- **MessagesBundle_de_DE.properties**
greetings = Hallo. farewell = Tschüß. inquiry = Wie geht's?
- **MessagesBundle_en_US.properties**
greetings = Hello. farewell = Goodbye. inquiry = How are you?
- **MessagesBundle_fr_FR.properties**
greetings = Bonjour. farewell = Au revoir.
inquiry = Comment allez-vous?
- **MessagesBundle_ja_JP.properties**
greetings = Ohayōgozaimasu. farewell = sayonara/sayounara. inquiry = Ogenkidesuka?

Figure 8.7 displays the output in Japanese language if the arguments are ja JP.



```
Output - InternationalApplication (run) × Inspector
run:
Ohayōgozaimasu.
O genkidesu ka?
sayonara/sayounara.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 8.7: InternationalApplication - Output

Unicode is a computing industry standard. It is used to uniquely encode characters for various languages in the world using hexadecimal values. In other words, Unicode provides a unique number for every character irrespective of platform, program, or language.

Note: Java uses Unicode as its native character encoding.

Java programs still required to handle characters in other encoding systems. The `String` class can be used to convert standard encoding systems to and from the Unicode system. To indicate Unicode characters that cannot be represented in ASCII, such as ö, you use the `\uXXXX` escape sequence. Each X in the escape sequence is a hexadecimal digit.

Following list defines the terminologies used in the Unicode character encoding:

- **Character:** This represents the minimal unit of text that has semantic value.
- **Character Set:** This represents set of characters that can be used by many languages. For example, the Latin character set is used by English and certain European languages.
- **Coded Character:** This is a character set. Each character in the set is assigned a unique number.
- **Code Point:** This is the value that is used in a coded character set. A code point is a 32-bit int data type. Here, the upper 11 bits are 0 and the lower 21 bits represent a valid code point value.
- **Code Unit:** This is a 16-bit charvalue.
- **Supplementary Characters:** These are the characters that range from U+10000 to U+10FFFF. Supplementary characters are represented by a pair of code point values called surrogates that support the characters without changing the char primitive data type. Surrogates also provide compatibility with earlier Java programs.
- **Basic Multilingual Plane (BMP):** These are the set of characters from U+0000 to U+FFFF. Consider following points for Unicode character encoding:
 - The hexadecimal value is prefixed with the string U+.
 - The valid code point range for the Unicode standard is U+0000 to U+10FFFF.

Table 8.3 shows code point values for certain characters.

Character	Unicode Code Point	Glyph
Latin A	U+0041	A
Latin sharp S	U+00DF	B

Table 8.3: Code Point Values for Certain Characters

8.9 Internationalization Process

If the internationalized source code is observed, notice that the hard-coded English messages are removed. The messages are no longer hard-coded and the language code is specified at runtime, so that the same executable can be distributed worldwide. No recompilation is required for localization.

For internationalization process the steps to be followed are as follows:

- Creating the Properties files
- Defining the Locale
- Creating a ResourceBundle
- Fetching the text from the ResourceBundle class

8.9.1 Creating the Properties Files

A properties file stores information about the characteristics of a program or environment. A properties file is in plain-text format. It can be created with any text editor.

In following example, the properties files store the text that must be translated. Note that before internationalization, the original version of text was hard-coded in the `System.out.println` statements.

The default properties file, **MessagesBundle.properties**, includes following lines:

```
greetings = Hello farewell = Goodbye inquiry = How are you?
```

Since the messages are in the properties file, it can be translated into various languages. No changes to the source code are required.

To translate the message in French, the French translator creates a properties file called **MessagesBundle_fr_FR.properties**, which contains following lines:

```
greetings = Bonjour.  
farewell = Au revoir.  
inquiry = Comment allez-vous?
```

Notice that the values to the right of the equal sign are translated. The keys on the left are not changed. These keys must not change because they are referenced when the program fetches the translated text.

The name of the properties file is important. For example, the name **MessagesBundle_fr_FR.properties** file contains the `fr` language code and the `FR` country code. These codes are also used when creating a `Locale` object.

8.9.2 Defining the Locale

The `Locale` object identifies a particular language and country. A `Locale` is simply an identifier for a particular combination of language and region.

A `java.util.Locale` class object represents a specific geographical, political, or cultural region. Any operation that requires a locale to perform its task is said to be locale-sensitive. These operations

use the `Locale` object to tailor information for the user.

For example, displaying a number is a locale-sensitive operation. The number should be formatted according to the customs and conventions of a user's native country, region, or culture.

A `Locale` object is created using following constructors:

- `public Locale(String language, String country)` - This creates a `Locale` object with the specified language and country. Consider following syntax:

Syntax

```
public Locale(String language, String country)
```

where,

language - is the language code consisting of two letters in lowercase.

country - is the country code consisting of two letters in uppercase.

- `public Locale(String language)` - This creates a `Locale` object with the specified language.

Consider following syntax:

Syntax

```
public Locale(String language)
```

where,

language - is the language code consisting of two letters in lowercase.

Following statement defines a `Locale` for which the language is English and the country is the United States.

```
aLocale = new Locale("en", "US");
```

Code Snippet 20 creates `Locale` objects for the French language for the countries Canada and France.

Code Snippet 20:

```
caLocale = new Locale("fr", "CA");
frLocale = new Locale("fr", "FR");
```

The program will be flexible when the program accepts the locale information from the command line at runtime, instead of using hard-coded language and country codes. Code Snippet 21 demonstrates how to accept the language and country code from command line:

Code Snippet 21:

```
String language = new String(args[0]);
String country = new String(args[1]);
currentLocale = new Locale(language, country);
```

`Locale` objects are only identifiers. After defining a `Locale`, the next step is to pass it to other objects that perform useful tasks, such as formatting dates and numbers. These objects are locale-

sensitive because their behavior varies according to `Locale`. A `ResourceBundle` is an example of a locale-sensitive object.

Following section describes certain important methods of the `Locale` class:

- **`public static Locale getDefault()`**: This method gets the default `Locale` for this instance of the JVM. Here, `Locale` is the return type. In other words, the method returns an object of the class `Locale`.
- **`public final String getDisplayCountry()`**: This method returns the name of the country for the current `Locale`, which is appropriate for display to the user. Here, `String` is the return type. In other words, the method returns a `String` representing the name of the country.
- **`public final String getDisplayLanguage()`**: This method returns the name of the language for the current `Locale`, which is appropriate for display to the user.
For example, if the default locale is `fr_FR`, the method returns French. Here, `String` is the return type. In other words, the method returns a `String` representing the name of the language.

8.9.3 Creating a `ResourceBundle`

`ResourceBundle` objects contain locale-specific objects. These objects are used to isolate locale-sensitive data, such as translatable text.

The `ResourceBundle` class is used to retrieve locale-specific information from the properties file. This information allows a user to write applications that can be:

- Localized or translated into different languages.
- Handled for multiple locales at the same time.
- Supported for more locales later.

The `ResourceBundle` class has a static and final method called `getBundle()` that helps to retrieve a `ResourceBundle` instance.

The `ResourceBundle getBundle(String, Locale)` method helps to retrieve locale-specific information from a given properties file and takes two arguments, a `String` and an object of `Locale` class. The object of `ResourceBundle` class is initialized with a valid language and country matching the available properties file.

To create the `ResourceBundle`, consider following statement:

```
messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
```

The arguments passed to the `getBundle()` method identify the properties file that will be accessed. The first argument, `MessagesBundle`, refers to following family of properties files:

- `MessagesBundle_en_US.properties`
- `MessagesBundle_fr_FR.properties`
- `MessagesBundle_de_DE.properties`
- `MessagesBundle_ja_JP.properties`

The `currentLocale`, which is the second argument of `getBundle()` method, specifies the selected `MessagesBundle` files. When the `Locale` was created, the language code and the country code were passed to its constructor. Note that the language and country codes follow the word `MessagesBundle` in the names of the properties files.

To retrieve the locale-specific data from the properties file, the `ResourceBundle` class object should first be created and then, following methods should be invoked:

- `public final String getString(String key)`: The `getString()` method takes a string as an argument that specifies the key from the properties file whose value is to be retrieved. The method returns a string, which represents the value from the properties file associated with the key. In the method, `key` is a string representing an available key from the properties file. The return value is a `String` representing the value associated with the key.
- `public abstract Enumeration<String>getKeys()`: The `getKeys()` method returns an enumeration object representing all the available keys in the properties file. In the method, `Enumeration` object represents all the available keys in the properties file.

After invoking all the required methods, the translated messages can be retrieved from the `ResourceBundle` class.

8.9.4 Fetching the Text from the ResourceBundle Class

The properties files contain key-value pairs. The values consist of the translated text that the program will display. The keys are specified when fetching the translated messages from the `ResourceBundle` with the `getString()` method. For example, to retrieve the message identified by the `greetings` key, the `getString()` method is invoked.

Following statement illustrates how to retrieve value from the key-value pair using the `getString()` method:

```
String msg1 = messages.getString("greetings");
```

The statement uses the key `greetings` because it reflects the content of the message.

The key is usually hard-coded in the program and it must be present in the properties files. If the translators accidentally modify the keys in the properties files, then `getString()` method will be unable to locate the messages.

8.10 Internationalization Elements

There are various elements that vary with culture, region, and language. Therefore, it should be ensured that all such elements are internationalized.

8.10.1 Component Captions

These refer to the GUI component captions such as text, date, and numerals. These GUI component captions should be localized because their usage vary with language, culture, and region.

Formatting the captions of the GUI components ensures that the look and feel of the application is in a locale-sensitive manner. The code that displays the GUI is locale-independent. There is no the required to write formatting routines for specific locales.

8.10.2 Numbers, Currencies, and Percentages

The format of numbers, currencies, and percentages vary with culture, region, and language. Hence, it is necessary to format them before they are displayed. For example, the number 12345678 should be formatted and displayed as 12,345,678 in the US and 12.345.678 in Germany.

Similarly, the currency symbols and methods of displaying the percentage factor also vary with region and language. Formatting is required to make an internationalized application, independent of local conventions with regards to decimal-point, thousands-separators, and percentage representation. The `NumberFormat` class is used to create locale-specific formats for numbers, currencies, and percentages.

➤ **Number**

The `NumberFormat` class has a static method `getNumberInstance()`. The `getNumberInstance()` method returns an instance of a `NumberFormat` class initialized to default or specified locale. The `format()` method of the `NumberFormat` class should be invoked next, where the number to be formatted is passed as an argument. The argument can be a primitive or a wrapper class object.

The syntaxes for some of the methods that are used for formatting numbers are as follows:

- `public static final NumberFormat getNumberInstance():` Here, `NumberFormat` is the return type. This method returns an object of the class `NumberFormat`.
- `public final String format(double number):` Here, `number` is the number to be formatted. `String` is the return type and represents the formatted text.
- `public static NumberFormat getNumberInstance(Locale inLocale):` Here, `NumberFormat` is the return type. This method returns an object of the class `NumberFormat`. The argument `inLocale` is an object of the class `Locale`.

Code Snippet 22 shows how to create locale-specific format of number for the country Japan.

Code Snippet 22:

```
import java.text.NumberFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {
static public void printValue(Locale currentLocale) {
Integer value = new Integer(123456);
Double amt = new Double(345987.246);
NumberFormat numFormatObj;
String valueDisplay;
```

```

String amtDisplay;
numFormatObj = NumberFormat.getNumberInstance(currentLocale);
valueDisplay = numFormatObj.format(value);
amtDisplay = numFormatObj.format(amt);
System.out.println(valueDisplay + " " + currentLocale.toString());
System.out.println(amtDisplay + " " + currentLocale.toString());
}
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
String language;
String country;
if (args.length != 2) {
language = new String("en");
country = new String("US");
}
else {
language = new String(args[0]);
country = new String(args[1]);
}
Locale currentLocale;
ResourceBundle messages;
currentLocale = new Locale(language, country);
messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
System.out.println(messages.getString("greetings"));
System.out.println(messages.getString("inquiry"));
System.out.println(messages.getString("farewell"));
printValue(currentLocale);
}
}

```

Figure 8.8 displays the output when the arguments passed are ja JP.

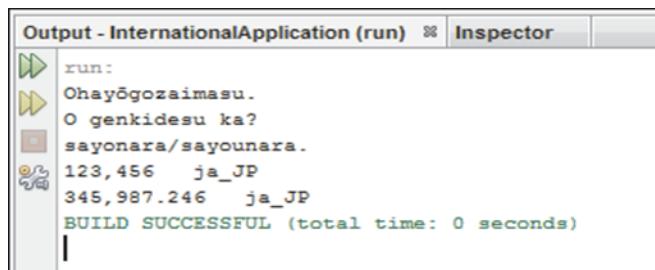


Figure 8.8: Number in Japanese Format

➤ Currencies

The `NumberFormat` class has a static method `getCurrencyInstance()` which takes an instance of `Locale` class as an argument. The `getCurrencyInstance()` method returns an instance of a `NumberFormat` class initialized for the specified locale.

The syntaxes for some of the methods to format currencies are as follows:

```
public final  
String  
format(double  
currency)
```

Here, currency is the currency to be formatted. String is the return type and represents the formatted text.

```
public static  
final  
NumberFormat  
getCurrencyInstan  
ce()
```

Here, NumberFormat is the return type, that is, this method returns an object of the class NumberFormat.

```
public static  
NumberFormat  
getCurrencyInstanc  
e(Locale inLocale)
```

Here, NumberFormat is the return type, that is, this method returns an object of the class NumberFormat. Further, inLocale is the specified Locale.

Code Snippet 23 shows how to create locale-specific format of currency for the country, France.

Code Snippet 23:

```
NumberFormat currencyFormatter;  
String strCurrency;  
// Creates a Locale object with language as French and country as  
//France  
Locale locale = new Locale("fr", "FR");
```

➤ Percentages

This class has a static method `getPercentInstance()`, which takes an instance of the `Locale` class as an argument. The `getPercentInstance()` method returns an instance of the `NumberFormat` class initialized to the specified locale.

The syntaxes for some of the methods to format percentages are as follows:

```
public final  
String  
format(double  
percent)
```

Here, percent is the percentage to be formatted. String is the return type and represents the formatted text.

```
public static  
final  
NumberFormat  
getPercentInstan  
ce()
```

NumberFormat is the return type, that is, this method returns an object of the class NumberFormat.

```
public static  
NumberFormat  
getPercentInstanc  
e(Locale
```

Here, NumberFormat is the return type, that is, this method returns an object of the class NumberFormat. inLocale is the specified Locale.

Code Snippet 24 shows how to create locale-specific format of percentages for the country, France.

Code Snippet 24:

```
NumberFormat percentFormatter;  
String strPercent;  
// Creates a Localeobject with language as French and country  
// as France  
Locale locale = new Locale("fr", "FR");
```

```

// Creates an object of a wrapper class
Double double percent = 25f / 100f;
// Retrieves the percentFormatter instance
percentFormatter = NumberFormat.getPercentInstance(locale);
// Formats the percent figure
strPercent = percentFormatter.format(percent);
System.out.println(strPercent);

```

8.10.3 Date and Times

The date and time format should conform to the conventions of the end user's locale. The date and time format varies with culture, region, and language. Hence, it is necessary to format them before they are displayed. For example, in German, the date can be represented as 20.04.07, whereas in US it is represented as 04/20/07.

Java provides the `java.text.DateFormat` and `java.text.SimpleDateFormat` classes to format date and time. The `DateFormat` class is used to create locale-specific formats for date. Next, the `format()` method of the `NumberFormat` class is also invoked. The date to be formatted is passed as an argument. The `DateFormat.getDateInstance(style, locale)` method returns an instance of the class `DateFormat` for the specified style and locale.

Syntax

```
public static final DateFormat getDateInstance(int style, Locale locale)
```

where,

`style` - is an integer and specifies the style of the date. Valid values are `DateFormat.LONG`, `DateFormat.SHORT`, and `DateFormat.MEDIUM`.
`locale` - is an object of the `Locale` class, and specifies the format of the locale.

`DateFormat` object includes a number of constants such as:

`SHORT`: Is completely numeric such as 12.13.45 or 4 :30 pm
`MEDIUM`: Is longer, such as Dec 25, 1945
`LONG`: Is longer such as December 25, 1945
`FULL`: Represents a complete specification such as Tuesday, April 12, 1945 AD

Code Snippet 25 demonstrates how to retrieve a `DateFormat` object and display the date in Japanese format.

Code Snippet 25:

```

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
public class DateInternationalApplication {
public static void main(String[] args) {
Date today;

```

```

String strDate;
DateFormat dateFormatter;
Locale locale = new Locale("ja", "JP");
dateFormatter = DateFormat.getDateInstance(DateFormat.MEDIUM,
locale);
today = new Date();
strDate = dateFormatter.format(today);
System.out.println(strDate);
}
}

```

Figure 8.9 displays the output.

```

2023/11/03
-----
BUILD SUCCESS
-----
Total time: 0.754 s
Finished at: 2023-11-03T14:55:31+05:30
-----
```

Figure 8.9: Output: Date in Japanese Format

8.10.4 Messages

Displaying messages such as status and error messages are an integral part of any software.

If the messages are predefined, such as "Your License has expired", they can be easily translated into various languages. However, if the messages contain variable data, it is difficult to create grammatically correct translations for all languages.

For example, consider following message in English:

"On 06/03/2007 we detected 10 viruses". In French, it is translated as:

"Sur 06/03/2007 nous avons détecté le virus 10".

In German, it is translated as:

"Auf 06/03/2007 ermittelten wir Virus 10".

The position of verbs and the variable data varies in different languages.

It is not always possible to create a grammatically correct sentence with concatenation of phrases and variables. The approach of concatenation works fine in English, but it does not work for languages in which the verb appears at the end of the sentence. If the word order in a message is hard-coded, it is impossible to create grammatically correct translations for all languages. The solution is to use the `MessageFormat` class to create a compound message.

To use the `MessageFormat` class, perform following steps:

1. Identify the variables in the message. To do so, write down the message, and identify all the variable parts of the message.

For example consider following message:

At 6:41 PM on April 25, 2007, we detected 7 virus on the disk D:

In the message, the four variable parts are underlined.

2. Create a template. A template is a string, which contains the fixed part of the message and the variable parts. The variable parts are encoded in {} with an argument number, for example {0}, {1}, and so on. Each argument number should match with an index of an element in an Object array containing argument values.
3. Create an Object array for variable arguments. For each variable part in the template, a value is required to be replaced. These values are initialized in an Object array. The elements in the Object array can be constructed using the constructors. If an element in the array requires translation, it should be fetched from the Resource Bundle with the getString() method.
4. Create a MessageFormat instance and set the desired locale. The locale is important because the message might contain date and numeric values, which are required to be translated.
5. Apply and format the pattern. To do so, fetch the pattern string from the Resource Bundle with the getString() method. The MessageFormat class has a method applyPattern() to apply the pattern to the MessageFormat instance. Once the pattern is applied to the MessageFormat instance, invoke the format() method.

Code Snippet 26 when executed will display the message in German using MessageFormater class.

Code Snippet 26:

```
import java.text.MessageFormat;
import java.util.Date;
import java.util.Locale;
import java.util.ResourceBundle;
public class MessageFormatterInternationalApplication {
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
String template = "At {2,time,short} on {2,date,long}, we detected
{1,number,integer} virus on the disk {0}";
MessageFormat formatter = new MessageFormat("");
String language, country;
if (args.length != 2) {
language = new String("en");
country = new String("US");
}
else {
language = new String(args[0]);
country = new String(args[1]);
}
Locale currentLocale;
currentLocale = new Locale(language, country);
formatter.setLocale(currentLocale);
ResourceBundle messages = ResourceBundle.getBundle (
"MessageFormatBundle", currentLocale);
```

```

Object[] messageArguments = {messages.getString("disk"), new
Integer(7), new Date()};
formatter.applyPattern(messages.getString("template"));
String output = formatter.format(messageArguments);
System.out.println(output);
}
}

```

Following are the three properties file required by the code.

- **MessageFormatBundle.properties**
 disk = D:
 template = At {2,time,short} on {2,date,long}, we detected
 {1,number,integer} virus on the disk {0}
- **MessageFormatBundle_fr_FR.properties**
 disk = D:
 template = À {2,time,short} {2,date,long}, nous avons détecté
 le virus {1,number,integer} sur le disque {0}
- **MessageFormatBundle_de_DE.properties**
 disk = D:
 template = Um {2,time,short} an {2,date,long}, ermittelten wir
 Virus {1,number,integer} auf der Scheibe {0}

Figure 8.10 displays the output for Code Snippet 26, assuming that the arguments are de DE.



```

Output - JavaApplication19 (run)
run:
Um 09:12 an 4. September 2020, ermittelten wir Virus 7 auf der Scheibe D:
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 8.10: Output: Message Displayed in German

8.11 Additional Classes

In addition, Java comes with two other classes in the `java.util.concurrent` package:

8.11.1 CompletableFuture Class

The `CompletableFuture` class implements the `CompletionStage` and the `Future` interface to simplify coordination of asynchronous operations. An implementation of the already existing `Future` interface represents the result of an asynchronous computation. The `get()` method of `Future` returns the result of a computation once the computation completes, explicitly canceled, or an exception gets thrown. This is a limitation in asynchronous programming that the `CompletableFuture` class is designed to address. The methods of the `CompletableFuture` class can run asynchronously and do not stop program execution.

Table 8.4 explains the important methods available in the `CompletableFuture` class.

Method	Description
supplyAsync()	Accepts a Supplier object that contains code to be executed asynchronously. This method after asynchronously executing the code returns a new CompletableFuture object on which other methods can be applied.
thenApply()	Returns a new CompletableFuture object that is executed with the result of the completed stage, provided the current stage completes normally.
join()	Returns the result when the current asynchronous computation completes or throws an exception of type CompletionException.
thenAccept()	Accepts a Consumer object. On the completion of the current stage, this method wraps the result with the Consumer object and returns a new CompletableFuture object.
whenComplete()	Uses BiConsumer as an argument. Once the calling completion stage completes, whenComplete() method applies completion stage result on BiConsumer. BiConsumer accepts the result as the first argument and error if any as the second argument.
getNow()	Sets the value passed to it as the result if the calling completion stage is not completed.

Table 8.4: Methods of the CompletableFuture Class

8.11.2 CountedCompleter Class

CountedCompleter class extends ForkJoinTask to represent a completion action performed when triggered, provided there are no pending actions. The compute() method of the CountedCompleter class does the main computation and typically invokes the tryComplete() method once before returning. The tryComplete() method checks if the pending count is nonzero and if so, decrements the count. Otherwise, the tryComplete() method invokes the onCompletion(CountedCompleter) method and attempts to complete this task's completer, and if successful, marks this task as complete.

Optionally, the CountedCompleter class may override following methods:

- onCompletion(CountedCompleter): To perform some action upon normal completion.
- onExceptionalCompletion(Throwable, CountedCompleter): To perform some action when an exception is thrown.

A CountedCompleter class is declared as CountedCompleter<Void> when the class does not generate results. Such a class returns null as a result value. For such a class, the getRawResult() method must be overridden to provide a result from join(), invoke(), and related methods.

Code Snippet 27 shows the implementation of the CountedCompleter class.

Code Snippet 27:

```
import java.util.ArrayList;
import java.util.concurrent.ConcurrentLinkedQueue;
```

```
import java.util.concurrent.CountedCompleter;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
public class CountedCompleterDemo {
    static class NumberComputator extends CountedCompleter<Void> {
        final ConcurrentLinkedQueue<String> concurrentLinkedQueue;
        final int start;
        final int end;
        NumberComputator(ConcurrentLinkedQueue<String>
concurrentLinkedQueue, int start, int end) {
            this(concurrentLinkedQueue, start, end, null);
        }
        NumberComputator(ConcurrentLinkedQueue<String>
concurrentLinkedQueue, int start, int end, NumberComputator
parent) {
            super(parent);
            this.concurrentLinkedQueue = concurrentLinkedQueue;
            this.start = start;
            this.end = end;
        }
        @Override
        public void compute() {
            if (end - start < 5) {
                String s = Thread.currentThread().getName();
                for (int i = start; i < end; i++) {
                    concurrentLinkedQueue.add(String.format("Iteration number: %d"
performed by thread %s", i, s));
                }
            }
            propagateCompletion();
        }
        else {
            int mid = (end + start) / 2;
            NumberComputator subTaskA = new NumberComputator
                (concurrentLinkedQueue, start, mid, this);
            NumberComputator subTaskB = new NumberComputator
                (concurrentLinkedQueue, mid, end, this);
            setPendingCount(1);
            subTaskA.fork();
            subTaskB.compute();
        }
    }
}
```

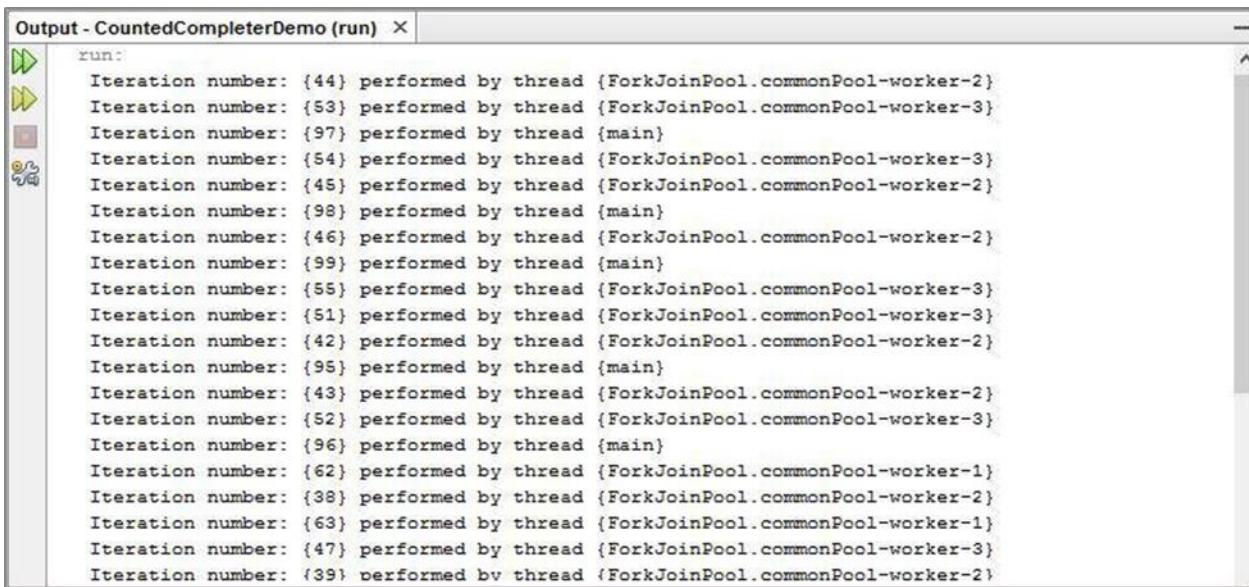
```

public static void main(String[] args) throws ExecutionException,
InterruptedException {
    ConcurrentLinkedQueue<String> linkedQueue = new
    ConcurrentLinkedQueue<>();
    NumberComputator numberComputator = new
    NumberComputator(linkedQueue, 10, 100);
    ForkJoinPool.commonPool().invoke(numberComputator);
    ArrayList<String> list = new ArrayList<>(linkedQueue);
    for (String listItem : list) {
        System.out.println(" " + listItem);
    }
}
}
}

```

The code creates a `CountedCompleterDemo` class with a static inner `NumberComputator` class. The `NumberComputator` class has two overloaded constructors. The first constructor accepts following parameters: a `ConcurrentLinkedQueue` and the start and end indices to perform iterations. The first overloaded constructor, in turn, calls the second one that has an additional parameter, a reference to itself. The `if` block in the overridden `compute()` method along with a `for` loop iterates through the start and end index if their difference is less than five and finally calls `propagateCompletion()` to mark that the current task as complete. Otherwise, two `NumberComputator` sub tasks are created. The `setPendingCount()` method with the argument `1` indicates that only the `subTaskA` sub-task is forked. The `compute()` method called on `subTaskB` makes `subTaskB` execute synchronously. Then, the `main()` method invokes an initialized `NumberComputator` using a `ForkJoinPool` and outputs the result to console.

Figure 8.11 displays one of the possible outputs on executing the `CountedCompleterDemo` class.



The screenshot shows an IDE's output window titled "Output - CountedCompleterDemo (run)". The window contains a list of log entries. Each entry starts with "Iteration number:" followed by a value like {44}, {53}, etc., and ends with "performed by thread {ForkJoinPool.commonPool-worker-2}" or "performed by thread {main}". The list is quite long, showing many iterations across multiple threads.

```

Output - CountedCompleterDemo (run) ×
run:
Iteration number: {44} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {53} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {97} performed by thread {main}
Iteration number: {54} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {45} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {98} performed by thread {main}
Iteration number: {46} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {99} performed by thread {main}
Iteration number: {55} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {51} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {42} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {95} performed by thread {main}
Iteration number: {43} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {52} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {96} performed by thread {main}
Iteration number: {62} performed by thread {ForkJoinPool.commonPool-worker-1}
Iteration number: {38} performed by thread {ForkJoinPool.commonPool-worker-2}
Iteration number: {63} performed by thread {ForkJoinPool.commonPool-worker-1}
Iteration number: {47} performed by thread {ForkJoinPool.commonPool-worker-3}
Iteration number: {39} performed by thread {ForkJoinPool.commonPool-worker-2}

```

Figure 8.11: CountedCompleterDemo – Output

8.12 Recent Enhancements

Java 15 and higher consists of several enhancements in the `java.util.concurrent` package. Utility classes are generally useful in concurrent programming. Specific standardized and expandable frameworks are present in the package. Also included are a few classes that contain helpful functionality, but are difficult to implement.

Following are the major modules with a brief description for each:

Executor Interfaces

The `Executor` interface can be used to define custom thread-like subsystems, including thread pools, Input/Output (I/O) that is asynchronous, and lightweight frameworks of a task. The class that implements the interface decides how and where tasks are executed, that is, in a thread that is newly created, a task-execution thread that is already existing, or the thread that calls `execute()`. These tasks may be executed either in a sequence or concurrently.

Implementation Modules

`ThreadPoolExecutor` and `ScheduledThreadPoolExecutor` classes provide flexible and adjustable thread pools. `ThreadPoolExecutor` executes the given task (`Callable` or `Runnable`) through one of its internally pooled threads.

In addition, the `Executors` class contains various factory methods for the configuration of executors. The utility methods to use them are also provided. The concrete classes `FutureTask` and `ExecutorCompletionService` are some of the other utilities that are based on `Executors`. `ForkJoinPool` class contains an `Executor` that is mainly designed to process instances of `ForkJoinTask` and its subclasses. These classes make use of a work-stealing scheduler strategy to obtain a high throughput for tasks requiring extensive computation. Work stealing in Java is a strategy to reduce conflict and improve processing time and use of resources in multi-threaded applications.

Code Snippet 28 demonstrates an example of using `Executor` class.

Code Snippet 28:

```
import java.util.concurrent.Executor;
import java.util.concurrent.RejectedExecutionException;
public class ExecutorExample {
public static void main(String[] args)  {
ImplementExecutor obj = new ImplementExecutor();
try {
obj.execute(new NewThrd());
}
catch (RejectedExecutionException
| NullPointerException exception) {
System.out.println(exception);
}
}
class ImplementExecutor implements Executor {
@Override
```

```

public void execute(Runnable command) {
    new Thread(command).start();
}
}
class NewThrd implements Runnable {
@Override
public void run() {
System.out.println("This thread executed under executor");
}
}

```

Code Snippet 29 demonstrates the implementation of a function that runs the command at some time. Depending on the Executor implementation, the command may run in a pooled thread, in a thread that is new, or in the calling thread.

Output: This thread executed under executor

Queues

The `ConcurrentLinkedQueue` class provides a flexible, efficient, thread-safe, and non-blocking First In First Out (FIFO) queue, that is also scalable. Another similar class is `ConcurrentLinkedDeque` which also supports the `Deque` interface. The extended `BlockingQueue` interface is responsible for defining the blocking versions of put and take.

Five implementations in `java.util.concurrent` that provide support to this interface are as follows:

- `LinkedBlockingQueue`
- `ArrayBlockingQueue`
- `SynchronousQueue`
- `PriorityBlockingQueue`
- `DelayQueue`

The extended interface `TransferQueue` and implementation `LinkedTransferQueue` introduce a transfer method that is synchronous. The `BlockingDeque` interface also extends `BlockingQueue` in order to provide assistance to both FIFO and Last In First Out (LIFO) stack-based operations. Implementation is provided by the class `LinkedBlockingDeque`.

➤ Synchronizers

Table 8.5 lists five classes that assist synchronization expressions for a particular purpose.

Class	Description
<code>Semaphore</code>	This is the regular tool used for concurrency.
<code>CountDownLatch</code>	This a standard and simple utility to block, until a particular number of conditions, events, or signals is satisfied.
<code>CyclicBarrier</code>	This is a multi-way synchronization point that can be reset. It is used in certain parallel programming styles.
<code>Phaser</code>	This provides a flexible barrier that is used to handle multiple threads with phased computation.

Class	Description
Exchanger	This class is used mostly in pipeline designs, as it permits exchange of objects between two threads at a particular point.

Table 8.5: Synchronizer Classes

Code Snippet 29 demonstrates the use of semaphore class for concurrent collection from memory.

Code Snippet 29:

```

import java.util.concurrent.*;
//A SharedData resource or class.
class SharedData {
static int count = 0;
}
class AppThread extends Thread {
Semaphore sema;
String thrdName;
public AppThread(Semaphore sema, String thrdName) {
super(thrdName);
this.sema = sema;
this.thrdName = thrdName;
}
@Override
public void run() {
// run by thread X
if(this.getName().equals("X")) {
System.out.println("Starting " + thrdName);
try {
// First, get a permit.
System.out.println(thrdName + " is waiting for a permit.");
// acquiring the lock
sema.acquire();
System.out.println(thrdName + " gets a permit.");
// Now, accessing the SharedData resource.
// Other waiting threads will wait, until this
// thread releases the lock
for(int i=0; i < 5; i++) {
SharedData.count++;
System.out.println(thrdName + ": " + SharedData.count);
// Now, allowing a context switch for thread
//Y to execute
Thread.sleep(10);
}
} catch (InterruptedException exc) {
System.out.println(exc);
}
// Release the permit.
System.out.println(thrdName + " releases the permit.");
}
}

```

```

sema.release();
}
// run by thread Y
else {
System.out.println("Starting " + thrdName);
try {
// First, get a permit.
System.out.println(thrdName + " is waiting for a permit.");
// acquiring the lock
sema.acquire();
System.out.println(thrdName + " gets a permit.");
// Now, accessing the SharedData resource.
// other waiting threads will wait, until this
// thread releases the lock
for(int i=0; i < 5; i++) {
SharedData.count--;
System.out.println(thrdName + ":" + SharedData.count);
// Now, allowing a context switch for thread X to
//execute
Thread.sleep(10);
}
} catch (InterruptedException exc) {
System.out.println(exc);
}
// Release the permit.
System.out.println(thrdName + " releases the permit.");
sema.release();
}
}
}

// Class that acts as Driver
public class SemaphoreExample {

public static void main(String args[]) throws InterruptedException {
// Creating a Semaphore object with number of permits as 1
Semaphore sema = new Semaphore(1);
// Creating two threads with name X and Y
// Note that thread X will increment the count
// and thread Y will decrement the count
AppThread mt1 = new AppThread(sema, "X");
AppThread mt2 = new AppThread(sema, "Y");
// starting threads X and Y
mt1.start();
mt2.start();
// waiting for threads X and Y
mt1.join();
mt2.join();
// count will always remain 0 after
// both threads will complete their execution
}

```

```
System.out.println("count: " + SharedData.count);
}
}
```

Code Snippet 29 illustrates how to use a semaphore to lock access to a resource. When a thread wants to use a resource, thread must first call `acquire()`. This calling action has to be done before accessing the resource to acquire the lock. When the thread is done with the resource, it has to release lock by calling `release()`.

Output:

```
Starting X
Starting Y
X is waiting for a permit.
Y is waiting for a permit.
X gets a permit.
X: 1
X: 2
X: 3
X: 4
X: 5
X releases the permit.
Y gets a permit.
Y: 4
Y: 3
Y: 2
Y: 1
Y: 0
Y releases the permit.
```

Concurrent Collections

Other than Queues, various Collection implementations are also given in the Concurrent Collections. These implementations such as `ConcurrentHashMap`, `ConcurrentSkipListMap`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`, can be used in multi-threaded contexts. Whenever multiple threads are required to access a particular collection, use `ConcurrentHashMap` instead of synchronized `HashMap` and `ConcurrentSkipListMap` instead of synchronized `TreeMap`.

Also, whenever the number of reads and traversals are more than the number of updates made to a list, one should use `CopyOnWriteArrayList` instead of synchronized `ArrayList`.

Some classes contained in this package use the `Concurrent` prefix. This prefix is a shorthand that indicates various differences when compared to similar synchronized classes.

Though a concurrent collection is thread-safe, it is not bound by one exclusion lock. Accessing a collection through a single lock results in poor scalability. To avoid this problem, use synchronized classes. Concurrent collections are normally used only when numerous threads have to access a common collection. Collections that are unsynchronized, are useful only when collections are not shared or when they are accessible while holding other locks. The methods that are present in all the classes in `java.util.concurrent` and its sub-packages ensure higher levels of synchronization.

8.13 Summary

- A design pattern is a clearly defined solution to problems that occur frequently. It can be considered as a template or a best practice suggested by expert programmers.
- The singleton design pattern provides complete information on class implementations that can be instantiated only once.
- Data Access Object (DAO) Design Pattern is used when an application is created that must persist its data.
- Factory pattern is one of the commonly used design patterns in Java and belongs to the creational pattern category and provides many ways to create an object.
- The Observer pattern helps to observe the behavior of objects such as change in state or change in property.
- In the internationalization process, the input and output operations of an application are specific to different locations and user preferences. throughput improvements as compared to atomic variables.
- `java.util.concurrent` includes many enhancements in the form of interfaces and classes to support concurrency and parallelism.

8.14 Check Your Progress

1. Identify the component required to create a hash function.
 - a. The instance of operator
 - b. String
 - c. Non zero integer constant
 - d. Design pattern

(A)	a	(C)	c
(B)	b	(D)	d

2. Which of the following statements are true for the logical equality?
 - a. When physical memory locations of the two strings are same, this is called logical equality.
 - b. When data of the objects are same, it is called logical equality.
 - c. The equality operator (==) is used to check for logical equality.
 - d. The equals() method that is used to check for logical equality is inherited from the Object class.

(A)	a, d	(C)	a, c
(B)	b, d	(D)	d

3. Which of the statements are true for Factory design pattern?
 - a. Prevents the application from being tightly coupled to a specific DAO implementation
 - b. Does not depend on concrete DAO classes
 - c. Uses a common interface to refer to the newly created object
 - d. Not used to implement the DAO pattern

(A)	a	(C)	a, c
(B)	b, c	(D)	b, d

4. Which one of the following design patterns provides complete information on class implementations that can be instantiated only once?
 - a. Factory
 - b. Singleton
 - c. Adapter
 - d. Proxy

(A)	a	(C)	c
(B)	b	(D)	d

5. Which of the following option represents the default properties file?

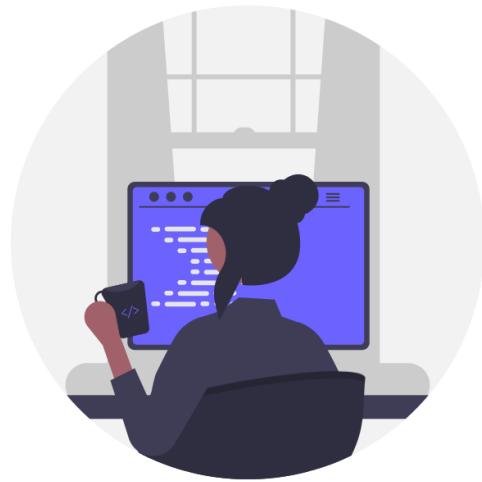
(A)	System.out.println	(C)	MessagesBundle.properties
(B)	MessagesBundle	(D)	System.out

8.14.1 Answers

1. C
2. B
3. C
4. B
5. C

Try It Yourself

1. Implement a Singleton pattern in Java. Create a class that ensures only one instance is created and provide a simple use case for it.
2. Design a simple Data Access Object (DAO) for a hypothetical database table (for example, 'Users'). Implement methods for Create, Read, Update, and Delete (CRUD) operations using the DAO pattern.
3. Implement a Factory Design Pattern for creating different types of shapes (for example, Circle, Square, and Triangle). Demonstrate how the factory method simplifies object creation.
4. Write a Java program that uses the Fork-Join Framework to calculate the sum of elements in a large array. Compare the performance of parallel execution using the Fork-Join Framework with a sequential approach.
5. Implement a parallel sorting algorithm for an array using the Fork-Join Framework. Compare the execution time of the parallel sort with the traditional sequential sorting algorithm for different input sizes.
6. Create a Java application that utilizes the Fork-Join Framework to solve a recursive problem, such as calculating Fibonacci numbers or performing a recursive task on a binary tree. Measure the performance improvements gained from parallelism.



Session 9

Java Data Structures

Welcome to the Session, **Java Data Structures**.

This session explores some of the legacy data structures in Java such as BitSet, Dictionary, and others.

In this Session, you will learn to:

- Explain the Enumeration interface
- Describe the BitSet class
- Describe the Stack classes
- Explain the Dictionary classes

9.1 Enumeration

Enumeration is an interface in the `java.util` package that defines methods to iterate through the elements of a collection. The methods of the Enumeration interface are as follows:

- `hasMoreElements()` : Checks whether or not the enumeration contains more elements.
- `nextElement()` : Returns the next element, if present, in the enumeration.

The exception associated with this interface is `NoSuchElementException`. This exception is raised if `nextElement()` is called when there are no more elements in the enumeration. Code Snippet 1 uses an Enumeration to iterate through the elements of an array.

Code Snippet 1:

```
import java.lang.reflect.Array;
import java.util.Enumeration;
class CustomEnumeration implements Enumeration {
    private final int arraySize;
    private int arrayCursor;
    private final Object array;
```

```

public CustomEnumeration(Object obj) {
    arraySize = Array.getLength(obj);
    array = obj;
}
@Override
public boolean hasMoreElements() {
    return (arrayCursor < arraySize);
}
@Override
public Object nextElement() {
    return Array.get(array, arrayCursor++);
}
}

```

The code creates a `CustomEnumeration` class that implements the `Enumeration` interface. The class constructor accepts an `Object` and stores its size and the object itself to the `arraySize` and `array` variables respectively. The overridden `hasMoreElements()` method returns true if the `arrayCursor` variable is less than the `arraySize` variable. The `nextElement()` overridden method returns an `Object` that represents an element in the array with the index specified by the `arrayCursor` variable.

Code Snippet 2 shows a class that uses the custom enumeration defined in Code Snippet 1.

Code Snippet 2:

```

import java.util.Enumeration;
public class EnumerationDemo {
    public static void main(String[] args) {
        String[] strArray = new String[] {"One", "Two", "Three"}; Enumeration
        customEnumeration = new CustomEnumeration(strArray); while
        (customEnumeration.hasMoreElements()) {
            System.out.println(customEnumeration.nextElement());
        }
    }
}

```

The `main()` method in the code creates a `CustomEnumeration` instance initialized with an array. The `hasMoreElements()` method in the `while` loop checks for more elements in the array. If `hasMoreElements()` method returns true, the `nextElement()` method prints out the array element.

Figure 9.1 displays the output of the `EnumerationDemo` class.

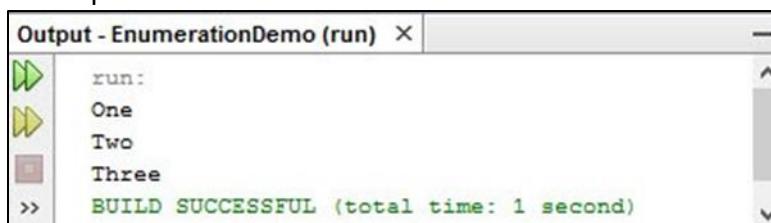


Figure 9.1: EnumerationDemo – Output

Though these codes used a very basic set of data with just three values, in practical scenarios, enumerations can be used for large volume data sets too. However, in recent versions of Java, Enumeration is considered as legacy and retained only for backward compatibility. It is recommended to use the Iterator interface instead of Enumeration.

9.2 BitSet Class

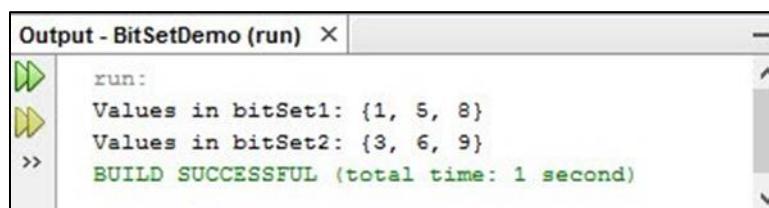
In computing, a bit is a smallest unit of data. Java applications often required to work with large volumes of bits and BitSet class is designed for that purpose. The BitSet class is a collection of bit values and can resize as required. A BitSet indexes its bits with non-negative integer and each bit in a BitSet can be accessed with its index. The advantage of BitSet is that it uses only one bit to store a value.

Code Snippet 3 shows the use of BitSet class.

Code Snippet 3:

```
import java.util.BitSet;
public class BitSetDemo {
    public static void main(String[] args) {
        BitSet bitSet1 = new BitSet();
        BitSet bitSet2 = new BitSet();
        bitSet1.set(1);
        bitSet1.set(5);
        bitSet1.set(8);
        bitSet2.set(3);
        bitSet2.set(6);
        bitSet2.set(9);
        System.out.println("Values in bitSet1: "+bitSet1+"\nValues in
                           bitSet2: "+bitSet2);
    }
}
```

The code creates two BitSet objects and calls the set () method to initialize them. Finally, the BitSet objects are printed out. Figure 9.2 displays the output of the BitSetDemo class.



```
Output - BitSetDemo (run) ×
run:
Values in bitSet1: {1, 5, 8}
Values in bitSet2: {3, 6, 9}
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 9.2: BitSetDemo - Output

In recent versions of Java, this class is considered legacy and not much used. It is retained for backward compatibility.

9.3 Stack Class

A Stack is a collection of objects based on the Last-in First-out (LIFO) principle. In a Stack, the last element added, or pushed to the stack is the first element to be removed, or popped out of the stack. The Stack class extends the Vector class and in addition to the inherited methods of Vector,

`Stack` defines five methods, as listed in Table 9.1.

Method	Description
<code>empty()</code>	Checks whether or not the <code>Stack</code> is empty.
<code>peek()</code>	Returns the object at the top of the <code>Stack</code> without removing the object.
<code>pop()</code>	Returns the object at the top of the <code>Stack</code> after removing the object from the <code>Stack</code> .
<code>push(E item)</code>	Pushes an object onto the top of this <code>Stack</code> .
<code>search(Object o)</code>	Returns the position of an object from the top of the <code>Stack</code> . This method returns 1 for the object at the top of the <code>Stack</code> , 2 for the object below it, and so on. If an object is not found, this method returns -1.

Table 9.1: Methods of the Stack Class

Code Snippet 4 shows the use of the `Stack` class.

Code Snippet 4:

```
import java.util.Stack;
public class StackDemo {
private static Stack getInitializedStack() {
Stack stack = new Stack();
stack.push("obj1");
stack.push("obj2");
stack.push("obj3");
stack.push("obj4");
return stack;
}
public static void main(String[] args) {
Stack initializedStack = StackDemo.getInitializedStack();
System.out.println("Object at top: " + initializedStack.peek());
System.out.println("Position of obj2 from top: " +
initializedStack.search("obj2"));

System.out.println("Object popped out: " + initializedStack.pop());
System.out.println("Object at top: " + initializedStack.peek());
System.out.println("---Elements in Stack---");
for (Object obj : initializedStack) {
System.out.println(obj);
}
}
}
```

The `getInitializedStack()` method of the code creates a `Stack` and pushes four objects to it. The `main()` method calls the `peek()` method to retrieve and print the object at the top of the `Stack`.

The `search()` method obtains and prints the position of the `obj2` object from the top. The `pop()` method pops out the element at the top and the `peek()` method again obtains and prints the element currently at the top. Finally, the enhanced `for` loop prints each element present in the Stack.

Figure 9.3 displays the output of the `StackDemo` class.

```
Output - StackDemo (run) X
run:
Object at top: obj4
Position of obj2 from top: 3
Object popped out: obj4
Object at top: obj3
---Elements in Stack---
obj1
obj2
obj3
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 9.3: StackDemo - Output

9.4 Dictionary Classes

In Java collections, a dictionary is used to store key-value pairs. Every key and value in a dictionary is an object. The `Dictionary` abstract class of the `java.util` package is the superclass of all dictionary implementation classes. Examples of dictionary implementation classes are `Hashtable` and `Properties`.

9.4.1 Hashtable Class

The `Hashtable` class implements a collection of key-value pairs that are organized based on the hash code of the key. A hash code is a signed number that identifies the key. Based on the hash code, a key-value pair, when added to a `Hashtable`, gets stored into a particular bucket. A `Hashtable` is significantly faster as compared to other dictionaries. When a lookup is performed for a key, the `Hashtable` searches for the key in only one particular bucket. As a result, the number of key comparisons significantly reduces.

When elements are added to a `Hashtable`, the `Hashtable` automatically resizes itself by increasing its capacity. When the `Hashtable` capacity reaches the capacity that you specified during its creation, the number of buckets is automatically increased. Internally, the number of buckets is increased to the smallest prime number, which is larger than twice the current number of buckets in the `Hashtable` class. For example, if the current number of buckets is 5 and the `Hashtable` reaches its initial capacity, the number of buckets automatically increases to 11.

Code Snippet 5 shows the use of the `Hashtable` class.

Code Snippet 5:

```
import java.util.Enumeration;
import java.util.Hashtable;
```

```

public class HashtableDemo {
    private static Hashtable initializeHashtable() {
        Hashtable hTable = new Hashtable();
        hTable.put("1", "East");
        hTable.put("2", "West");
        hTable.put("3", "North");
        hTable.put("4", "South");
        return hTable;
    }
    public static void main(String[] args) {
        Hashtable initializedHtable = HashtableDemo.initializeHashtable();
        Enumeration e = initializedHtable.keys();
        System.out.println("---Hashtable Key-Value Pairs---");
        while (e.hasMoreElements()) {
            String key = (String) e.nextElement();
            System.out.println(key + ":" + initializedHtable.get(key));
        }
        e = initializedHtable.keys();
        System.out.println("---Hashtable Keys---");
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
        e = initializedHtable.elements();
        System.out.println("---Hashtable Values---");
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}

```

The `getInitializedHashtable()` method of the code creates and initialize a `Hashtable` with **key-value pairs**. The `main()` method creates an `Enumeration` of the `Hashtable` with a call to the `keys()` method. The `get(key)` method retrieves the value of a particular key. The `elements()` method returns all the values of the `Hashtable` as an `Enumeration` object. The code uses the `Enumeration` object to print the key-value pairs, keys, and values of the `Hashtable`.

Figure 9.4 displays the output of the `HashtableDemo` class.

```
Output - HashtableDemo (run) X
run:
---Hashtable Key-Value Pairs---
4 : South
3 : North
2 : West
1 : East
---Hashtable Keys---
4
3
2
1
---Hashtable Values---
South
North
West
East
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 9.4: HashtableDemo - Output

9.4.2 Properties Class

The `Properties` class extends `Hashtable` to implement a collection of key-value pairs where both the types of the keys and values are `String`. Although being a `Hashtable` subclass, the `Properties` class inherits the `put()` method to add a key-value pair, you should avoid it. This is because, if you add a non-string key or value, the method will fail at runtime. Instead, you should use the `setProperty()` method to add key-value pairs and the `getProperty()` method to retrieve one.

Code Snippet 6 shows the use of the `Properties` class.

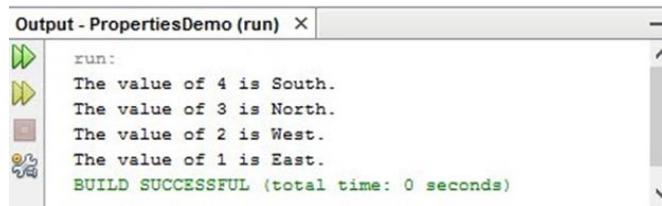
Code Snippet 6:

```
import java.util.Iterator;
import java.util.Properties;
import java.util.Set;
public class PropertiesDemo {
private static Properties initializeProperties() {
Properties properties = new Properties();
properties.setProperty("1", "East");
properties.setProperty("2", "West");
properties.setProperty("3", "North");

properties.setProperty("4", "South");
return properties;
}
public static void main(String[] args) {
Properties initializedProperties =
PropertiesDemo.initializeProperties();
Set set = initializedProperties.keySet();
Iterator itr = set.iterator();
while (itr.hasNext()) {
String str = (String) itr.next();
```

```
System.out.println("The value of " + str + " is " +
initializedProperties.getProperty(str) + ".");
}
}
}
```

The `getInitializedProperties()` method of the code creates a `Properties` object and initializes it with key-value pairs through calls to the `setProperty()` method. The `main()` method calls the `keySet()` method to retrieve a `Set` object containing a collection of keys. The `iterator()` method returns an `Iterator` object that is used to iterate through the key-value pairs. For each iteration, the key-value pairs are printed by `Iterator.next()` and `Properties.getProperty()`. Figure 9.5 displays the output of the `PropertiesDemo` class.



The screenshot shows the Eclipse IDE's Output window titled "Output - PropertiesDemo (run)". The window contains the following text:
run:
The value of 4 is South.
The value of 3 is North.
The value of 2 is West.
The value of 1 is East.
BUILD SUCCESSFUL (total time: 0 seconds)

Figure 9.5: PropertiesDemo – Output

9.5 Summary

- Java includes a few legacy data structures such as Enumeration, BitSet, and so on for backward compatibility.
- Enumeration interface is used to iterate through the elements of a collection.
- BitSet is a collection of bit values.
- Stack extends Vector to provide an implementation of a LIFO collection.
- Dictionary is used to store key-value pairs.
- Hashtable stores key-value pairs where keys are organized based on their hash code.
- Properties stores key-value pairs where both the types of the keys and values are String.

9.6 Check Your Progress

1. Which element of Enumeration returns the next element, if present, in the enumeration?

(A)	nextElement()	(C)	iterate()
(B)	next()	(D)	getNext()

2. Which of the following code snippets correctly adds an element to a BitSet object named bitSetObject?

(A)	bitSetObject.add(1);	(C)	bitSetObject.push(1);
(B)	bitSetObject.put(1);	(D)	bitSetObject.set(1);

3. Which method of the Stack class removes the object from the top of the stack?

(A)	push()	(C)	pop()
(B)	remove()	(D)	peek()

4. Which of the following Hashtable methods returns an Enumeration for the Hashtable?

(A)	elements()	(C)	next()
(B)	enumeration()	(D)	getEnumeration()

5. Which method adds a key-value pair to a Properties object?

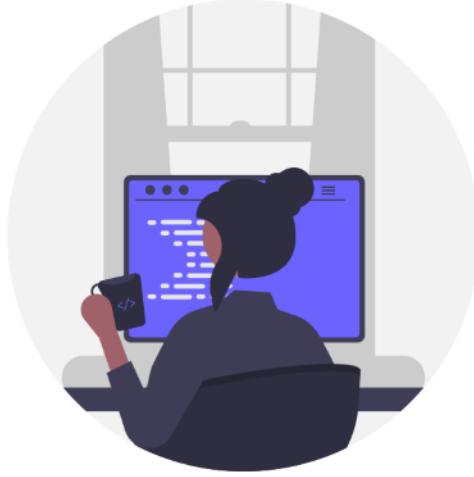
(A)	addProperty()	(C)	push()
(B)	add()	(D)	setProperty()

9.6.1 Answers

1. A
2. D
3. C
4. A
5. D

Try It Yourself

1. Write a Java program that demonstrates following basic operations of the BitSet class:
 - Create two BitSet objects of the same size.
 - Set some bits in the first BitSet.
 - Set different bits in the second BitSet.
 - Perform bitwise AND, OR, and XOR operations between the two BitSet objects.
 - Print the results of each operation.
2. Implement a program that evaluates a postfix expression using a stack. The input is a string containing numbers and operators, such as "3 4 + 5 *".
3. Create a Hashtable to store country names and their corresponding capitals. Add a few entries to the hashtable. Retrieve and display the capital for a specific country.



Session 10

New and Advanced Features of Java

Welcome to the Session, **New and Advanced Features of Java**.

This session explores some of the new and advanced features introduced in recent versions of Java.

In this Session, you will learn to:

- List the enhancements in switch-case
- Explain new FileSystems methods
- Explain the not Predicate method
- Describe Records and Text Blocks
- Explain Flow, Stack Walking, and HTTP Client APIs
- Describe accounting style currency format support
- Explain CompactNumberFormat class
- Describe Sealed Classes for encapsulation and enhancing security

10.1 Enhancements in Switch

Until now, a traditional style of `switch` statement has been commonly used in Java. This `switch` follows the design of C and C++. The most critical point to note in this old `switch` is the fall-through behavior that works well as long as `break` statements are included. Here, when a `case` is matched and executed, the `switch` automatically continues to the next `case`, unless there is a `break` statement. However, consider a situation where a `break` statement is missed. The execution will simply continue to other matching cases, thus making the code prone to errors. Therefore, this must be used more in low-level codes rather than high-level codes.

Java 14 onwards includes enhancements to the `switch` statement. Features of the enhanced `switch` are as follows:

- **Supports multiple values per case**

The traditional old `switch` does not support specifying multiple values per `case`. On the other hand, the enhanced `switch` allows you to specify multiple values per `case`, each of which is delimited by commas.

Code Snippet 1 demonstrates multiple values per case in `switch` statement. Observe the lines in bold.

Code Snippet 1:

```
import java.util.Scanner;
public class EnhancedSwitchDemo {
public static void main(String[] args) {
Scanner sc= new Scanner(System.in);
System.out.print("Enter Product ID to check the Product
label:");
int prodID = sc.nextInt();
switch (prodID) {
case 101, 102, 103 :
System.out.println("You have selected a smartwatch!");
break;
case 104, 105:
System.out.println("You have selected a smartphone!");
break;
}
}
}
```

Output:

```
Enter Product ID to check the Product label:105
You have selected a smartphone!
```

In Code Snippet 1, the code checks the product id entered by the user. If the id matches with any `case` value provided, it executes the respective code block/statements. Since 105 is specified here as the input, the case corresponding to this value is matched successfully and the appropriate statement is printed.

→ **Uses `yield` to return a value**

A new keyword `yield` has been introduced which is similar to the `return` statement, but is exclusively used with `switch`. It is used to specify value to be returned from a `switch` branch. A `yield` terminates the `switch` expression, because a `break` is not required after it.

Code Snippet 2 demonstrates the usage of `yield` keyword.

Code Snippet 2:

```
import java.util.Scanner;
public class EnhancedSwitchDemo2 {
public static void main(String[] args) {
Scanner sc= new Scanner(System.in);
System.out.print("Enter the Product Name to check the Product
```

```

availability: ");
String prodName= sc.next();
int res = getResultViaYield(prodName);
String status = res==1? "Available." : "Not Available.";
System.out.println("The Product is "+ status);
}
private static int getResultViaYield(String name) {
int result = switch (name) {
case "Bolt", "Nut":
//if we enter Bolt or Nut, this yields/returns 1
yield 1;
case "Rivet", "Screw":
//if we enter Rivet or Screw, this yields/returns 2
yield 2;
case "Nail":
//if we enter Nail, this yields/returns 3
yield 3;
default:
yield -1;
};
return result;
}
}

```

Output:

Enter the Product Name to check the Product availability: Rivet
The Product is Not Available.

In Code Snippet 2, `switch` and `yield` are used to check the product name entered by the user. In this code, there are no `return` statements or `break` in the `switch case` blocks. The `switch` block simply checks for matching product name and yields or returns respective results. The program then uses the returned result in a statement with ternary operator to assign "Available." or "Not Available." status. The status is then appended to a string "The Product is" in a `println` statement. Thus, the final outcome will inform the user whether the product whose name has been entered is currently available or not available.

→ Serves as an expression

The old traditional `switch` was allowed only as a statement, whereas, the enhanced `switch` serves as an expression as well. The difference between the two is that you can now use `switch` as an expression that is based on the input and it can also directly return some value. This is similar to the ternary operators that achieve the same logic by using `if-else` block. Code Snippet 3 shows an example to understand this better.

Code Snippet 3:

```

import java.util.Scanner;
public class EnhancedSwitchDemo3 {
public static void main(String[] args) {

```

```

Scanner sc= new Scanner(System.in);
System.out.print("Enter a Code to check State stats:"); int
ivar = sc.nextInt();
// Retrieve the result of a switch expression
// and assign it to a variable.
String numberYieldColon = switch (ivar) {
case 0: yield "Texas ";
case 1: yield "California"; case 2: {
String colResult = "Exclusively ";
colResult = colResult + "Seattle"; yield colResult;
}
case 3: yield "finally, Chicago";
default: yield "NA";
}; //switch ends with semicolon
System.out.println("Leading State is " + numberYieldColon);
}
}

```

Output:

Enter a Code to check State stats: 1
Leading State is California

On the other hand, if 2 is entered as shown here, the output will be different:

Enter a Code to check State stats: 2
Leading State is Exclusively Seattle

In this snippet, `switch` is serving as an expression rather than a statement, since the result of the `switch` is being assigned to a variable. Based on the code entered by user, the `switch` block matches against various `case` one by one. `yield` has been used to return a string for each `case`. There is also a `case` wherein a string is appended and then, returned via `yield`. Observe that the `switch` block ends with a semicolon because it is now an expression. The entire block to the right of the assignment operator assigning value to `numberYieldColon` constitutes an expression.

→ **Is necessary to return a value/exception**

When using `switch` expression, the developer must cover all the possible inputs. This can be done either by providing `case` for all the possible values or by providing default `case`.

It is mandatory for the `switch` expression to return some value or explicitly throw an exception, irrespective of the input value. In addition to this, one should ensure that there is no `NullPointerException` raised as a result of the `switch` expression.

For example, if you comment out the default block in Code Snippet 2, a compiler error shows up, saying that the `switch` failed to cover all possible values. Hence, you can either use a `default` block with some action or throw an exception in the `default` block.

Code Snippet 4 shows a modified version of Code Snippet 2 wherein an explicit exception is thrown when the product name entered is not found.

Code Snippet 4:

```
import java.util.Scanner;
public class EnhancedSwitchDemo4 {
    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        System.out.print("Enter the Product Name to check the Product availability:");
        String prodName= sc.next();
        int res = getResultViaYield(prodName);
        String status = res==1? "Available" : "Not Available";
        System.out.println("The Product is "+ status);
    }
    private static int getResultViaYield(String name) {
        int result = switch (name) {
            case "Bolt", "Nut":
                //if we enter Bolt or Nut, this yields or returns 1
                yield 1;
            case "Rivet", "Screw":
                //if we enter Rivet or Screw, this yields or returns 2
                yield 2;
            case "Nail":
                //if we enter Nail, this yields or returns 3
                yield 3;
            default:
                throw new IllegalArgumentException(name + " is an unknown product and not found in catalog.");
        };
        return result;
    }
}
```

Output:

```
Enter the Product Name to check the Product availability:
Mitten
```

```
Exception in thread "main"
java.lang.IllegalArgumentException: Mitten is an unknown product and not found in catalog.
at EnhancedSwitchDemo4.getResultViaYield(EnhancedSwitchDemo4.java:24)
at EnhancedSwitchDemo4.main(EnhancedSwitchDemo4.java:7)
```

In this code, the switch expression will match the given value against each of the case statements and when no match is found, it goes to the default block and throws an exception. The message in the exception is the custom message given by the developer.

→ **Uses arrows**

Another new addition to the `switch` statement or expression is the arrow syntax. In this syntax, an arrow symbol is used in each `case` and in the `default` block.

`switch` with arrow `->` syntax can be used both as an expression and a statement. If both left and right statement cases match, then statements present on right side of the arrow `->` are executed.

Elements that can be present on the right side of arrow `->` are namely, a `throw` statement, a statement or an expression, and a `{}` block.

In the new arrow syntax, it is not necessary to include `break` and it does not have fall-through behavior. Developers can still use multiple values per case. The main benefit is that it does not require a `break` statement to bypass the default fall-through.

`case`: should be used if a fall-through is required, otherwise `case ->` should be used.

Code Snippet 5 shows the syntax utilized per case for multiple values.

Code Snippet 5:

```
import java.util.Scanner;
public class EnhancedSwitchDemo5 {
public static void main(String[] args) {
Scanner sc= new Scanner(System.in);
System.out.print("Enter Product ID to check the Product
label: ");
int prodId = sc.nextInt();
System.out.println(getResultViaYield(prodId));
}
private static String getResultViaYield(int id) {
String res = switch (id) {
case 001 -> "This id represents a smart television";
case 002 -> "This id represents a smartphone";
case 003,004 -> "This id represents a smart microwave";
default -> "Sorry, No match found";
} ;
return res;
}
}
```

Output:

```
Enter Product ID to check the Product label: 003
This id represents a smart microwave
```

In Code Snippet 5, the statements on right of `->` are executed if an exact case matches the left side. Hence, when the product id is entered as 003, it shows up the result as 'This id represents a smart microwave'.

→ **Has changed Scope**

In a traditional `switch`, the variables declared will exist till the `switch` statement is executed. Suppose if you declare a variable in one of the case branches, it exists in all the

subsequent branches until the end of the switch. Hence, if you want to keep individual case branches as separate scope, it is necessary to provide a {} block. With this, you can avoid variable clashing.

Code Snippet 6 shows an example.

Code Snippet 6:

```
import java.util.Scanner;
public class EnhancedSwitchDemo6 {
public static void main(String[] args) {
Scanner sc= new Scanner(System.in);
System.out.print("Enter Product ID to check the Product
label: ");
int prodID = sc.nextInt();
switch (prodID) {
case 101: {
// The num variable exists just in this {} block
int num = 200;
System.out.println("The value of num is "+num);
break;
}
case 102: {
// This is ok, {} block has a separate scope
int num = 300;
System.out.println("The value of num is "+num);
break;
}
default:
{
System.out.println("Oops! No matches");
}
}
}
}
```

Output

```
Enter Product ID to check the Product label: 101
The value of num is 200
```

This code checks for the id and returns a integer value. Here, a local variable is declared in a case {} block. Hence, the scope of **num** is local to each block demarcated by {}.

10.2 New FileSystems Methods

There are three new methods that have been added to the `FileSystems` class. This makes it easy to handle file system providers, which consider the contents of a file as a file system. Following are the new file method/syntax added:

- `newFileSystem(Path)`
- `newFileSystem(Path, Map<String, ?>)`

→ **newFileSystem(Path, Map<String, ?>, ClassLoader)**

Purpose of each of these methods is to construct a new FileSystem based on given parameters. Code Snippet 7 demonstrates use of one of these newly introduced methods in `FileSystems` class.

Code Snippet 7:

```
//This Java Program illustrates use of new methods of
//FileSystems class

//Importing URI class from java.net package
import java.net.URI;
//Importing required file classes from java.nio package
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;
//Importing Map and HashMap classes from java.util package
import java.util.HashMap;
import java.util.Map;

//Main class
public class NewFileSystemDemo {
public static void main(String[] args) {
try {
Map<String, String> env = new HashMap<>();

// In the following line, we are trying to get path of zip file
Path zipPath = Paths.get("ASample.zip");
// Creating URI from zip path received
URI Uri = new URI("jar:file", zipPath.toUri().getPath(), null);

// Create new file system from uri
FileSystem filesystem = FileSystems.newFileSystem(Uri, env);

// Display message to inform user
System.out.println("Hurray, you have created File System
successfully.");

// Here, we check if file system is open or not, using isOpen()
// method
if (filesystem.isOpen())
System.out.println("It seems File system is open");
else
System.out.println("It seems File system is closed");
}

catch (Exception e) {
// Print the exception with line number
e.printStackTrace();
}
```

```
}
```

```
}
```

Output:

Hurray, you have created File System successfully.

It seems File system is open

`java.nio.file.FileSystems` class is intended as a factory for creating new file systems. The `filesystem` object that has been created in Code Snippet 7 helps to access files and other objects in the file system. In this code, the developer provides a Zip file path and uses `FileSystems` methods to check whether it is open or closed. This file system acts as a factory for creating different objects such as `Path`, `PathMatcher`, `UserPrincipalLookupService`, and `WatchService`.

Code Snippet 8 shows another example of using new methods in `FileSystems` class. This code makes use of a `HashMap`.

Code Snippet 8:

```
import java.util.*;
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.*;
public class NewFileSystemDemo2 {
public static void main(String [] args) throws Throwable {
Map<String, String> env = new HashMap<>();
env.put("create", "true");

// locate file system by using the syntax
// defined in java.net.JarURLConnection
URI uri = URI.create("jar:file:/c:/first/ASample.zip");

try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {

    Path externalTxtFile = Paths.get("/first/Java1.txt");
    Path pathInZipfile = zipfs.getPath("/Java1.txt");
    // Here, we copy a file into the zip file
    Files.copy( externalTxtFile, pathInZipfile,
    StandardCopyOption.REPLACE_EXISTING );
    System.out.println("Copy Successful");
}
catch(Exception e){ System.out.print(e);
}
}
}
```

Output:

Copy Successful

In Code Snippet 8, a `Map` instance is created. Then, a zip file system is created by specifying

the path of the zip file. Configuration options for the zip file system are specified in the `java.util.Map` object passed to the `FileSystems.newFileSystem()` method. Following this, a copy operation is performed. The path of the source file in the default filesystem is specified. The destination file system has also been specified as the zip folder. The given text file will be copied to the new filesystem, that is, the zip file. When you open the zip file, you will see that the text file has been added to it.

10.3 Predicate Not Method

The `Predicate.not()` static method is used to negate an existing predicate. The `Predicate` interface is available in the `java.util.function` package.

Syntax

```
negate = Predicate.not( positivePredicate);
```

where,

`negate`: a predicate that negates the results of the supplied predicate

`positivePredicate`: predicate to negate

The procedure involves creating a predicate and then, initializing the conditions to it. Alternatively, you can also create another predicate to create a negate and then, assign it with `not()` method. Code Snippet 9 shows the use of `Predicate.not()` method.

Code Snippet 9:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;
public class PredicateNotDemo {
    public static void main(String[] args) {
        List<Integer> sampleList= Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        /* Let us create a predicate for negation */ Predicate<Integer>
        findEven = i -> i % 2 == 0;
        /* Now, we create a predicate object which is negation of supplied
        predicate*/
        Predicate<Integer> findOdd = Predicate.not(findEven);
        /* start filtering the even number using even predicate */
        List<Integer> evenNumbers=
        sampleList.stream().filter(findEven).collect( Collectors.toList());
        /* start filtering the odd number using odd predicate */
        List<Integer> oddNumbers =
        sampleList.stream().filter(findOdd).collect( Collectors.toList());
        /* Try to print the Lists for odd or even numbers */
        System.out.println("Here is the list of even numbers
        "+evenNumbers);
        System.out.println("Here is the list of odd numbers "+oddNumbers);
```

```
}
```

Output:

```
Here is the list of even numbers [2, 4, 6, 8, 10]
Here is the list of odd numbers [1, 3, 5, 7, 9]
```

Code Snippet 9 gives the list of odd numbers and even numbers separately.

Predicate.negate() method

The `Predicate.negate()` method first creates the logical negation of the existing predicate and then, returns it. Code Snippet 10 shows an example of the `Predicate.negate()` method.

Code Snippet 10:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;
public class PredicateNegateDemo {
public static void main(String[] args)
{
List<Integer> sampleList = Arrays.asList(2020, 2021, 2022, 2023,
2024, 2025, 2026, 2027, 2028, 2029);
//This is to check whether the predicate is leap or not
Predicate<Integer> isLeap = i -> i % 4 == 0;
Predicate<Integer> isNotLeap = isLeap.negate();
List<Integer> leapList = sampleList.stream()
.filter(isLeap)
.collect(Collectors.toList());
List<Integer> notLeapList = sampleList.stream()
.filter(isNotLeap)
.collect(Collectors.toList());
//print both the lists
System.out.println("Leap Years are "+ leapList);
System.out.println("Not Leap Years are "+ notLeapList );
}
}
```

Output:

```
Leap Years are [2020,2024,2028]
Not Leap Years are [2021, 2022, 2023, 2025, 2026, 2027, 2029]
```

In this code, first you create a list with all the integers. Then, add the predicates to `leapList` for checking whether its leap or not. Use `negate()` to find the 'not leap' predicates.

10.4 Records

In Java, records are a new restricted form of class to declare types such as an enum.

First introduced in Java 14, records have been enhanced in Java 16. Additional changes have been brought about in this version. Defining a record is an easy way of defining an immutable data-

holding object. It commits to an API that matches the representation after declaring it.

Records cannot separate API from the presentation. Due to this reason, records become more concise. The record keyword is used to create a record class. A record class declaration comprises a name, optional type parameters, a header, and a body. The header is a list of components of the record class, which are variables that constitute its state (also called as state description).

Since records are considered simple and are transparent holders for data, they automatically acquire multiple standard members. Following are some of the standard members of a record:

- The state description's individual component contains a private final field.
- The state description's individual component contains a public read accessor method. The type and name is the same as the component.
- A public constructor is used to initialize each field using the corresponding arguments. This constructor's signature matches that of the state description.
- Implementations of `hashCode()` and `equals()` methods. Consider two records containing the same state and type. These two records will be treated as equal on account of these method implementations.
- `toString()` method will be implemented. This is inclusive of all the strings of all the record components, along with their names.

Note the following while creating record classes:

- A record class cannot be extended and is final. It cannot be abstract.
- Record classes extend `java.lang.Record` class implicitly.
- The fields defined in the record declaration are final.
- Constructor, `equals()`, `hashCode()`, and `toString()` methods are created automatically for a record class. Accessor methods are also automatically from the record class. The method name and field name are the same, unlike conventional and generic getter methods. The constructor generated by default can be quite cumbersome to use practically, so as an alternative, you can declare a compact constructor whose signature is implicit. This is easier for coding and for readability as well.
- Developers can edit the record fields depending on the type.
- Using all the fields given in the record definition, a single constructor can be created. Code Snippet 11 shows a simple example of creating and using a record class named Employee.

Code Snippet 11:

```
//Defining a record class
record Employee(int id, String ename, float salary, String address)
{
// Instance fields must be present in the record's
// parameters but record can define static fields.
static int count;
// Constructor 1 of this class
// Compact Constructor
public Employee {
if (ename.length() < 2) {
throw new IllegalArgumentException(
"First name must be 2 characters or more.");
}
}
// Static methods
public static int generateCount() {
return ++count;
}
@Override
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + ((address == null) ? 0 : address.
hashCode());
result = prime * result + ((ename == null) ? 0 : ename.
hashCode());
result = prime * result + id;
result = prime * result + Float.floatToIntBits(salary);
return result;
}
@Override
public boolean equals(Object obj) {
if (this == obj)
return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
Employee other = (Employee) obj;
if (address == null) {
if (other.address != null)
return false;
} else if (!address.equals(other.address))
return false;
if (ename == null) {
if (other.ename != null)
return false;
} else if (!ename.equals(other.ename))
return false;
```

```

if (id != other.id)
return false;
if (Float.floatToIntBits(salary) != Float.floatToIntBits(other.
salary))
return false;
return true;
}
}

// Driver class to demonstrate use of records in Java
public class EmployeeDemo {
public static void main(String[] args) {
Employee emplrecord = new Employee(15, "Nancy", 1000, null);
Employee emp2record = new Employee(15, "Nancy", 1000, null);
System.out.println(emplrecord);
// accessing fields
System.out.println("Name: "+emplrecord.ename());
System.out.println("ID: "+emplrecord.id());
// Using equals()
System.out.println(emplrecord.equals(emp2record));
// Using hashCode()
System.out.println(emplrecord == emp2record);
}
}

```

Output:

Name: Nancy
ID: 15
true
false

This program creates two employee records and checks whether they are same or not. The record definition overrides `hashCode()` and `equals()` methods to provide custom functionality.

10.5 Text Blocks

Though a preview feature in earlier versions, text blocks are a standard feature in Java 16. Text blocks provide another way to write String literals in source code. They allow you to include literal fragments of HTML, JSON, SQL, or whatever you want, in a more precise and understandable way.

The main benefit of using text blocks is that new lines along with quotes can be freely used, without the necessity for escaping line breaks.

For example, a simple text block is as follows:

```

String
example =
""" Sample
text""";

```

The text block feature has been introduced in Java to:

- Express the lengthy strings easily, avoiding escape sequences. Thus, text blocks make the task of writing Java programs easier.
- Help understand strings that denote non-Java code in Java programs, thereby improving readability.
- Enable manipulation of a construct similar to a string literal. You can migrate from string literals as any new construct can express the strings similar to a string literal and interpret the identical escape sequences.

Earlier, if a snippet of HTML, XML, SQL, or JSON in a string literal "..." was to be embedded in Java, extensive edits were required, such as adding delimiters and escapes, joining two strings (called concatenation), and so on. For developers, this made code snippets complex and hard to maintain. Text blocks now help solve this problem by negating the requirement for all this additional work. In this way, more control is with developer with respect to formatting.

Code Snippet 12 shows an example of multiline string declaration.

Code Snippet 12:

```
public class jsonDemo {  
    public static void main(String args[])  
    {  
        String json = "{\"name\": \"Dune\", \"year\": 2021, "  
        + "\"details\": {\"actors\": 25, \"budget \": 35,  
        \"units\": 6},\"tags\": [\"films\", \"epic\"],"  
        + "\"rating\": 9};  
        System.out.println(json);  
    }  
}
```

Output:

```
{"name": "Dune", "year": 2021, "details": {"actors": 25, "budget  
(millions)": 35, "units": 6}, "tags": ["films", "epic"], "rating": 9}
```

Here, the String literal represents a JSON document embedded in Java code. However, it is accompanied by String concatenation operators (+) and escapes sequences for double quotes (""). This makes the code hard to read and maintain, despite IDE assistance. The text block feature allows programmers to show multi-line Strings while avoiding escape sequences in general cases. Code Snippet 13 shows how you can rewrite Code Snippet 12 JSON string literal, using a text block.

Code Snippet 13:

```
public class jsonDemo {  
    public static void main(String args[])  
    {  
        String json = """ {
```

```

"name": "Dune", "year": 2021,
"details": {"actors": 25, "budget (millions)": 35,
"units": 6}, "tags": ["films", "epic"], "rating": 9
} """;
System.out.println(json);
}
}

```

The output will be the same as that displayed for Code Snippet 12.

10.6 Flow API

Flow API, is the official support for Reactive Streams Specification. It is a combination of the Iterator and Observer patterns, that is, the Pull and Push patterns. Unlike RxJava which is an end-user API, the Flow API is an interoperation specification.

Flow API consists of four basic interfaces:

- **Subscriber**: The `Subscriber` subscribes to `Publisher` so that callbacks generated can be tracked.
- **Publisher**: The `Publisher` publishes the stream of data items to subscribers who are registered.
- **Subscription**: This is the link between the publisher and the subscriber.
- **Processor**: The `Processor` lies between `Publisher` and `Subscriber` and performs the task of transforming one stream to another.

Code Snippets 14a and 14b show a primary subscriber that accepts one data object, prints it, and expects one more. A publisher implementation provided by Java (`SubmissionPublisher`) can be used to complete the streaming session. A sample for subscription class is also shown. On implementing the `Flow.Subscriber` class, the methods `onNext()` and `onComplete()` are overridden.

Code Snippet 14(a):

```

import java.util.concurrent.Flow;
import java.util.List;
import java.util.concurrent.SubmissionPublisher;
public class SubscriberDemo<T> implements Flow.Subscriber<T>
{ private Flow.Subscription subs;
@Override
public void onSubscribe(Flow.Subscription subs) {
this.subs = subs;
this.subs.request(1);
}
@Override
public void onNext(T item) {
System.out.println(item);
subs.request(1);
}

```

```
@Override  
public void onError(Throwable throwable)  
{ throwable.printStackTrace();  
}  
@Override  
public void onComplete() {  
System.out.println("Control has reached OnComplete method");  
}  
}
```

Code Snippet 14(b):

```
//Driver class to demonstrate the working of Flow API  
import java.util.concurrent.Flow;  
import java.util.List;  
import java.util.concurrent.SubmissionPublisher;  
public class FlowAPIDemo {  
public static void main(String args[]) {  
List<String> items = List.of("1", "2", "3", "4", "5", "6", "7",  
"8", "9", "10");  
SubmissionPublisher<String> samplePublisher = new  
SubmissionPublisher<>();  
samplePublisher.subscribe(new SubscriberDemo<>()); items.forEach(s  
-> {  
try {  
Thread.sleep(1000);  
} catch (InterruptedException e) {  
e.printStackTrace();  
}  
samplePublisher.submit(s);  
});  
samplePublisher.close();  
}  
}
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
Control has reached OnComplete
```

Here, a sample publisher was created. Then, a list was created. The `submit()` method of `SubmissionPublisher` class was used to publish each item in the list to each subscriber.

10.7 HTTP Client API

Earlier versions of Java have only provided a low-level `HttpURLConnection` API. This API does not have high performing features and is not user-friendly. To overcome the drawbacks, third-party libraries such as Apache `HttpClient`, Jetty, and Spring's `RestTemplate` were commonly used.

Then, Java introduced a new HTTP Client API which is available since version 11. This API is used to send requests and get back their responses. The immutable HTTP Client module was first created as an incubator module in JDK 9 and became stable since Java 11. It supports `HTTP/2` with backward compatibility. To use it, developers should define the module using a `module-info.java` file that indicates required modules to run the application. Unlike `HttpURLConnection`, HTTP Client offers both synchronous and asynchronous request mechanisms.

The new HTTP API types can be found in `java.net.http` package. They include classes, interfaces, enums, and more.

Three core classes in the API are as follows:

- **HttpRequest** – This class is for the request to be sent through the `HttpClient`.
- **HttpClient** – This class is used as a container for configuring information that is common to multiple requests.
- **HttpResponse** – This class stands for the result of an `HttpRequest` call.

For handling a request, you should first create an `HttpClient` through a builder (`HttpClient.Builder`).

For example, one can create a client, request, and response as follows:

```
HttpClient testclient =  
    HttpClient.newHttpClient();  
HttpRequest request = HttpRequest.newBuilder()  
    .uri(URI.create(uri))  
    .build();  
HttpResponse<String>  
response =  
    testclient.send(request, BodyHandlers.ofString());
```

An `HttpClient` gives resource sharing and configuration information for the requests that are sent through it. Then, developers can use the builder to configure as per the client state, such as the preferred protocol version (`HTTP/1.1` or `HTTP/2`), whether to follow redirects, a proxy, or an authenticator.

Each `HttpRequest` sent is supplied with a `BodyHandler`. The response body if any, is handled by the `BodyHandler`. Once you receive an `HttpResponse`, the response's headers, response code, and body (typically) are accessible. Based on the type `T` of the response body, the response body bytes are read or not read. The HTTP response arrives with several parameters, whether the body part of the HTML is read-only or modifiable because of any reason such as form

filling or JavaScript will be known by the parameter `T` and `BodyHandler` takes care of the HTML Script.

You can send the requests either asynchronously or synchronously.

`send(HttpRequest, BodyHandler)` blocks until the entire process of sending the request and receiving the response is completed.

When the request is sent by `sendAsync(HttpRequest, BodyHandler)`, the response is received asynchronously. The `sendAsync()` method returns immediately with a `CompletableFuture<HttpResponse>`. The `CompletableFuture` ends once the response is available to declare dependencies among several asynchronous tasks. The returned `CompletableFuture` can be combined in different ways.

Code Snippet 15 shows a small snippet with a synchronous example.

Code Snippet 15:

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.Authenticator;
import java.net.InetSocketAddress;
import java.net.ProxySelector;
import java.net.URI;
import java.net.http.*;
import java.nio.file.Paths;
import java.time.Duration;
...
HttpClient myclient = HttpClient.newBuilder()
.version(Version.HTTP_1_1)
.followRedirects(Redirect.NORMAL)
.connectTimeout(Duration.ofSeconds(10))
.proxy(ProxySelector.of(new InetSocketAddress("proxy.lect.com",
80)))
.authenticator(Authenticator.getDefault())
.build();
//HTTP response upon client request
HttpResponse<String> response = myclient.send(request,
BodyHandlers.ofString());
System.out.println(response.statusCode());
System.out.println(response.body());
```

In this case, requests are synchronous and blocking which means that processing is blocked until the response comes. This can lead to issues when processing large amounts of data.

Code Snippet 16 shows an Asynchronous example. In this case, processing will be non-blocking, that is, it does not wait for the response to come in. Tasks will be asynchronously performed.

Code Snippet 16:

```
HttpRequest myrequest = HttpRequest.newBuilder()
```

```

.uri(URI.create("https://lect.com/"))
.timeout(Duration.ofMinutes(2))
.header("Content-Type", "application/json")
.POST(BodyPublishers.ofFile(Paths.get("file.json")))
.build();
//Finally the Client request is processed
client.sendAsync(myrequest, BodyHandlers.ofString())
.thenApply(HttpResponse::body)
.thenAccept(System.out::println);

```

Security checks

While using `send` methods of `HttpClient`, security checks are performed provided a security manager is available. You will require a proper `URLPermission` to access the destination server and proxy server if one is configured.

10.8 Accounting Currency Format Support

Most of the applications today targeted for a larger audience, for example, Internet users, usually deal with money. Therefore, in such applications, there is a requirement to display money or the required currency in a format specific to that location or country.

In certain countries and locales, a special style for currency is used in which the amount is represented in parentheses. This is called as accounting style. In this style, in a locale such as `en_US`, a value `-$9.44` will appear as `($9.44)`. In Java 14 onwards, such currency format instances with accounting style can be obtained by calling the method `NumberFormat.getCurrencyInstance(Locale)` with the `u-cf-account` Unicode locale extension.

Code Snippet 17 demonstrates use of this extension.

Code Snippet 17:

```

import java.text.*;
import java.util.*;
public class CurrencyDemo {
public static void main(String[] args) {
DecimalFormat df1 = (DecimalFormat)
DecimalFormat.getCurrencyInstance(Locale.US);
System.out.println(df1.format(-9.44)); //-$9.44
Locale myLocale = new Locale.Builder().setLocale(Locale.US)
.setExtension(Locale.UNICODE_LOCALE_EXTENSION, "cf-
account").build();
DecimalFormat df2 = (DecimalFormat)
NumberFormat.getCurrencyInstance(myLocale);
System.out.println(df2.format(-9.44)); //($9.44)
}
}

```

Output:

-\$9.44 (\$9.44)

10.9 CompactNumberFormat Class

CompactNumberFormat is a concrete subclass of NumberFormat that formats a decimal number based on patterns in a compact form.

One can create a new CompactNumberFormat instance for a locale using one of the factory methods provided by NumberFormat. An example of CompactNumberFormat is shown in Code Snippet 18.

Code Snippet 18:

```
import java.text.*;
import java.util.*;
public class CurrencyDemo {
public static void main(String[] args) {
    long nf1 =15000000;
    NumberFormat numFormatObj2 =
        NumberFormat.getNumberInstance(Locale.GERMANY);
    NumberFormat numFormatObj3 =
        NumberFormat.getNumberInstance(Locale.ITALY);
    System.out.println("Currencies without compact numbering:");
    System.out.printf("%s %s %s %s", numFormatObj2.format(nf1),
        "Germany", numFormatObj3.format(nf1), "Italy");
    final NumberFormat numberFormatGrShort = NumberFormat.
        getCompactNumberInstance(Locale.GERMANY,
        NumberFormat.Style.SHORT);
    final NumberFormat numberFormatGrLong = NumberFormat.
        getCompactNumberInstance(Locale.GERMANY,
        NumberFormat.Style. LONG);
    final NumberFormat numberFormatItShort = NumberFormat.
        getCompactNumberInstance(Locale.ITALY,
        NumberFormat.Style.SHORT);
    final NumberFormat numberFormatItLong = NumberFormat.
        getCompactNumberInstance(Locale.ITALY,
        NumberFormat.Style.LONG);
    System.out.println("\nDemonstrating Compact Number Formatting
on " + nf1);
    System.out.println("\tDE/Short: " +
        numberFormatGrShort.format(nf1));
    System.out.println("\tDE/Long: " +
        numberFormatGrLong.format(nf1));
    System.out.println("\tIT/Short: " +
        numberFormatItShort.format(nf1));
    System.out.println("\tIT/Long: " +
        numberFormatItLong.format(nf1));
}
}
```

In this code, the `long` value `15000000` is initially formatted and displayed as a currency in two countries, Germany and Italy respectively, without compact numbering. Later, compact numbering is applied and they are displayed again.

Output:

Currencies without compact numbering:

15.000.000 Germany 15.000.000 Italy

Demonstrating Compact Number Formatting on 15000000

DE/Short:	15 Mio.
DE/Long:	15 Millionen
IT/Short:	15 Mln
IT/Long:	15 milioni

Code Snippet 19 shows another example of simple compact number formatting.

Code Snippet 19:

```
import java.text.NumberFormat;
import java.util.Locale;
public class CompactNumberFormatDemo {
public static void main(String[] args)
{
NumberFormat sampleNoFormat = NumberFormat
.getCompactNumberInstance(Locale.US, NumberFormat.
Style.LONG);
System.out.println(sampleNoFormat.format(200) );
System.out.println(sampleNoFormat.format(2000) );
System.out.println(sampleNoFormat.format(20000) );
System.out.println(sampleNoFormat.format(200000) );
NumberFormat sampleShortFormat = NumberFormat
.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
System.out.println(sampleShortFormat.format(200) );
System.out.println(sampleShortFormat.format(2000) );
System.out.println(sampleShortFormat.format(20000) );
System.out.println(sampleShortFormat.format(200000) );
}
}
```

Output:

200
2 thousand
20 thousand
200 thousand
200
2K
20K
200K

It is always better to apply locale-sensitive compact/short number formatting to general purpose

numbers for instance decimal, currency, and percentage. Numbers that represent thousands, for example 2000, can be formatted as '2K' in short style or '2 thousand' in long style. A compact number formatting refers to the observation of a number in a shorter form, based on the patterns provided for a given locale.

Custom CompactNumberFormat instance

A custom instance of `CompactNumberFormat` can be used to represent numbers in shorter form using the constructor `CompactNumberFormat(String, DecimalFormatSymbols, String[])`.

Code Snippet 20 shows a customized `CompactNumberFormat` instance.

Code Snippet 20:

```
import java.text.CompactNumberFormat;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import java.util.Currency;
import java.util.Locale;
public class CustomCompactDemo {
@SuppressWarnings("deprecation")
public static void main(String[] args)
{
final String[] cmpctPttrns = {"", "", "", "0k", "00k", "000k",
"0m", "00m", "000m", "0b", "00b", "000b", "0t", "00t", "000t"};
final DecimalFormat decimalFormat = (DecimalFormat) NumberFormat.
getNumberInstance(Locale.US);
final CompactNumberFormat customCompactNoFormat = new
CompactNumberFormat( decimalFormat.toPattern(),
decimalFormat.getDecimalFormatSymbols(), cmpctPttrns);
System.out.println(decimalFormat.toPattern());
}
```

Output:

```
#,##0.##
```

`compactPttrns` is used to represent a series of patterns. Here, each pattern is utilized in formatting a range of numbers. For example, one to fifteen patterns can be provided in an array, but the first pattern provided, always resembles 100. Based on the number of array elements, these values for the range 100 to 1014.

Set fractional number

It explains the minimum fraction part digits that are acceptable in a number. By default, the fractional part is set to '0' digits. Code Snippet 21 shows an example of formatting with fractional digits.

Code Snippet 21:

```
import java.text.NumberFormat;
import java.util.Locale;
```

```
public class FractionFormatDemo {  
    public static void main(String[] args) {  
        NumberFormat format =  
            NumberFormat.getCompactNumberInstance(Locale.US,  
                NumberFormat.Style.SHORT);  
        format.setMinimumFractionDigits(3);  
        System.out.println(format.format(20000));  
        System.out.println(format.format(20012));  
        System.out.println(format.format(200201));  
        System.out.println(format.format(2222222));  
    }  
}
```

Output:

20.000K
20.012K
200.201K
2.222M

In this code, a CompactNumberFormat instance based on current locale is created using NumberFormat. Then, using setMinimumFractionDigits() method, the number of fraction digits are specified. The given numeric values are formatted and the output is displayed.

Compact number parsing

Compact number parsing is a process used to parse compact number into a long pattern. Code Snippet 22 shows the parsing of a compact number.

Code Snippet 22:

```
import java.text.NumberFormat;  
import java.util.Locale;  
public class ParsingFormatDemo {  
    public static void main(String[] args) throws Exception  
    {  
        NumberFormat format = NumberFormat  
            .getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);  
        System.out.println(format.parse("200") );  
        System.out.println(format.parse("2 thousand") );  
        System.out.println(format.parse("20 thousand") );  
        System.out.println(format.parse("200 thousand") );  
    }  
}
```

Output:

200
2000

```
20000  
200000
```

This code parses given values using `parse()` method of `CompactNumberFormat` based on the `Locale` specified.

10.10 Concept of Sealed Classes for Encapsulation and Enhancing Security

Sealed classes feature from Java 17 onwards enhance encapsulation and security in OOP applications with Java. The feature allows you to control which other classes or interfaces can extend or implement a particular class. By sealing a class, you specify a limited set of classes that can extend it, thereby restricting the hierarchy and preventing unauthorized subclasses.

Key benefits and use cases of sealed classes include:

1. Enhanced Security: Sealed classes help prevent unauthorized or unintended subclasses. This can be especially useful in security-sensitive applications where strict control over class inheritance is required.
2. Encapsulation: By sealing a class, you can enforce a clear and controlled interface for class extension, ensuring that the class's internal state and behavior are maintained as expected.
3. Improved Maintenance: Sealed classes make it easier to maintain and evolve a codebase because you have a clear understanding of which classes can extend them. Changes to the sealed hierarchy are more predictable.

Code Snippet 23 shows how sealed classes can be used in Java.

Code Snippet 23:

```
sealed class Animal permits Dog, Cat {  
    // Common properties and methods for all animals  
}  
  
final class Dog extends Animal {  
    // Dog-specific properties and methods  
}  
  
final class Cat extends Animal {  
    // Cat-specific properties and methods  
}
```

In this example, the `Animal` class is sealed and explicitly permits only `Dog` and `Cat` to extend it. This ensures that no other classes can create subclasses of `Animal` without modifying the permissions explicitly.

10.11 Summary

- The switch statement has been enhanced in many ways in recent Java versions.
- With the multiple value case option, it is possible to provide many values at the same time in a single case.
- A new keyword yield can be used with switch-case blocks to return values.
- switch can now be used as an expression to assign a value to another object.
- FileSystems class has three new methods added to it, namely, newFileSystem(Path), newFileSystem(Path, Map<String, ?>), and newFileSystem(Path, Map<String, ?>, ClassLoader)
- HttpClient provides configuration information and resource sharing for all requests sent through it.
- Records are a new type in Java.
- Java 14 onwards includes support for currency number accounting styles.
- CompactNumberFormat class enables developers to format numbers into compact values.
- Java 17 onwards supports sealed classes.

10.12 Check Your Progress

1. How are records different from objects?

(A)	Records can have multiple values	(C)	Records do not contain methods.
(B)	Objects can have multiple values	(D)	Objects do not have instances.

2. Which class is CompactNumberFormat a subclass of?

(A)	NumberFormat	(C)	Integer
(B)	Math	(D)	Decimal

3. Which class do you use for formatting and parsing numbers for any locale?

(A)	FormatNumber	(C)	GeoLocal
(B)	NumberFormat	(D)	GPS

4. Which of these methods helps to represent all of the record components with their names as strings?

(A)	String	(C)	LineString
(B)	Strcat	(D)	toString

5. Which basic interface does Flow API consist of?

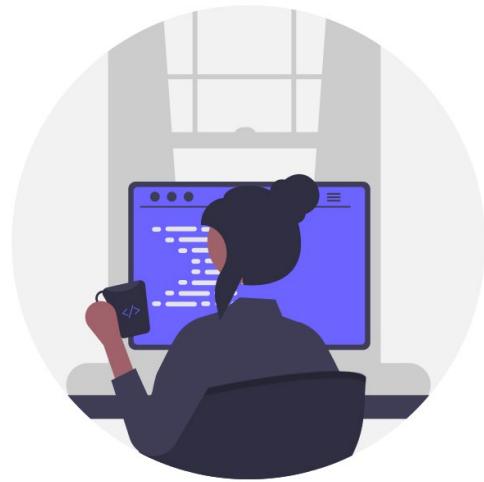
(A)	Publisher	(C)	Source
(B)	Current	(D)	Destination

10.12.1 Answers

1	C
2	A
3	B
4	D
5	A

Try It Yourself

1. Create a Java program that lists all files and directories in a specified directory using the new FileSystems methods.
2. Implement a Java program that uses Predicate.not() to filter a list of integers to find all numbers that are not divisible by three.
3. Implement a simple flow using the Flow API. Create a publisher and a subscriber to demonstrate the flow of data between them.



Session 11

Introduction to JavaFX

Welcome to the Session, **Introduction to JavaFX**.

This session provides a comprehensive introduction to JavaFX, a powerful framework for building modern, interactive, and visually appealing user interfaces for Java applications. This session will give an understanding of the fundamentals of JavaFX and its essential components.

In this Session, you will learn to:

- Describe an overview of JavaFX
- Summarize the history of JavaFX
- Explain installation process of Eclipse
- Elaborate JavaFX Architecture
- Explain how to work with JavaFX using Eclipse
- Explain JavaFX application structure
- Elaborate features of JavaFX

11.1 Overview of JavaFX

JavaFX is a powerful software framework for creating and delivering rich desktop applications with Java. It provides a modern and feature-rich Graphical User Interface (GUI) toolkit for Java developers. JavaFX also allows developers to build cross-platform applications with a highly interactive and visually appealing user interface.

JavaFX remains a separate library from the Java Standard Library in Java 20, requiring developers to include it as an additional dependency when using it in their applications. It offers a wide range of UI controls, multimedia support, 2D and 3D graphics capabilities, and a powerful scene graph for managing the user interface components.

11.1.1 Why Use JavaFX?

JavaFX offers several compelling reasons for developers to choose it over other GUI frameworks.

Some of the key advantages include:

Rich User Interface:

JavaFX provides a modern and attractive UI with support for animations, transitions, and styling, making it suitable for creating visually appealing applications.

Cross-platform Compatibility:

JavaFX applications can run on multiple platforms, including Windows, macOS, Linux, and even mobile devices, without major modifications.

Integration with Java:

JavaFX seamlessly integrates with Java, allowing developers to leverage the power of the Java language and its extensive ecosystem.

Community and Support:

JavaFX has an active developer community and is supported by Oracle, ensuring that developers can find resources and assistance when required.

11.2 History of JavaFX

JavaFX's story begins in the mid-2000s when Sun Microsystems (later acquired by Oracle Corporation) recognized the necessity for a modern platform to create Rich Internet Applications (RIAs). This was a time when Web applications were becoming more interactive and technologies such as Adobe Flash and Microsoft Silverlight were gaining popularity. Sun Microsystems decided to develop its own solution to compete in this space.

11.2.1 JavaFX 1.0 (2008)

JavaFX 1.0 was a groundbreaking release by Sun Microsystems (later acquired by Oracle Corporation), aimed at addressing the growing demand for rich and interactive Internet applications.

Some key aspects of JavaFX 1.0 were:

➤ **Focus on Web and Mobile:**

- **Web-Based Applications:** JavaFX 1.0 was primarily designed to cater to the requirements of Web-based applications. During this era, Web applications were evolving beyond static Web pages, and there was an increasing demand for more interactive and visually appealing user interfaces.
- **Mobile Devices:** In addition to Web applications, JavaFX 1.0 also had a strong emphasis on mobile devices, such as smartphones and feature phones. It aimed to enable developers to create JavaFX-powered mobile applications that could run on various Java-enabled mobile platforms.

- **Rich User Experience:** The central goal of JavaFX 1.0 was to provide a platform for creating applications that offered a richer, more engaging user experience compared to traditional Web and mobile apps.

➤ **Java Integration:**

- **JavaFX Script:** One of the distinctive features of JavaFX 1.0 was the introduction of a new scripting language called JavaFX Script. JavaFX Script was designed to be a concise and expressive language for building user interfaces and animations. It allowed developers to create complex UIs and animations with less code compared to traditional Java code.
- **Seamless Integration with Java:** While JavaFX Script was the scripting language for creating the UI and defining animations, it was closely integrated with Java. This integration allowed developers to leverage the power of the Java platform for the application's logic and business functionality. Developers could use Java alongside JavaFX Script to create complete applications.
- **JavaFX API:** JavaFX 1.0 also introduced a rich set of APIs that could be accessed from Java code to interact with and manipulate JavaFX Script-based UI components. This seamless integration between Java and JavaFX Script was a key selling point for JavaFX 1.0.

11.2.2 JavaFX 2.0 (2011)

JavaFX 2.0 came with following features:

- **Shift to Desktop:** JavaFX 2.0 marked a significant shift in focus from Web and mobile to desktop applications.
- **Rich UI Controls:** It introduced a rich set of UI controls and layouts, making it more competitive with other GUI frameworks such as Swing.
- **Java Integration:** Developers could continue to use Java alongside JavaFX, making it easier to migrate existing Java applications to JavaFX.
- **CSS Styling:** The introduction of Cascading Style Sheets (CSS) for styling UI elements provided a flexible way to design modern-looking interfaces.
- **FXML:** JavaFX 2.0 also introduced FXML, an XML-based markup language for creating JavaFX UIs declaratively.

11.2.3 JavaFX 8 (2014)

JavaFX 8, released in 2014, marked a milestone by being bundled with Java 8. This integration meant that JavaFX became an integral part of the Java Standard Library, making it easier for Java developers to work with JavaFX without requiring separate installations or dependencies. This integration also ensured that JavaFX could take full advantage of the enhancements and features introduced in Java 8.

Lambda Expressions:

One of the standout features of Java 8 was the introduction of lambda expressions, which allowed for more concise and expressive code. JavaFX 8 fully embraced lambda expressions for event handling and simplifying code in UI development.

Here is how lambda expressions benefited JavaFX:

Concise Event Handling:

With lambda expressions, JavaFX event handling became more straightforward and less verbose. Developers could define event handlers inline, making the code more readable and maintainable. For example, attaching an event handler to a button click became as simple as `(event) -> { /* handle event */ }.`

Improved Readability:

Lambda expressions improved the readability of JavaFX code by reducing the necessity for anonymous inner classes, which were commonly used for event handling in previous versions of Java.

Enhanced Functionality:

Lambda expressions allowed developers to pass behavior as a parameter to methods, which was particularly useful for customizing UI components and interactions.

3D Graphics:

JavaFX 8 introduced support for 3D graphics, expanding its capabilities beyond 2D graphics. This addition enabled developers to create more immersive and visually appealing applications. Key features of JavaFX 3D included:

- **3D Shapes and Models:** JavaFX 8 provided APIs for creating and rendering 3D shapes and models. Developers could create 3D scenes with objects such as spheres, cubes, and custom models.
- **3D Transformations:** The framework allowed for 3D transformations, such as translation, rotation, and scaling, making it possible to create complex 3D animations and visualizations.
- **Integration with 2D:** JavaFX 8 seamlessly integrated 3D graphics with existing 2D capabilities, enabling developers to build hybrid 2D/3D interfaces.

Performance Enhancements:

JavaFX 8 also brought substantial performance improvements, particularly in rendering complex user interfaces. The performance enhancements included:

- **Hardware Acceleration:** JavaFX leveraged hardware acceleration whenever possible, making rendering and animation smoother and more efficient.
- **Reduced Memory Footprint:** Efforts were made to optimize memory usage, resulting in more responsive applications and reduced memory consumption.
- **Improved Startup Time:** JavaFX applications launched faster, improving the overall user experience.

11.2.4 JavaFX 11

JavaFX underwent significant transformations, becoming an open-source with the release of JavaFX 11, fostering community contributions and enhancing its accessibility and flexibility. Some of the changes were:

- **Open Sourcing (2018):** In 2018, Oracle made a significant move by open-sourcing JavaFX, releasing it under the open-source GNU General Public License (GPL). This decision allowed the JavaFX community to actively participate in its development and contribute to its growth. Open sourcing opened up new possibilities for the framework, including bug fixes, feature enhancements, and better community support.
- **Separation of JavaFX 11:** With the release of JavaFX 11, the framework was decoupled from the Java Development Kit (JDK). This decoupling made JavaFX available as a separate module, which could be easily maintained and updated independently. This change simplified JavaFX distribution and allowed developers to use it with different Java versions.
- **Community Contributions:** The open-source nature of JavaFX encouraged community-driven contributions. Developers from around the world started actively participating in the development of JavaFX. They fixed bugs, added new features, and improved documentation. This increased collaboration enriched the platform and expanded its capabilities.

11.2.5 JavaFX 16

Subsequent releases of JavaFX continued to enhance the framework in various ways. Some of the notable developments in recent versions of JavaFX include:

- **Enhanced CSS Support:** CSS styling in JavaFX was improved and extended, making it easier for developers to create and manage modern and visually appealing UIs. This enhancement gave developers more flexibility and control over the look and feel of their applications.
- **WebView Enhancements:** JavaFX's WebView component, which allows embedding Web content within JavaFX applications, received updates to improve its compatibility with modern Web technologies. This made it easier to integrate Web content into JavaFX applications seamlessly.
- **Better Compatibility:** JavaFX maintained compatibility with the latest versions of the Java platform. This ensured that developers could leverage the latest features of both Java and JavaFX in their applications, keeping them up-to-date with industry standards.
- **Performance Improvements:** Ongoing efforts were made to further optimize the performance of JavaFX applications. These improvements included faster rendering, reduced memory consumption, and smoother animations, making JavaFX more efficient for demanding applications.

11.2.6 JavaFX 20/21

JavaFX 20 has been compiled with the `--release 17` option, making it compatible with JDK 17 or later versions for proper execution. Attempting to run JavaFX 20 with an older JDK will result in an error message, specifically indicating that the `javafx.base` module cannot be read.

Key enhancements in this version include following new APIs:

Enhanced Skin Class	ObservableValue Improvement	Additional Constrained Resize Policies for Tree/TableView
The Skin class in JavaFX now includes a new method named <code>install()</code> . This method enables skins to safely make various changes to their associated controls, such as registering listeners, adding child nodes, and modifying properties and event handlers.	The ObservableValue class has received an enhancement in the form of a new method named <code>when()</code> . This simplifies the management of listeners associated with observable values.	JavaFX 20 introduces additional constrained resize policies for TreeView and TableView components, offering greater control and flexibility when working with these UI elements.

11.3 JavaFX Architecture

JavaFX is a modern framework for building rich, interactive, and visually appealing cross-platform desktop applications. To understand how JavaFX works, it is essential to delve into its architecture, which comprises several key components and concepts. The components of JavaFX architecture are listed in Table 11.1.

JavaFX Architecture Components	Description
Scene Graph	A hierarchical structure representing the GUI, containing nodes for visual and non-visual elements.
Stage and Scene	The main window (Stage) that contains Scenes. Scenes serve as containers for content.
Nodes and Controls	Building blocks of the UI, including buttons, text fields, labels are represented as nodes.
Layout Managers	They are responsible for arranging nodes within a scene; options include VBox, HBox, GridPane, and more.
Event Handlers	Event-driven programming for responding to user interactions (for example, button clicks, and mouse events).
Animation and Transitions	Support for creating animations and transitions to enhance the visual appeal of the user interface.
Media and Multimedia	They provide capability to play audio and video, supporting various formats and streaming.
3D Graphics	They provide 3D rendering capabilities for creating 3D scenes and objects in applications.

JavaFX Architecture Components	Description
Java Integration APIs	Seamless integration with the Java language, allowing developers to leverage Java's ecosystem.
APIs or Libraries	Cross-platform compatibility, enabling applications to run on different OSes with minimal changes.

Table 11.1: Components of JavaFX Architecture

11.3.1 JavaFX Architecture Components in Detail

JavaFX, with its rich set of features and capabilities, empowers developers to create versatile and interactive GUI applications. Here is a comprehensive overview of the key components and functionalities that make JavaFX a powerful tool for building cross-platform applications:

➤ **Scene Graph:**

In JavaFX, the construction of GUI applications revolves around the utilization of a Scene Graph, which serves as the foundational framework. This Scene Graph serves as the canvas upon which GUI applications are crafted, holding various graphical elements referred to as **nodes**.

A node, in the context of JavaFX, encapsulates a visual or graphical entity within the GUI application. In essence, a collective assembly of nodes forms what is known as a **scene graph**. This scene graph acts as a hierarchical structure, allowing for the arrangement and organization of nodes in a structured manner. Nodes of scene graph are shown in Figure 11.1.

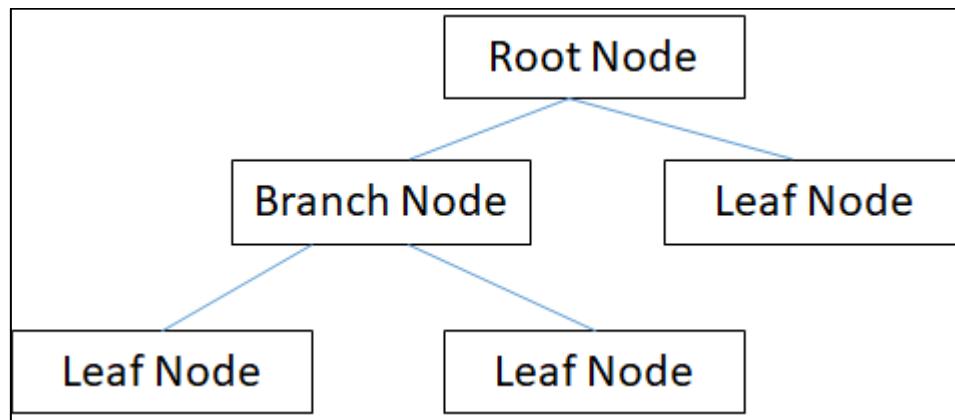


Figure 11.1: Nodes of Scene Graph

Nodes can encompass a wide range of elements, including:

Geometrical (Graphical) Objects:

This category includes both 2D and 3D shapes, such as circles, rectangles, polygons, and more.

Geometrical (Graphical) Objects:	This category includes both 2D and 3D shapes, such as circles, rectangles, polygons, and more.
UI Controls:	Nodes can also represent user interface controls such as buttons, check boxes, choice boxes, text areas, and numerous others.
Containers (Layout Panes):	Certain nodes, known as layout panes, function as containers for organizing and positioning other nodes. Examples of layout panes include Border Pane, Grid Pane, Flow Pane, and more.
Media Elements:	JavaFX enables the inclusion of media elements, encompassing audio, video, and image objects, as part of the node hierarchy.

Key characteristics of the scene graph include:

- **Parent-Child Relationships:** Each node within the scene graph maintains a parent-child relationship. A node with no parent is designated as the 'root node', signifying the starting point of the scene graph.
 - **Hierarchy:** Nodes in the scene graph are structured hierarchically. Nodes with no child nodes are identified as 'leaf nodes', while those with children are referred to as 'branch nodes'.
 - **Uniqueness:** It is imperative to note that a single node instance can be added to the scene graph only once, ensuring consistency within the structure.
 - **Customization and Interaction:** Nodes within the scene graph are versatile, permitting the incorporation of effects, opacity adjustments, transformations, and event handlers. These attributes empower developers to tailor the appearance and behavior of GUI elements and create interactive user experiences.
- **Stage and Scene:** In JavaFX, the application's main window is referred to as the 'Stage'. A Stage can contain one or more 'Scenes'. A Scene represents a container for the content of a stage. Developers can switch between scenes within the same stage to change the visible content dynamically. In a JavaFX application, it is common to work with three key components: Stage, Scene, and Nodes. These components together form the essential structure for building GUIs and interactive applications in JavaFX which are shown in Figure 11.2.

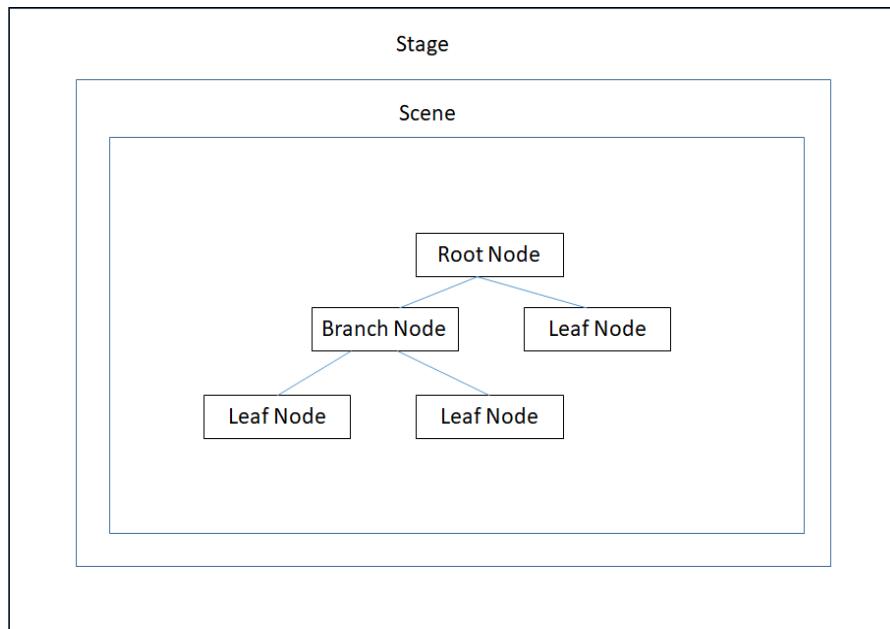


Figure 11.2: JavaFX Components

- **Layout Managers:** Layout managers are responsible for determining how nodes are arranged within a scene. JavaFX offers different layout managers, including VBox, HBox, GridPane, and more, to help to design responsive and well-organized user interfaces.
- **CSS Styling:** JavaFX allows developers to apply CSS styles to your UI components, providing extensive flexibility in designing the visual appearance of your application. One can define styles for individual nodes or use external CSS files to maintain consistency across your application.
- **Event Handlers:** Event-driven programming is fundamental in JavaFX. One can attach event handlers to nodes to respond to user interactions such as button clicks, mouse movements, and keyboard input. JavaFX leverages lambda expressions for concise event handling code.
- **Animation and Transitions:** JavaFX provides robust support for animations and transitions, allowing to create visually engaging user interfaces. Developers can animate node properties, apply transitions, and create complex animations with ease.
- **Media and Multimedia:** JavaFX supports media playback, making it possible to integrate audio and video into your applications. It provides features for playing media files, including support for various formats and streaming.
- **3D Graphics:** JavaFX has 3D graphics capabilities, enabling developers to render 3D scenes and objects within your applications. This feature is particularly useful for applications requiring 3D visualization or modeling.
- **Java Integration APIs:** JavaFX seamlessly integrates with the Java programming language. Developers can use JavaFX in conjunction with Java to develop complete applications. JavaFX APIs are accessible from Java code, allowing to leverage the extensive Java ecosystem.

- **APIs or Libraries:** One of JavaFX's strengths is its cross-platform compatibility. Developers can write JavaFX applications on one platform (for example, Windows) and run them on others (example, macOS and Linux) without significant modifications.

In order to work with JavaFX in applications, it is recommended to install Eclipse IDE.

11.4 Why Eclipse and not NetBeans?

Some of the key advantages of using Eclipse for JavaFX in place of NetBeans include:

Flexibility and ecosystem:	Eclipse is known for its flexibility and a wide range of plugins. You can find JavaFX plugins for Eclipse, such as e(fx)clipse, which can provide similar JavaFX development features as NetBeans.
Project Requirements	Consider the specific requirements of your JavaFX project. If you must integrate with other technologies, libraries, or frameworks, your choice of IDE might be influenced by the compatibility and availability of plugins for those technologies.
Adaptability	Eclipse is known for its adaptability. You can customize it extensively to suit your specific requirements. This can be advantageous when working on complex or specialized JavaFX projects where you may require to integrate with other technologies or libraries seamlessly.
Active Development	Eclipse has an active development community, and the e(fx)clipse plugin, specifically tailored for JavaFX development, is actively maintained. This means you can expect updates, bug fixes, and improvements that cater to JavaFX development requirements.

11.5 Installation of Eclipse

To install Eclipse for Windows, follow these steps:

1. **Download the Eclipse Installer**

To initiate the Eclipse installation process, first, obtain the Eclipse Installer by visiting the official download page at <http://www.eclipse.org/downloads>.

2. **Execute the Eclipse Installer**

For Windows users, once the Eclipse Installer executable has completed downloading, they can find it in their designated download directory. They should then execute the Eclipse Installer executable. One might encounter a security warning prompting to run this file. If the Publisher is identified as the Eclipse Foundation, they can proceed by selecting **Run**.

3. **Choose the Desired Package**

The new Eclipse Installer provides an array of packages available to Eclipse users. Developers have the option to either search for their preferred package or simply scroll

through the list. Once they have made their choice, they should select the package they wish to install. Figure 11.3 displays the package selection for Eclipse installation.

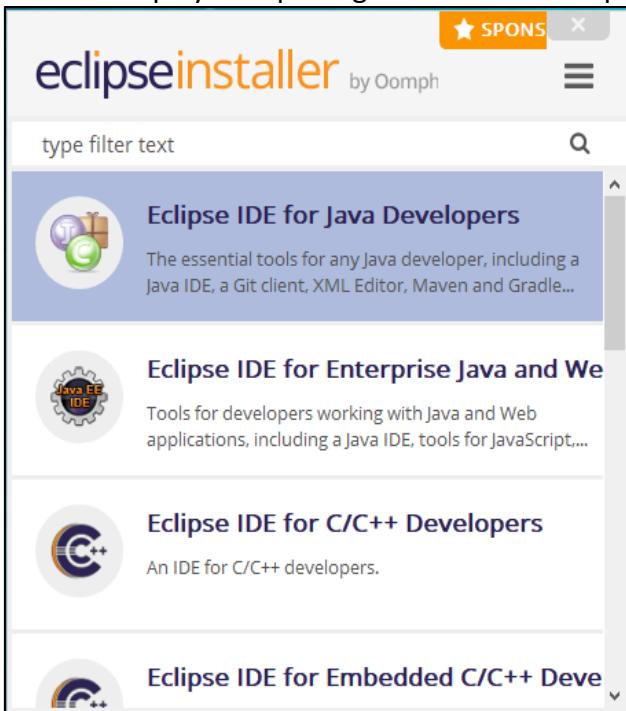


Figure 11.3: Package Selection for Eclipse Installation

4. Specify the Installation Directory

Indicate the directory where you intend to install Eclipse. By default, this folder will be located within the User directory. To commence the installation, click **Install** button. Figure 11.4 displays the directory selection for installation.

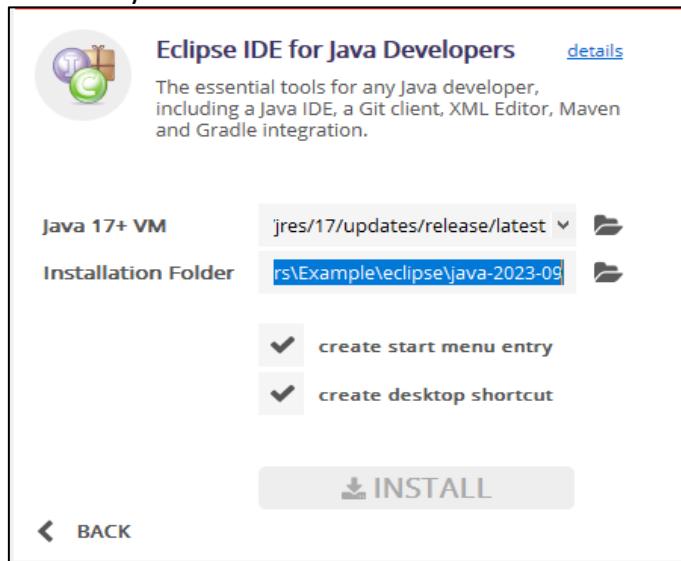


Figure 11.4: Directory Selection for Installation

5. Launch Eclipse

Following the completion of the installation process, you are now ready to launch Eclipse. The Eclipse Installer has successfully executed its tasks, and you can begin using Eclipse for your development requirements.

11.6 Getting Started with JavaFX in Eclipse

11.6.1 Installing JavaFX Plugin in Eclipse

To install JavaFX plugin, follow these steps:

Step 1: Open Eclipse.

Step 2: Click **Help** on the menu bar.

Step 3: Click **Eclipse Marketplace** as shown in Figure 11.5.

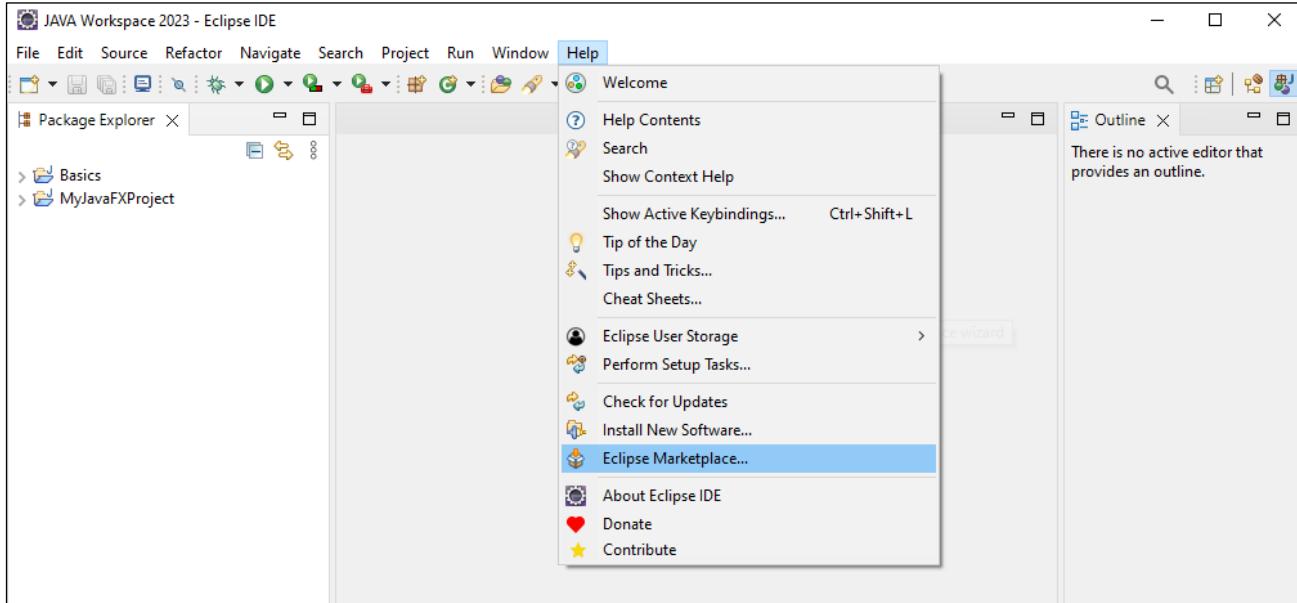


Figure 11.5: Eclipse Marketplace

Step 4: Search for fx and click **Install** as shown in Figure 11.6.

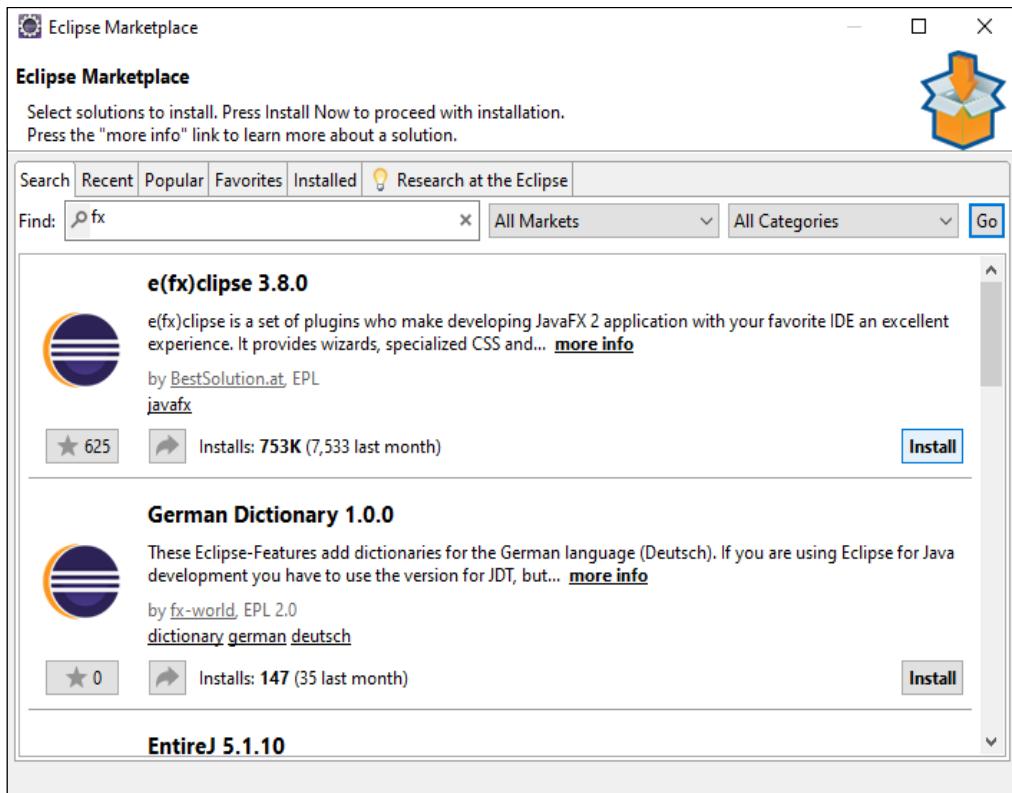


Figure 11.6: JavaFX Installation from Eclipse Marketplace

Step 5: Accept the agreement and let the installation be complete.

Step 6: Restart Eclipse IDE to apply the update.

11.6.2 Downloading and Configuring JavaFX

So far, the process for installing a plugin related to JavaFX in Eclipse was completed. This plugin offers specific features or tools for JavaFX development within the Eclipse IDE. Now, the steps to download and configure JavaFX SDK (which is essential for developing JavaFX applications regardless of any specific Eclipse plugin) are as follows:

Step 1: Download JavaFX SDK for Windows from <https://gluonhq.com/products/javafx/> as shown in Figure 11.7.

Figure 11.7: Downloading of JavaFX SDK

Step 2: Extract the ZIP file to the local system.

11.6.3 Adding JavaFX Library in Eclipse

Steps to add JavaFX library to Eclipse are as follows:

Step 1: In Eclipse, select **Window** and then, **Preferences**. The Preferences dialog box is displayed as shown in Figure 11.8.

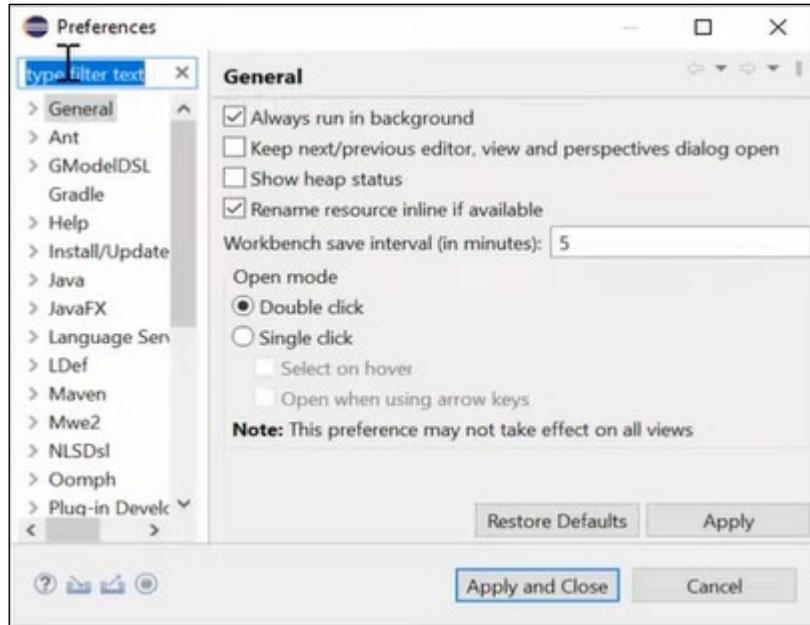


Figure 11.8: Adding JavaFX Library

Step 2: Search for **User Library** in Preferences and select it as shown in Figure 11.9.

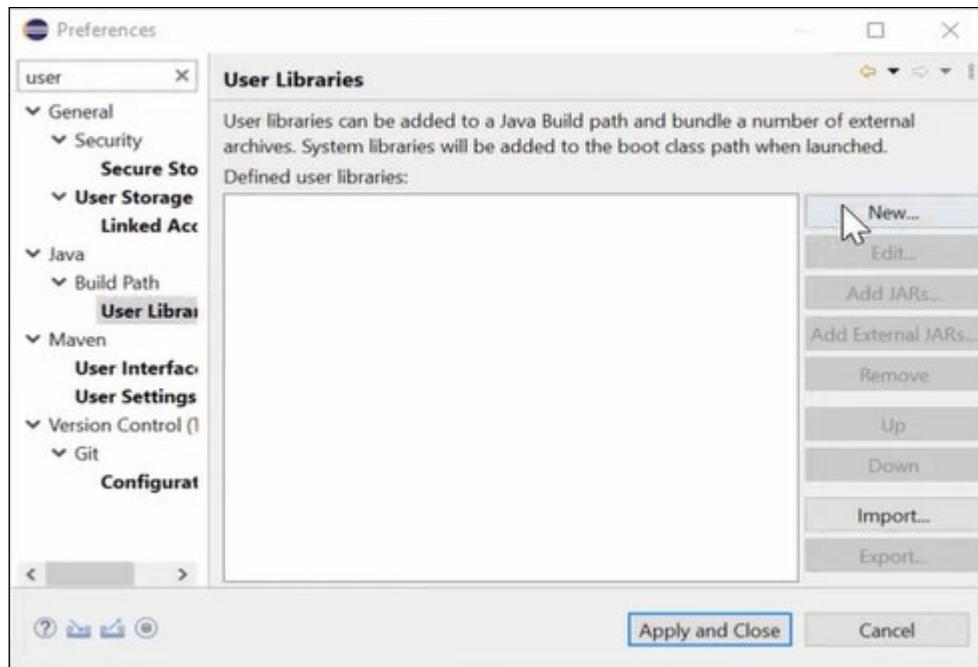


Figure 11.9: Adding SDK External JAR

Step 3: Select **New** and give library name (such as **JavaFX**).

Step 4: Select **Add External JARs** as shown in Figure 11.10.

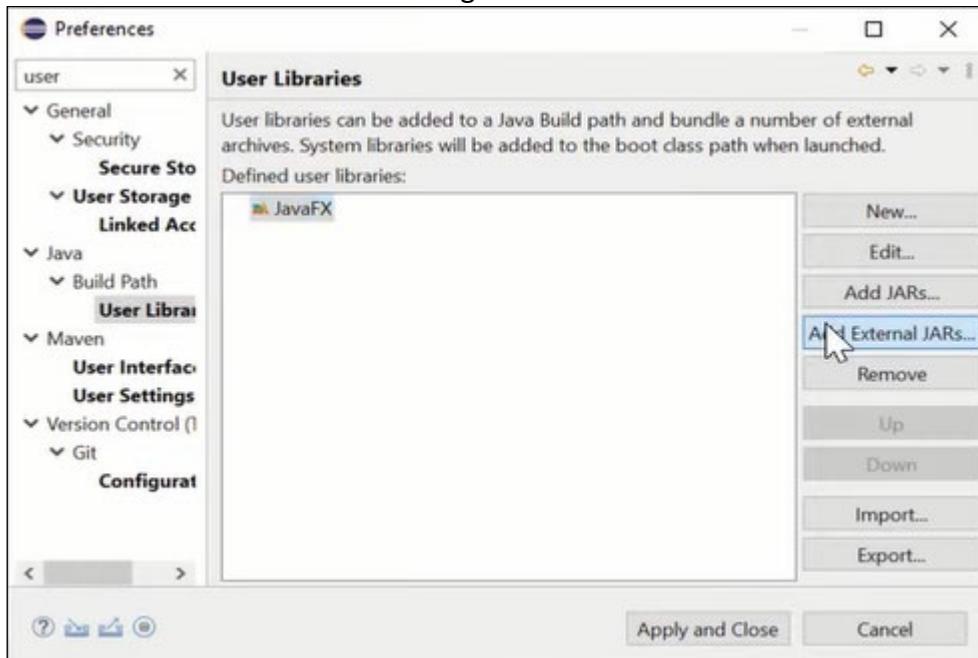


Figure 11.10: Adding JAR Path

Step 5: Then, go to extracted SDK of JavaFX and go to **lib** folder, copy path, and add the files as External JARs.

This user library can then, be added to every JavaFX project that is created. Unless it is added, Eclipse will not recognize the built-in JavaFX classes and interfaces.

11.6.4 Creating First JavaFX Project

To create a JavaFX in Eclipse, following steps must be done:

Step 1: Create a new empty project by selecting **File → New**.

Step 2: Select **Other**.

Step 3: Search for JavaFX as shown in Figure 11.11.

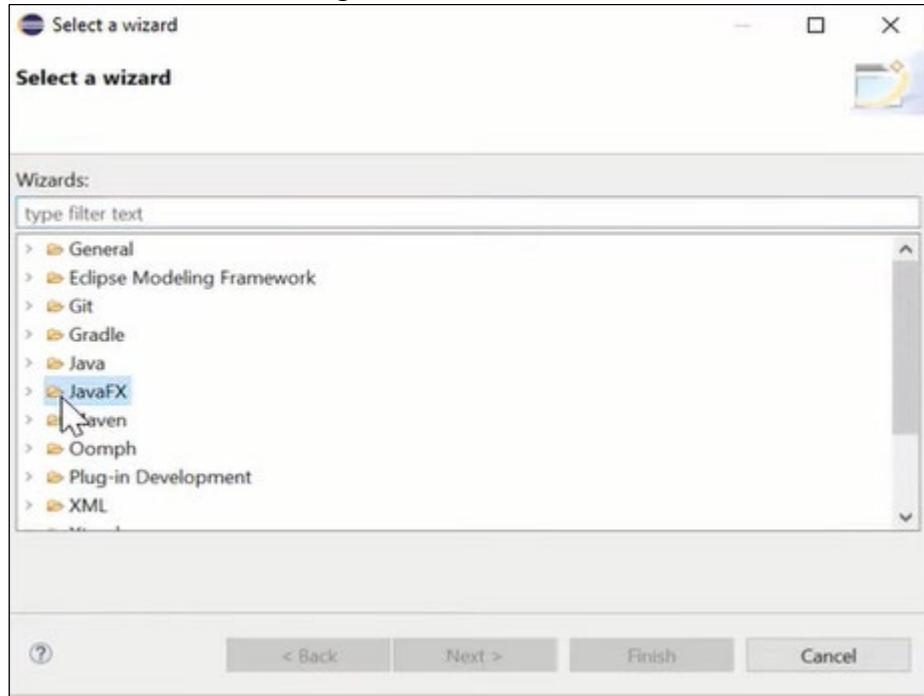


Figure 11.11: Create New JavaFX Project

Step 4: Select JavaFX project as shown in Figure 11.12 and provide a project name.

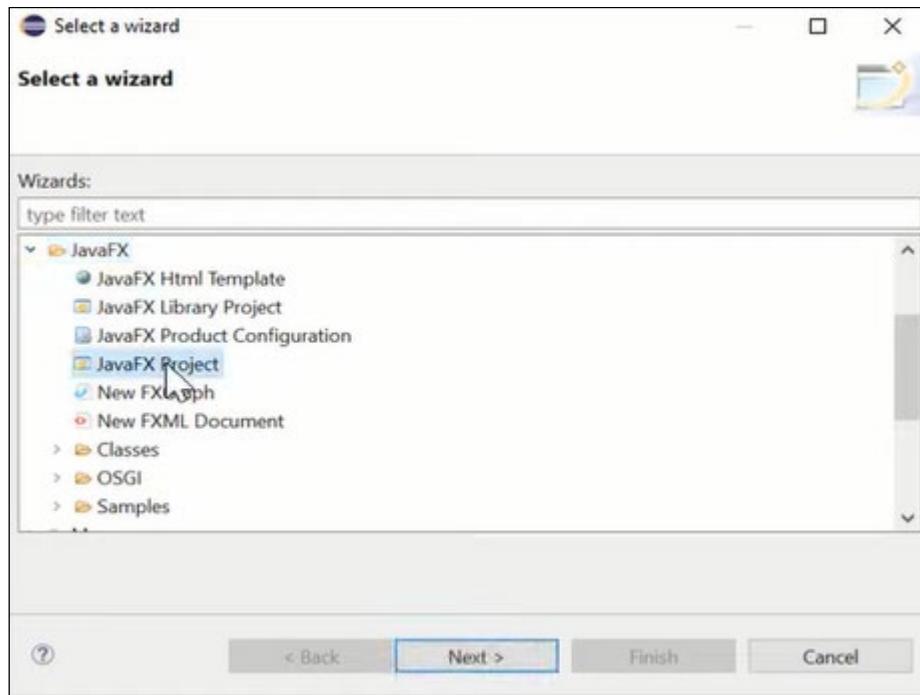


Figure 11.12: Selecting New JavaFX Project

Step 5: Retain the default options in subsequent screens of the wizard.

Step 6: The project will be created with auto-generated code.

Step 7: Right-click the project, select **Build → Configure Build Path** and add the user library created earlier into this project. Then, click **Apply and Close**.

Step 8: Replace the default code in Main.java (under src folder) with the code shown in Code Snippet 1.

Code Snippet 1 illustrates a basic JavaFX program in which a graphical window with a single label that says **Hello, JavaFX!** The layout is organized using a StackPane and the application starts with the `main` method and the `start` method being called automatically, setting up and displaying the UI components.

Code Snippet 1:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorldJavaFX extends Application {

    @Override
    public void start(Stage primaryStage) {
        // Create a Label with the text "Hello, JavaFX!"
        Label label = new Label("Hello, JavaFX!");
    }
}
```

```

// Create a StackPane layout and add the Label to it
StackPane root = new StackPane();
root.getChildren().add(label);

// Create a Scene with the StackPane as its root
Scene scene = new Scene(root, 300, 200);

// Set the Scene on the Stage (window)
primaryStage.setScene(scene);

// Set the title of the Stage
primaryStage.setTitle("JavaFX Hello World");

// Show the Stage
primaryStage.show();
}

public static void main(String[] args) {
    // Launch the JavaFX application
    launch(args);
}
}

```

When this JavaFX program is run, it will display a window with the text "Hello, JavaFX!" centered on it as shown in Figure 11.13.

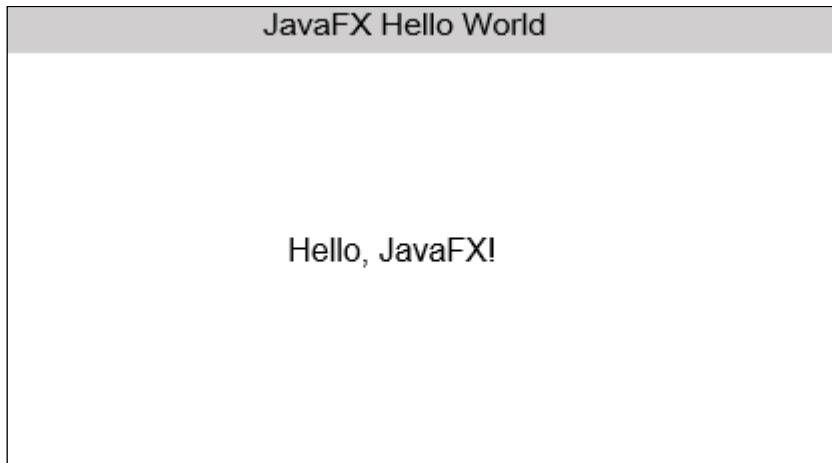


Figure 11.13: Output of Code Snippet 1

Similarly other applications can be created. Code Snippet 2 illustrates code to create a JavaFX application with a button and a label that keeps track of how many times the button has been clicked as shown in Figure 11.14.

Code Snippet 2:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;

```

```

import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ClickCounter extends Application {

    private int count = 0;

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Click Counter");

        Button button = new Button("Click me!");
        Label label = new Label("Click count: 0");

        button.setOnAction(e -> {
            count++;
            label.setText("Click count: " + count);
        });

        VBox layout = new VBox(10);
        layout.getChildren().addAll(button, label);

        Scene scene = new Scene(layout, 300, 200);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

In Code Snippet 2, import statements are used to import necessary JavaFX classes for building the application. The code `public class ClickCounter extends Application` defines a class named ClickCounter that extends Application, indicating that it is a JavaFX application. The code `private int count = 0;` is used to keep track of the number of clicks and is initially set to 0. `main` is the application's entry point. It launches the JavaFX application by calling `launch(args)`. Figure 11.14 depicts the output.

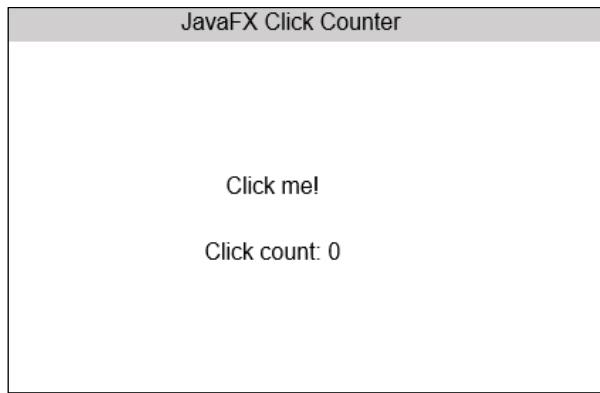


Figure 11.14: Output of Code Snippet 2

11.7 JavaFX Application Structure

When developing JavaFX applications, it is essential to understand the typical structure that these applications follow. A well-organized structure ensures maintainability, scalability, and ease of development. Here is an overview of the JavaFX application structure:

1. Main Class:

- Every JavaFX application starts with a main class that contains the `main` method. This class serves as the entry point for the application.
- In the `main` method, the JavaFX application is launched by calling `launch()` from the `Application` class, passing the application's main class as an argument.

Code Snippet 3 illustrates example of using `Application` class.

Code Snippet 3:

```
public class MainApp extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) {  
        // Initialize and set up the primary stage  
        // ...  
    }  
}
```

2. Application Class:

- The main class extends the `Application` class, which is part of the JavaFX framework.
- The `Application` class provides the `start(Stage primaryStage)` method, where one can initialize and configure the primary stage of the application.

3. Primary Stage:

- The primary stage represents the main window of your JavaFX application.
- Developer can set up the primary stage's properties, such as title, dimensions, and layout, in the `start()` method.

- This stage can contain one or more scenes, and it is where the user interface (UI) is displayed.

4. Scenes:

- A scene is a container that holds the UI elements of your application.
- Developers can create multiple scenes and switch between them as required.
- Each scene is associated with a root node, which is the top-level node in the scene graph.

5. FXML (Optional):

- For complex UI layouts, JavaFX provides FXML, an XML-based markup language. FXML allows to define the structure and appearance of the UI in a separate file.
- FXML files can be loaded and associated with controllers in the Java code.

6. Controllers:

- Controllers are Java classes that define the behavior of the UI components.
- They interact with the UI elements defined in the FXML file (if used) and handle user interactions and application logic.
- Controllers are typically associated with specific FXML files or scenes.

7. CSS Styling (Optional):

- Developers can apply CSS styles to your JavaFX UI elements to control their appearance.
- CSS allows for easy customization and theming of your application's UI.

8. Event Handling:

- JavaFX applications respond to user interactions through event handling.
- Developers can attach event handlers to UI elements to define what happens when events such as button clicks or mouse movements occur.

9. Resources and Assets:

- Developers may include resources such as images, icons, and fonts in the application.
- These resources can be loaded and used within your JavaFX application.

10. Project Organization:

- Organize the project into packages to structure your code logically.
- Follow best practices for separation of concerns, modular design, and code reusability.

11. Build and Deployment:

- Once the JavaFX application is developed, developers can build it into executable JAR files or native installers for various platforms.

11.8 Features of JavaFX

Features of JavaFX are listed in Table 11.2.

Feature	Description
Rich User Interfaces (UIs)	Comprehensive set of customizable UI controls and components for creating modern interfaces.
Scene Graph	Hierarchical structure simplifying the organization and management of UI components.
FXML Support	XML-based markup language for separating UI layout from application logic.

Feature	Description
3D Graphics	Support for creating 3D scenes and objects within applications.
Multimedia Integration	Features for playing audio, video, and working with images.
Event Handling	Event-driven programming for responding to user interactions.
Layout Managers	Variety of layout managers for precise control over UI component positioning and sizing.
CSS Styling	Application of CSS styles for customization and theming of UI elements.
Concurrency and Multithreading	Support for multithreading and UI updates on the JavaFX Application Thread.
Modularity	Modular structure allowing inclusion of only required components.
Cross-Platform Compatibility	Ability to run on multiple platforms, including Windows, macOS, Linux, and mobile devices.
Integration with Java	Seamless integration with the Java language and ecosystem.
Open Source	Open-source framework with an active community and ongoing development.
Web Integration	WebView for embedding Web content within applications.
Accessibility	Built-in accessibility features for usability by individuals with disabilities.
Internationalization and Localization	Support for multiple languages and locales.
Deployment Options	Packaging as executable JAR files, native installers, or deployment via Web browsers.
Security	Benefit from the security features of the Java platform, including sandboxing and code signing.
Community Support	Active developer community and extensive documentation resources.

Table 11.2: Features of JavaFX

11.9 Summary

- JavaFX is a powerful framework for building cross-platform desktop applications.
- JavaFX provides a rich set of UI controls, 2D and 3D graphics, and multimedia support.
- JavaFX applications use a scene graph for UI organization.
- In JavaFX Nodes represent visual and non-visual elements in a hierarchical structure.
- Scene, Stage, and Nodes are key components in JavaFX.
- Application class extends Application and provides the start method.
- In JavaFX UI components, scenes, FXML, controllers, CSS, event handling, and resources are structured logically.

11.10 Check Your Progress

1. What was the primary focus of JavaFX 1.0 when it was initially released in 2008?
 - A. Desktop applications
 - B. Web-based and mobile applications
 - C. 3D graphics
 - D. Game development

2. Which Java version was JavaFX 8 bundled with?
 - A. Java 7
 - B. Java 8
 - C. Java 11
 - D. Java 14

3. What is the primary function of the JavaFX Scene Graph?
 - A. Data storage
 - B. Handling user input
 - C. UI organization
 - D. Database management

4. Which JavaFX component represents the main window of an application?
 - A. Node
 - B. Scene
 - C. Stage
 - D. Controller

5. What is the purpose of FXML in JavaFX applications?
 - A. Handling events
 - B. Defining UI layout
 - C. Implementing business logic
 - D. Managing database connections

6. Which JavaFX feature allows developers to create visually appealing and interactive user experiences?
 - A. Event handling
 - B. 3D graphics
 - C. CSS styling
 - D. Layout managers

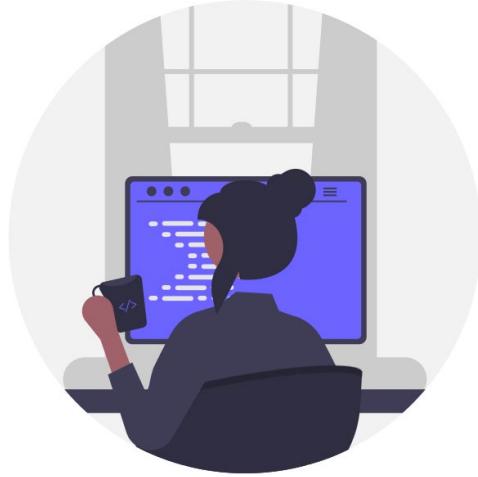
7. What does CSS provide in JavaFX?
 - A. Database connectivity
 - B. UI controls
 - C. Styling for UI elements
 - D. Animation support

11.10.1 Answers

1. B
2. B
3. C
4. C
5. B
6. D
7. C

Try It Yourself

1. Create a JavaFX application that converts temperatures between Celsius and Fahrenheit. Include text fields for input and labels to display the results.
2. Develop a JavaFX program that counts the number of characters in a text entered into a text field. Display the character count when a button is clicked.
3. Create an application with a text field for entering a name and a button. When the button is clicked, display a greeting message, such as "Hello, [Name]!".



Session 12

JavaFX Text, Transformation, and Shapes

Welcome to the Session, **JavaFX Text, Transformation, and Shapes**.

In JavaFX, developers have access to a rich set of features for working with text, transformations, and shapes, each contributing to the creation of compelling user interfaces and engaging user experiences. The session explains JavaFX Text, Transformation, and Shapes in detail.

In this Session, you will learn to:

- Explain JavaFX Text
- Identify JavaFX Text Properties
- Explain JavaFX Transformation
- Describe JavaFX Translation
- Explain JavaFX Rotation
- Describe JavaFX Scaling
- Explain JavaFX Shearing
- Describe JavaFX 2D and 3D Shapes

12.1 JavaFX Text

JavaFX Text is a versatile component within the JavaFX library used to display text and manipulate its properties in GUIs. Whether there is a requirement to create labels, headings, or any other textual content within a JavaFX application, the `Text` class provides a powerful solution for text rendering and customization.

Imagine we are creating a simple weather application that displays the current weather conditions for a city. In this application, we can use the `Text` component to show the city name and the temperature. The image for such a GUI is shown in Figure 12.1.

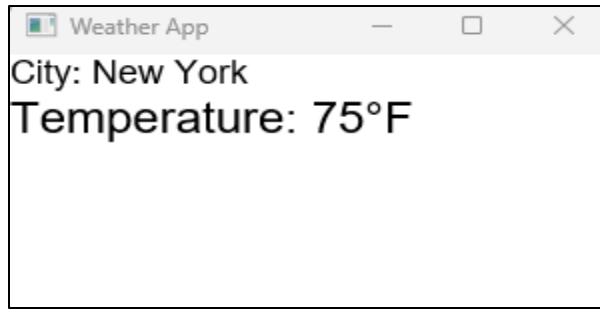


Figure 12.1: Image of JavaFX GUI

Some of the key features of JavaFX Text component are as follows:

- **Text Content:** The primary function of JavaFX Text component is to display text content.
- **Font Customization:** JavaFX Text allows for precise control over text appearance through font customization.
- **Color Control:** This property determines the text foreground color.
- **Text Alignment:** This property determines the text alignment and adjustment allowing to position the text within its layout bounds. Common alignment values include LEFT, CENTER, and RIGHT.
- **Text Wrapping:** This property is used to enable text wrapping within a specified width. This is particularly useful for ensuring that text fits within a certain area.
- **Text Effects:** JavaFX Text supports various text effects, such as drop shadows and inner shadows allowing for creative text styling.

Table 12.1 describes some of the commonly used methods of Text.

Method	Description
<code>setText()</code>	This method is used to set the text to be displayed.
<code>setFont()</code>	This method is used to specify the font family, size, and style (For example, bold or italic).
<code>setFill()</code>	This method is used to set the fill color of the text.
<code>setStroke()</code>	This method is used to define the color of the text outline or stroke.
<code>setStrokeWidth()</code>	This method is used to control the thickness of the outline.
<code>setTextAlignment()</code>	This method is used to align and adjust the text.
<code>setWrappingWidth()</code>	This method is used to enable the text wrapping within a specified width.
<code>setEffect()</code>	This method supports various text effects, such as drop shadows and inner shadows.

Table 12.1: Methods of JavaFX Text

Code Snippet 1 illustrates the implementation of the JavaFX Text. Create a new JavaFX project named **JavaFXTextExample** in Eclipse and then, add this code to the class.

Code Snippet 1:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
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 JavaFXTextExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Text Example");

        // Create a Text node
        Text text = new Text("Hello, JavaFX!");

        // Customize the font
        Font font = Font.font("Arial", FontWeight.BOLD,
            FontPosture.ITALIC, 24);
        text.setFont(font);

        // Set text fill color
        text.setFill(Color.BLUE);

        // Set text outline color and width
        text.setStroke(Color.BLACK);
        text.setStrokeWidth(1.5);

        // Set text alignment
        text.setTextAlignment(javafx.scene.text.TextAlignment.CENTER);

        // Enable text wrapping
        text.setWrappingWidth(200);

        StackPane root = new StackPane();
        root.getChildren().add(text);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

- The code begins by importing the necessary JavaFX classes and setting up the main class, `JavaFXTextExample`, which extends `Application`.
- Create a new JavaFX stage (`primaryStage`) in `start` method having the title "JavaFX Text Example."
- Create a `Text` object named `text` and set its content to "Hello, JavaFX!". This represents the text that will be displayed in the application.
- Next, customize the font for the text using the `Font` class. In this case, use Arial font, set it to bold (`FontWeight.BOLD`), and apply italic style (`FontPosture.ITALIC`). The font size is set to 24 points.
- Set the fill color of the text to blue using `text.setFill(Color.BLUE)`. This determines the color of the text itself.
- Additionally, set the outline (stroke) color of the text to black using `text.setStroke(Color.BLACK)` and specify the stroke width as 1.5 units using `text.setStrokeWidth(1.5)`. This gives the text a black outline.
- Set the fill color of the text to blue using `text.setFill(Color.BLUE)`. This determines the color of the text itself.
- Additionally, set the outline (stroke) color of the text to black using `text.setStroke(Color.BLACK)` and specify the stroke width as 1.5 units using `text.setStrokeWidth(1.5)`. This gives the text a black outline.
- Set the text alignment to center using `text.setTextAlignment(TextAlignment.CENTER)`. This positions the text in the center of its layout bounds.
- Enable text wrapping within a specified width, use `text.setWrappingWidth(200)`. This means that if the text exceeds a width of 200 units, it will automatically wrap to the next line.
- Create a `StackPane` named `root` and add the `text` object as a child to it. `StackPane` is a simple layout container that stacks its children on top of each other. Here, it contains only the `text` node.
- Create a `Scene` object named `scene` and initialize it with the `root` layout and dimensions of 300x150 pixels. The `Scene` represents the content of the stage.
- Finally, set the `scene` as the content of the `primaryStage` using `primaryStage.setScene(scene)` and show the `primaryStage` using `primaryStage.show()`.

When this JavaFX application is executed, it displays a window with the customized text "Hello, JavaFX!" in the center as shown in Figure 12.2. The text is blue, with a black outline and it wraps to the next line if its width exceeds 200 units. The font is Arial, bold, and italicized.



Figure 12.2: Output of JavaFX Text on Eclipse IDE

12.1.1 JavaFX Text Properties

JavaFX Text properties refer to various attributes and settings that can be applied to JavaFX Text objects to customize their appearance and behavior. These properties allow developers to precisely control how text is rendered within a JavaFX application.

Code Snippet 2 illustrates the use of JavaFX Text properties.

Code Snippet 2:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class JavaFXTextPropertiesExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Text Properties Example");

        // Create a Text node
        Text text = new Text("JavaFX Text Properties");

        // Customize the font
        Font font = Font.font("Arial", 24);
        text.setFont(font);

        // Set text fill color
        text.setFill(Color.BLUE);
```

```

        // Set text outline color and width
        text.setStroke(Color.BLACK);
        text.setStrokeWidth(1.5);

        // Set text alignment
        text.setTextAlignment(javafx.scene.text.TextAlignment.CENTER);

        // Enable text wrapping
        text.setWrappingWidth(200);

        StackPane root = new StackPane();
        root.getChildren().add(text);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

- The code starts by importing necessary JavaFX classes and defining the **JavaFXTextPropertiesExample** class, which extends **Application**.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX Text Properties Example". The stage represents the main window of a JavaFX application.
- A **Text** object named **text** is created with the content "JavaFX Text Properties." This **Text** object represents the text that will be displayed in the application.
- Font customization is applied using the **Font** class. In this case, the font is set to Arial with a font size of 24 points.
- The fill color of the text is set to blue using **text.setFill(Color.BLUE)**. This determines the color of the text itself.
- Additionally, an outline (stroke) color is applied to the text using **text.setStroke(Color.BLACK)** and the stroke width is set to 1.5 units with **text.setStrokeWidth(1.5)**. This gives the text a black outline.
- Text alignment is adjusted to center using **text.setTextAlignment(javafx.scene.text.TextAlignment.CENTER)**. This positions the text content in the center of its layout bounds.
- Text wrapping is enabled with **text.setWrappingWidth(200)**, which means that if the text exceeds a width of 200 units, it will automatically wrap to the next line.
- A **StackPane** named **root** is created, and the **text** object is added as its child. The **StackPane** is a layout container that stacks its children on top of each other. Here, it contains only the **text** node.
- A **Scene** object named **scene** is created with the **root** layout and dimensions of 300x150 pixels. The **Scene** represents the content of the **primaryStage**.
- Finally, the **scene** is set as the content of the **primaryStage** using **primaryStage.setScene(scene)** and the **primaryStage** is displayed using **primaryStage.show()**.

When this JavaFX application is executed, it displays a window with the customized text "JavaFX Text Properties" in the center as shown in Figure 12.3. The text is in Arial font, sized at 24 points, with blue fill color, a black outline, and text wrapping enabled.



Figure 12.3: Output of JavaFX Text Properties on Eclipse IDE

12.2 JavaFX Transformation

In JavaFX, transformations are a fundamental concept used to manipulate the position, size, and orientation of graphical objects, such as shapes and text. Transformations allow developers to create animations, adjust the layout of UI elements, and apply various visual effects. JavaFX provides several types of transformations, including translation, rotation, scaling, and shearing.

Example: A Simple Educational Game

Imagine you are creating an educational game for children that teaches them about basic geometry and shapes. In this game, you want to show a shape (let us say a square) and allow the user to interact with it using transformations. The types of transformations can be:

- **Translation:** You can use translation to move the square around the screen. For instance, you can ask the child to drag the square to a specific location. By applying translation transformations, the square can smoothly move to the desired position when the child interacts with it.
- **Rotation:** To teach about angles and orientation, you can apply rotation transformations. For example, you can ask the child to rotate the square by a certain number of degrees. This helps them visually understand concepts such as clockwise and counterclockwise rotation.
- **Scaling:** Scaling transformations can be used to make the square larger or smaller. For instance, you can ask the child to resize the square to be twice as big or half its size. This demonstrates the concept of scaling in a fun and interactive way.
- **Shearing:** Shearing transformations can be applied to skew the square. This can be used to teach concepts such as parallelograms. For example, you can ask the child to shear the square to transform it into a parallelogram shape.

These types of actions can also be performed in JavaFX.

Common Types of Transformations in JavaFX:

Translation Transformation	Rotation Transformation	Scaling Transformation	Shearing Transformation
In JavaFX, use Translate class to apply translation transformations to nodes, including text and shapes.	In JavaFX, the Rotate class is commonly used for this purpose. It allows to control the rotation angle and pivot point, enabling objects to rotate around a specific axis.	The Scale class is used for applying scaling transformations. To specify scaling factors to make an object larger or smaller and can control the scaling pivot point.	The Shear class can be used to apply shearing transformations. To set the shear factors to achieve the desired shearing effects.

Code Snippet 3 illustrates the use of JavaFX transformations by applying translation, rotation, scaling, and shearing to a simple rectangle.

Code Snippet 3:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.transform.*;

public class JavaFXTransformationsExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Transformations Example");

        Rectangle rectangle = new Rectangle(50, 50, 100, 50);
        rectangle.setFill(Color.BLUE);

        // Apply transformations
        Translate translate = new Translate(50, 50);
        Rotate rotate = new Rotate(30, 75, 75);
        Scale scale = new Scale(1.5, 1.5, 75, 75);
        Shear shear = new Shear(0.3, 0.2);

        rectangle.getTransforms().addAll(translate, rotate, scale,
shear);
```

```

        StackPane root = new StackPane();
        root.getChildren().add(rectangle);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

- The code begins by importing the necessary JavaFX classes and defining the **JavaFXTransformationsExample** class, which extends **Application**.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX Transformations Example." The stage represents the main window of a JavaFX application.
- A Rectangle object named **rectangle** is created with initial position (50, 50) and dimensions (100, 50). This rectangle will serve as the graphical object that undergoes transformations.
- The rectangle is filled with a blue color using **rectangle.setFill(Color.BLUE)**.
- Four types of transformations are applied to the **rectangle**:
 - Translation:** A Translate transformation named **translate** is created with X and Y translation values of (50, 50). This moves the rectangle 50 units to the right and 50 units down.
 - Rotation:** A Rotate transformation named **rotate** is created with an angle of 30 degrees and a pivot point at (75, 75). This rotates the rectangle by 30 degrees around the point (75, 75).
 - Scaling:** A Scale transformation named **scale** is created with scaling factors of (1.5, 1.5) and a pivot point at (75, 75). This scales the rectangle to 1.5 times its original size around the point (75, 75).
 - Shearing:** A Shear transformation named **shear** is created with shear factors of (0.3, 0.2). This skews the rectangle along the X and Y axes.
- All four transformations are added to the **rectangle** using **rectangle.getTransforms().addAll(...)**. This means that the rectangle will undergo all these transformations.

When this JavaFX application is executed, it displays a window with a blue rectangle that has undergone translation, rotation, scaling, and shearing transformations as shown in Figure 12.4.

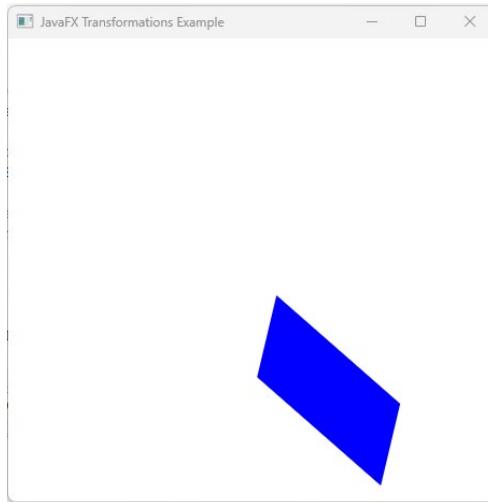


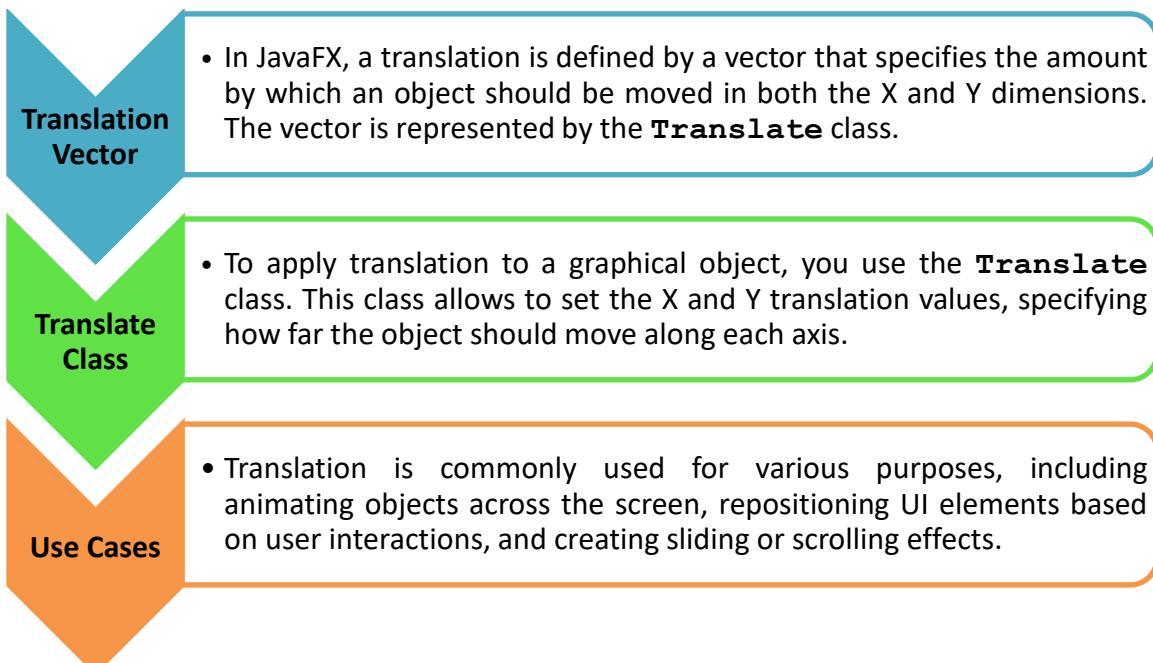
Figure 12.4: Output of JavaFX Transform on Eclipse IDE

Note: Resize the output window if the shape is not clearly visible.

12.3 JavaFX Translation

JavaFX Translation is a transformation that allows to move graphical objects, such as shapes or nodes, along a specified vector in a JavaFX application. It is a fundamental transformation for creating animations and positioning UI elements dynamically.

Key Points about JavaFX Translation:



Code Snippet 4 illustrates how to apply translation to a JavaFX shape, such as a rectangle.

Code Snippet 4:

```
import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.transform.Translate;

public class JavaFXTranslationExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Translation Example");

        Rectangle rectangle = new Rectangle(50, 50, 100, 50);
        rectangle.setFill(Color.BLUE);

        // Apply translation
        Translate translate = new Translate(50, 50);
        rectangle.getTransforms().add(translate);

        StackPane root = new StackPane();
        root.getChildren().add(rectangle);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

- The code starts by importing the necessary JavaFX classes and defining the **JavaFXTranslationExample** class, which extends **Application**.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX Translation Example." The stage represents the main window of a JavaFX application.
- A **Rectangle** object named **rectangle** is created with an initial position at (50, 50) and dimensions of 100x50 pixels. This rectangle will serve as the graphical object that undergoes the translation transformation.
- The **rectangle** is filled with a blue color using **rectangle.setFill(Color.BLUE)**.
- A **Translate** transformation named **translate** is created with X and Y translation values of (50, 50). This means that the rectangle will be moved 50 units to the right and 50 units down from its original position.
- The **translate** transformation is added to the **rectangle** using **rectangle.getTransforms().add(translate)**, indicating that the translation transformation should be applied to the rectangle.

When this JavaFX application is executed, it displays a window with a blue rectangle as shown in Figure 12.5. The rectangle has undergone a translation transformation, moving 50 units to the right and 50 units down from its original position.

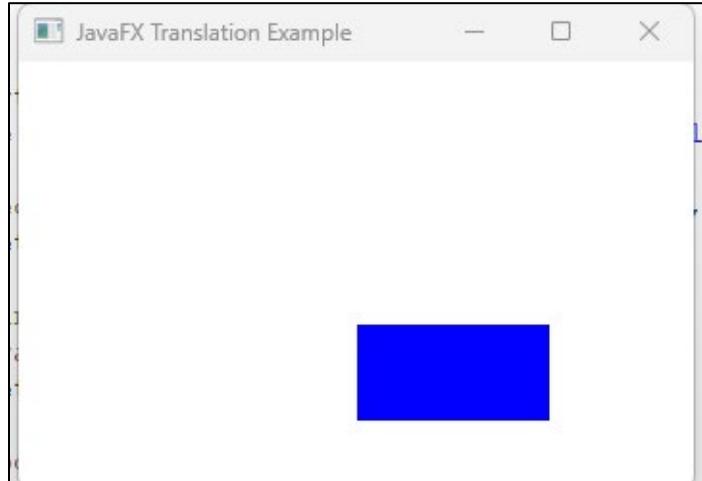


Figure 12.5: Output of JavaFX Translation on Eclipse IDE

12.4 JavaFX Rotation

JavaFX Rotation is a transformation that allows developers to rotate graphical objects, such as shapes or nodes, around a specified pivot point. Rotation is a fundamental transformation for creating animations, spinning objects, and achieving dynamic visual effects in JavaFX applications.

Key Points about JavaFX Rotation:

Rotation Angle	Pivot Point	Rotate Class	Use Cases
<ul style="list-style-type: none">In JavaFX, rotation is defined by an angle in degrees. You specify how much an object should be rotated, and it rotates clockwise by default.	<ul style="list-style-type: none">Rotation occurs around a pivot point, which is a fixed point around which the object rotates. You can choose the pivot point's location to control the rotation effect.	<ul style="list-style-type: none">To apply rotation to a graphical object, you use the Rotate class. This class allows you to set the rotation angle, pivot point coordinates, and other properties.	<ul style="list-style-type: none">Rotation is commonly used for creating spinning animations, rotating images, creating clock hands, and achieving various visual effects in user interfaces.

Code Snippet 5 illustrates how to apply rotation to a JavaFX shape, such as a rectangle.

Code Snippet 5:

```
import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.transform.Rotate;

public class JavaFXRotationExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Rotation Example");

        Rectangle rectangle = new Rectangle(50, 50, 100, 50);
        rectangle.setFill(Color.BLUE);

        // Apply rotation
        Rotate rotate = new Rotate(30, 75, 75);
        // 30 degrees, pivot at (75, 75)
        rectangle.getTransforms().add(rotate);

        StackPane root = new StackPane();
        root.getChildren().add(rectangle);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

- The code starts by importing the necessary JavaFX classes and defining the **JavaFXRotationExample** class, which extends **Application**.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX Rotation Example." The stage represents the main window of a JavaFX application.
- A **Rectangle** object named **rectangle** is created with an initial position at (50, 50) and dimensions of 100x50 pixels. This rectangle will serve as the graphical object that undergoes the rotation transformation.
- The rectangle is filled with a blue color using **rectangle.setFill(Color.BLUE)**.
- A **Rotate** transformation named **rotate** is created with an angle of 30 degrees and a pivot point at coordinates (75, 75). This means the rectangle will rotate by 30 degrees clockwise around the point (75, 75).
- The **rotate** transformation is added to the **rectangle** using **rectangle.getTransforms().add(rotate)**, indicating that the rotation

transformation should be applied to the rectangle.

When this JavaFX application is executed, it displays a window with a blue rectangle as shown in Figure 12.6. The rectangle will have undergone a rotation transformation, rotating by 30 degrees clockwise around the point (75, 75).

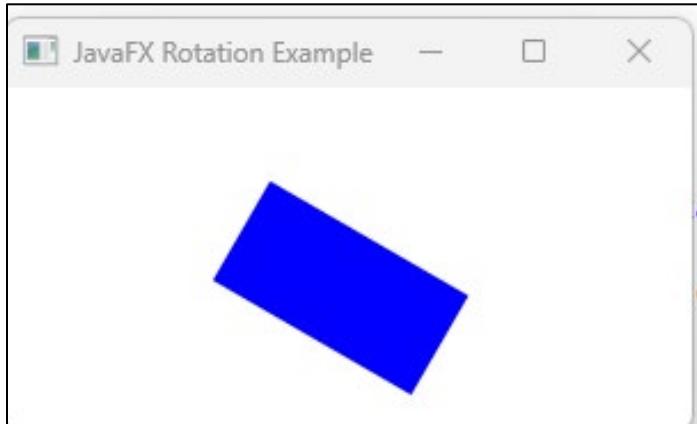


Figure 12.6: Output of JavaFX Rotation on Eclipse IDE

12.5 JavaFX Scaling

JavaFX Scaling is a transformation that allows you to resize graphical objects, such as shapes or nodes, along both the X and Y axes. Scaling is a fundamental transformation for creating animations, zooming effects, and altering the size of UI elements dynamically in JavaFX applications.

Key Points about JavaFX Scaling:

Scaling Factors

In JavaFX, scaling is defined by scaling factors for both the X and Y dimensions. You specify how much an object should be scaled along each axis. Scaling factors less than 1.0 reduce the size, while scaling factors greater than 1.0 increase the size.

Pivot Point

Scaling occurs around a pivot point, which is a fixed point around which the object scales. You can choose the pivot point's location to control the scaling effect.

Scale Class

To apply scaling to a graphical object, you use the **Scale** class. This class allows you to set scaling factors for both X and Y dimensions, specify the pivot point coordinates, and control other scaling properties.

Use Cases

Scaling is commonly used for zooming in and out of images, resizing UI components based on user interactions, and creating dynamic visual effects such as growing or shrinking animations.

Code Snippet 6 illustrates how to apply scaling to a JavaFX shape, such as a rectangle:

Code Snippet 6:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
```

```

import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.transform.Scale;

public class JavaFXScalingExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Scaling Example");

        Rectangle rectangle = new Rectangle(50, 50, 100, 50);
        rectangle.setFill(Color.BLUE);

        // Apply scaling
        Scale scale = new Scale(1.5, 1.5, 75, 75);
        // 1.5 times, pivot at (75, 75)
        rectangle.getTransforms().add(scale);

        StackPane root = new StackPane();
        root.getChildren().add(rectangle);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

- The code starts by importing the necessary JavaFX classes and defining the **JavaFXScalingExample** class, which extends Application.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX Scaling Example." The stage represents the main window of a JavaFX application.
- A Rectangle object named **rectangle** is created with an initial position at (50, 50) and dimensions of 100x50 pixels. This rectangle will serve as the graphical object that undergoes the scaling transformation.
- The rectangle is filled with a blue color using **rectangle.setFill(Color.BLUE)**.
- A Scale transformation named **scale** is created with scaling factors of 1.5 for both X and Y dimensions. This means that the rectangle will be scaled to 1.5 times its original size in both dimensions.
- The pivot point for scaling is set at coordinates (75, 75) using the **Scale** constructor. This is the point around which the scaling effect will occur.
- The **scale** transformation is added to the **rectangle** using **rectangle.getTransforms().add(scale)**, indicating that the scaling transformation should be applied to the rectangle.

When this JavaFX application is executed, it displays a window with a blue rectangle as shown in Figure 12.7. The rectangle has undergone a scaling transformation, enlarging it to 1.5 times its original size in both dimensions around the pivot point at (75, 75). In a practical scenario, this code can be modified to depict scaling based on a button click.

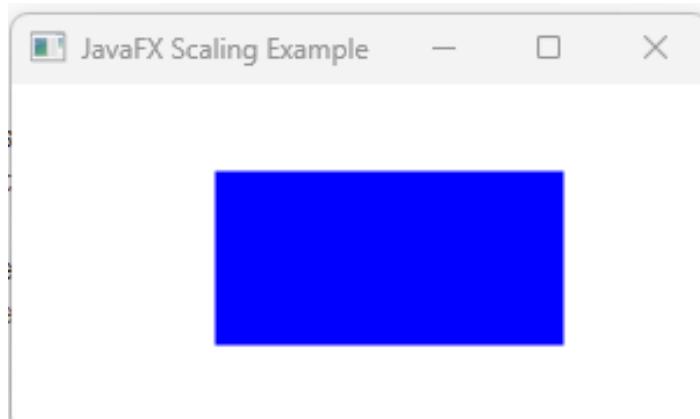


Figure 12.7: Output of JavaFX Scaling on Eclipse IDE

12.6 JavaFX Shearing

JavaFX Shearing is a transformation that distorts or skews graphical objects, such as shapes or nodes, by displacing their vertices along one axis while keeping the other axis fixed. Shearing is a powerful transformation for creating unique visual effects and simulating perspective in JavaFX applications.

Key Points about JavaFX Shearing:

Shear Factors

In JavaFX, shearing is defined by shear factors for both the X and Y axes. These factors determine the amount of shear along each axis. A shear factor of 0 means no shear, while non-zero values introduce the shear effect.

Pivot Point

Shearing occurs around a pivot point, which is a fixed point around which the object shears. Choose the pivot point's location to control the shearing effect.

Shear Class

To apply shearing to a graphical object, use the **Shear** class. This class allows to set shear factors for both X and Y dimensions, specify the pivot point coordinates, and control other shearing properties.

Use Cases

Shearing is commonly used for creating graphical distortions, simulating slanted or tilted perspectives, and achieving creative visual effects in animations.

Code Snippet 7 illustrates how to apply shearing to a JavaFX shape, such as a rectangle:

Code Snippet 7:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.transform.Shear;

public class JavaFXShearingExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Shearing Example");

        Rectangle rectangle = new Rectangle(50, 50, 100, 50);
        rectangle.setFill(Color.BLUE);

        // Apply shearing
        Shear shear = new Shear(0.3, 0.2);
        // Shear factors (0.3, 0.2)
        rectangle.getTransforms().add(shear);

        StackPane root = new StackPane();
        root.getChildren().add(rectangle);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

- The code starts by importing the necessary JavaFX classes and defining the **JavaFXShearingExample** class, which extends Application.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX Shearing Example." The stage represents the main window of a JavaFX application.
- A Rectangle object named **rectangle** is created with an initial position at (50, 50) and dimensions of 100x50 pixels. This rectangle will serve as a graphical object that undergoes the shearing transformation.
- The rectangle is filled with a blue color using **rectangle.setFill(Color.BLUE)**.
- A Shear transformation named **shear** is created with shear factors of (0.3, 0.2). This means that the rectangle will be sheared by 0.3 along the X-axis and 0.2 along the Y-axis.
- The **shear** transformation is added to the **rectangle** using

```
rectangle.getTransforms().add(shear), indicating that the shearing transformation should be applied to the rectangle.
```

When this JavaFX application is executed, it displays a window with a blue rectangle, as shown in Figure 12.8. The rectangle will have undergone a shearing transformation, distorting its shape based on the specified shear factors (0.3 along the X-axis and 0.2 along the Y-axis). In a practical scenario, this code can be modified to depict shearing based on a button click.

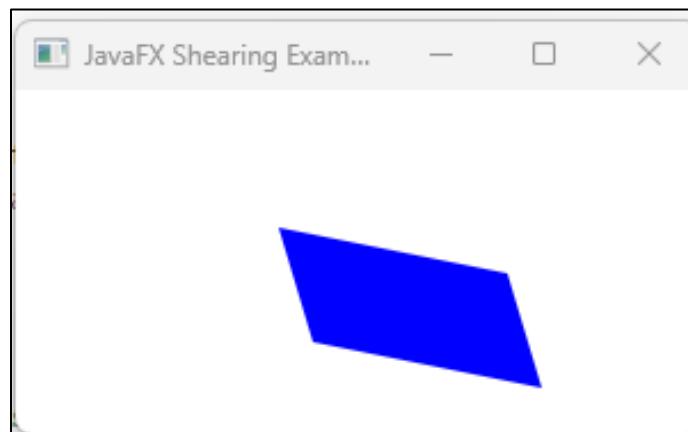
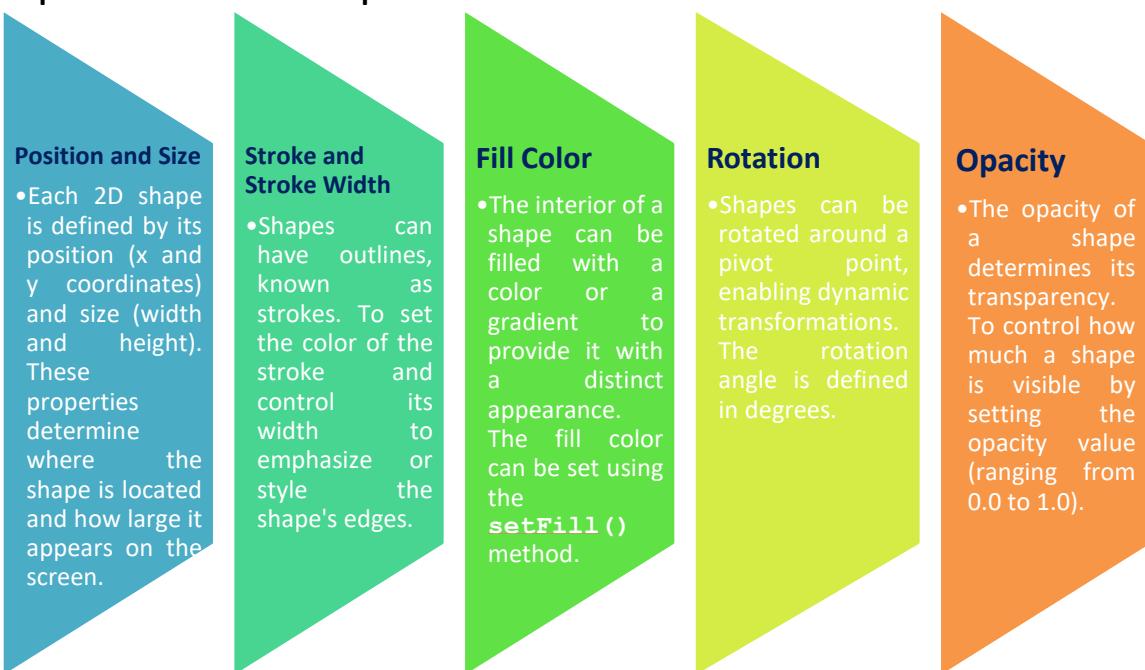


Figure 12.8: Output of JavaFX Shearing on Eclipse IDE

12.7 JavaFX 2D Shapes

In JavaFX, 2D shapes are essential components for creating GUIs, diagrams, charts, and various visual elements. JavaFX provides several built-in 2D shapes such as rectangles, circles, ellipses, lines, and polygons, which can be customized and styled to suit application's requirements.

Properties of JavaFX 2D Shapes:



Color in JavaFX 2D Shapes:

Colors play a crucial role in the visual design of 2D shapes in JavaFX. The `Color` class is used to define colors and it provides several ways to specify a color, including by name, RGB values, or Hue, Saturation, Brightness (HSB) values.

Code Snippet 8 illustrates creating and styling a Rectangle.

Code Snippet 8:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class JavaFXRectangleExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Rectangle Example");

        Rectangle rectangle = new Rectangle(50, 50, 100, 50);
        // x, y, width, height
        rectangle.setFill(Color.BLUE); // Set fill color
        rectangle.setStroke(Color.RED); // Set stroke color
        rectangle.setStrokeWidth(2.0); // Set stroke width

        StackPane root = new StackPane();
        root.getChildren().add(rectangle);

        Scene scene = new Scene(root, 300, 150);
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

- The code starts by importing the necessary JavaFX classes and defining the `JavaFXRectangleExample` class, which extends `Application`.
- In the `start` method, a new JavaFX stage (`primaryStage`) is created with the title "JavaFX Rectangle Example." The stage represents the main window of a JavaFX application.
- A `Rectangle` object named `rectangle` is created with specific dimensions and position. It is positioned at (x=50, y=50) and has a width of 100 pixels and a height of 50 pixels.

- `rectangle.setFill(Color.BLUE)` sets the fill color of the rectangle to blue, making the interior of the rectangle blue.
- `rectangle.setStroke(Color.RED)` sets the stroke color of the rectangle to red, creating a red outline (stroke) around the rectangle.
- `rectangle.setStrokeWidth(2.0)` sets the width of the stroke to 2.0 pixels, making the red outline thicker.
- The scene is set as the content of the primaryStage using `primaryStage.setScene(scene)`.
- Finally, `primaryStage.show()` is called to display the stage with the configured content.

When this JavaFX application is executed, it displays a window with a blue rectangle that has a red outline (stroke) around it as shown in Figure 12.9.

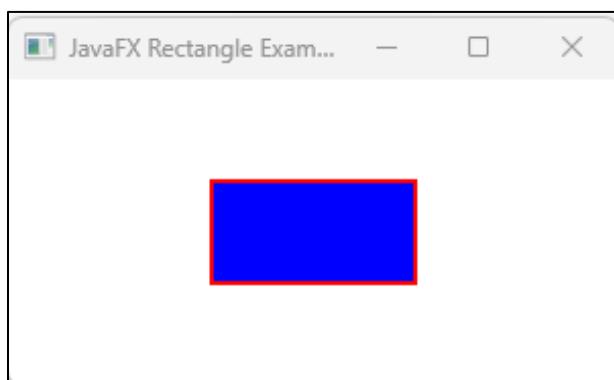
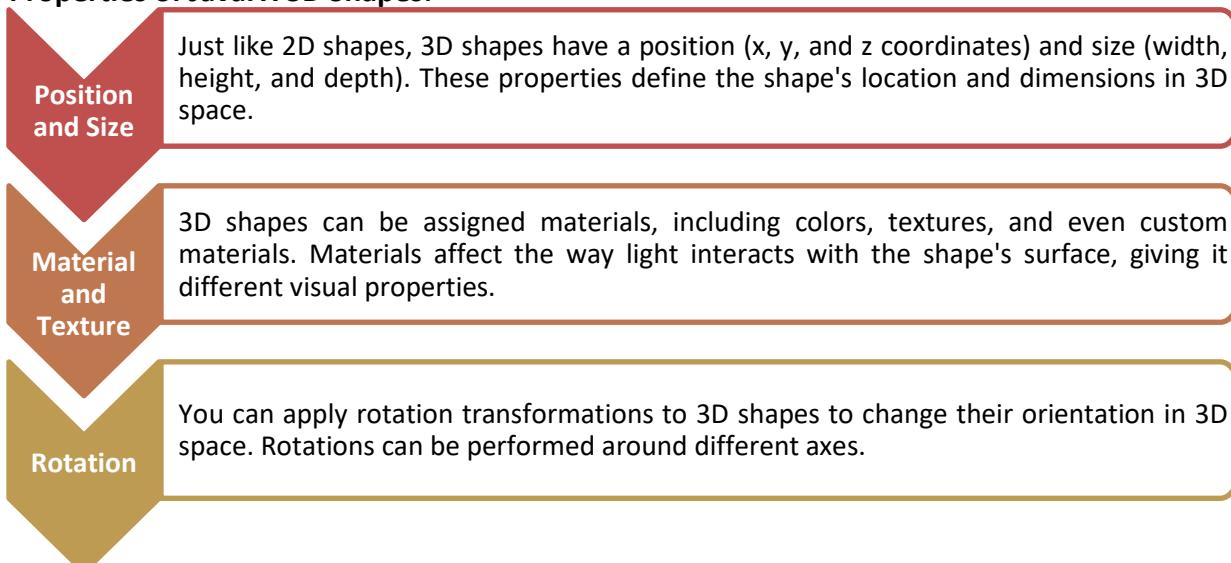


Figure 12.9: Output of JavaFX Rectangle on Eclipse IDE

12.8 JavaFX 3D Shapes

JavaFX provides a robust framework for working with 3D shapes, allowing developers to create and manipulate three-dimensional objects in your applications. These 3D shapes can be used to build immersive user interfaces, games, simulations, and more.

Properties of JavaFX 3D Shapes:



Scale

Scaling transformations can be used to resize 3D shapes along the X, Y, and Z axes independently, allowing you to create variations in size.

Opacity

Opacity controls the transparency of a 3D shape. You can specify how much light passes through the shape's surface.

Color in JavaFX 3D Shapes:

Color plays a significant role in the visual appearance of 3D shapes in JavaFX. Colors can be applied to the surface materials of 3D objects to achieve various visual effects.

Code Snippet 9 illustrates creating and coloring a 3D Box.

Code Snippet 9:

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.stage.Stage;

public class JavaFX3DBoxExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX 3D Box Example");

        // Create a 3D Box
        Box box = new Box(100, 100, 100); // Width, Height, Depth

        // Create a material with a specific color
        PhongMaterial material = new PhongMaterial();
        material.setDiffuseColor(Color.BLUE);

        // Apply the material to the box
        box.setMaterial(material);

        Group root = new Group();
        root.getChildren().add(box);

        Scene scene = new Scene(root, 400, 400, true);
        primaryStage.setScene(scene);
    }
}
```

```
        primaryStage.show();
    }
}
```

- The code starts by importing the necessary JavaFX classes and defining the **JavaFX3DBoxExample** class, which extends **Application**.
- In the **start** method, a new JavaFX stage (**primaryStage**) is created with the title "JavaFX 3D Box Example." The stage represents the main window of a JavaFX application.
- A **Box** object named **box** is created. This box represents a 3D cube with dimensions of 100x100x100 units (width, height, and depth). These dimensions define the size and shape of the 3D box.
- A **PhongMaterial** named **material** is created. **PhongMaterial** is a class that allows you to define the material properties of 3D shapes, including their color.
- **material.setDiffuseColor(Color.BLUE)** sets the diffuse color of the material to blue. The diffuse color determines how the shape reflects light and in this case, it is set to a solid blue color.
- The **material** created earlier is applied to the **box** using **box.setMaterial(material)**. This means that the 3D box will have blue material applied to its surface, giving it a blue color.
- A **Group** named **root** is created. In JavaFX, a **Group** is a container for holding other nodes (in this case, the 3D box). The 3D box is added as a child to the **root** using **root.getChildren().add(box)**.
- A 3D scene named **scene** is created with a size of 400x400 pixels. The **root** group is set as the content of this scene.
- The last argument, **true**, enables depth buffering. Depth buffering is essential for handling the depth of 3D objects and ensuring that objects in front occlude objects behind them correctly.
- Finally, **primaryStage.show()** is called to display the JavaFX stage, showing the 3D box with the specified blue color.

When this JavaFX application is executed, it displays a 3D box with a blue color. The color of the box is defined by the applied material as Figure 12.10.

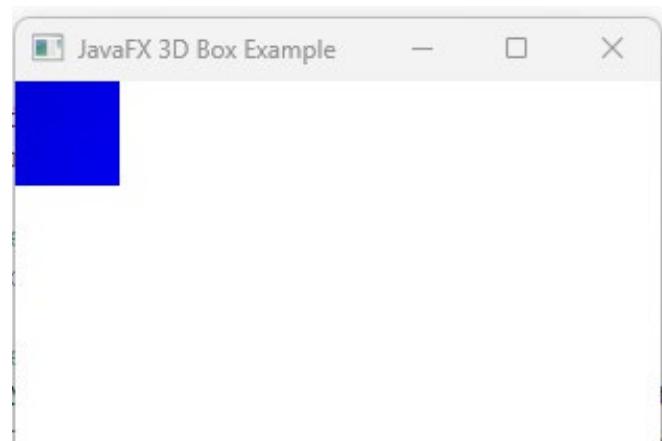


Figure 12.10: Output of JavaFX 2D Box on Eclipse IDE

12.9 Summary

- JavaFX Text allows to display text in JavaFX applications.
- Developers can set text properties such as font, size, color, and alignment to customize appearance of a text component in a GUI.
- JavaFX Transformation includes Translation, Rotation, Scaling, and Shearing, which allow to manipulate the position, orientation, size, and shape of graphical objects.
- JavaFX provides 2D shapes such as rectangles, circles, and polygons and 3D shapes for creating immersive graphics.
- Color in JavaFX is specified using the `Color` class and allows to define colors by name, RGB values, or HSB values, enabling rich color customization.
- Colors in JavaFX are crucial for shaping the visual appearance of objects. You can apply colors to fill, stroke, and material properties to achieve specific visual effects in both 2D and 3D graphics.

12.10 Check Your Progress

1. What does JavaFX text allow developers to do?

(A)	Create 3D shapes
(B)	Display text in JavaFX applications
(C)	Apply transformations to objects
(D)	Create colorful gradients

2. Which of the following is a property of JavaFX text?

(A)	Fill color
(B)	Translation
(C)	Rotation
(D)	Scaling factor

3. Which JavaFX transformation allows developers to change an object's position?

(A)	Scaling
(B)	Rotation
(C)	Translation
(D)	Shearing

4. Identify the property of a 2D shape that defines the outline of the shape.

(A)	Fill color
(B)	Width
(C)	Height
(D)	Stroke color

5. Which of these JavaFX classes is used to create 3D shapes?

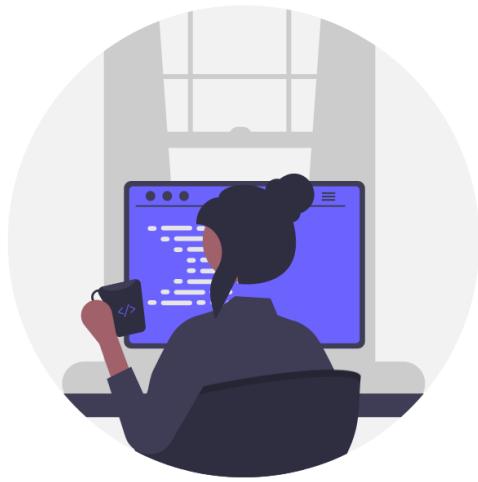
(A)	Shape3D
(B)	Box
(C)	Rectangle3D
(D)	Circle3D

12.10.1 Answers

1. B
2. A
3. C
4. D
5. B

Try It Yourself

1. Create a JavaFX program that allows users to input text and apply various text styles dynamically. Provide options for changing font, size, color, and alignment. Implement a preview area to show the styled text.
2. Develop an interactive JavaFX application that lets users manipulate 2D shapes using transformations. Allow them to translate, rotate, scale, and shear shapes using sliders or buttons.
3. In a 3D scene, create an object (For example a 3D cube) and implement advanced transformations such as rotation around specific axes (X, Y, and Z) and scaling along individual dimensions.
4. Build a JavaFX application that displays a gallery of various 2D shapes, including circles, rectangles, and polygons. Allow users to select a shape from the gallery and add it to the canvas.
5. Develop a 3D model viewer using JavaFX. Load and display 3D models (For example in OBJ or STL format) and allow users to rotate and zoom in/out to inspect the model from different angles.
6. Design a JavaFX application that generates color palettes. Users can input a base color, and the program should suggest complementary and analogous colors based on color theory principles.
7. Create an advanced color mixer that allows users to interactively adjust RGB and HSB values separately for precise color customization. Display the color in real-time as users adjust.



Session 13

JavaFX Layouts, UI, and Charts

Welcome to the Session, **JavaFX Layouts, UI, and Charts**.

In simplest terms, JavaFX is a powerful framework for building rich and interactive GUIs in Java applications. One of the fundamental aspects of designing a JavaFX GUI is understanding and effectively using layouts. Layouts in JavaFX help to arrange and position user interface components, such as buttons, labels, and text fields, in a structured and organized manner.

The session explains JavaFX Layouts, UI, and Charts in detail.

In this Session, you will learn to:

- Explain JavaFX Layouts
- Describe JavaFX CSS
- Describe JavaFX BorderPane
- Explain JavaFX HBox and VBox
- Describe JavaFX UI Controls
- Describe JavaFX UI Properties
- Explain JavaFX Charts and their types

13.1 JavaFX Layouts

Layouts are an essential component of GUI applications, including those developed using JavaFX. They provide a structured and organized way to arrange and position graphical elements (such as buttons, labels, text fields, and other UI controls) within a user interface.

Here are some key reasons why layouts are required and how they help in GUI applications in JavaFX:

- **Consistency and Readability:** Layouts help in maintaining a consistent and visually appealing design across different screen sizes and resolutions. They ensure that UI components are organized and positioned uniformly, making the interface more readable and user-friendly.

- **Adaptability to Different Screen Sizes:** In JavaFX applications, layouts allow you to create responsive designs that adapt to various screen sizes and orientations. Different devices have different screen sizes, and layouts enable your application to adjust its appearance accordingly.
- **Automatic Resizing:** Layouts can automatically resize and reposition UI components as the window or screen dimensions change. This makes your application more user-friendly and reduces the requirement for manual adjustments when users resize the window.
- **Ease of Maintenance:** Using layouts makes your code more maintainable and easier to understand. You can clearly define the relationships between UI components and their containers, making it simpler to modify and extend your UI in the future.
- **Accessibility:** Properly designed layouts improve accessibility for users with disabilities. They ensure that screen readers can interpret the UI structure correctly, making your application more inclusive.

JavaFX offers a variety of layout managers to help developers design user-friendly and responsive GUIs. Two of the most used layout managers are `HBox` and `VBox`.

JavaFX HBox and VBox Layouts: `HBox` and `VBox` are layout managers in JavaFX that help organize GUI components in a horizontal (`HBox`) or vertical (`VBox`) manner. These layouts are useful for arranging elements sequentially in a row or column, respectively.

➤ **VBox (Vertical Box):**

The `VBox` layout manager stacks its children in a vertical column, placing them on top of each other.

It is commonly used for creating forms, lists, or any layout where elements should be displayed vertically.

The `VBox` layout arranges its children in a vertical column, stacking them on top of each other.

Consider an analogy or real-world application example to understand this better.

Imagine you are designing a digital library application where users can browse and read books. Each book in your library is represented as a graphical element (a cover image) with additional information such as the title and author's name. Here is how the `VBox` layout could be useful in this scenario:

Bookshelf Layout Example:

- **Vertical Stacking:** The `VBox` layout allows to stack the book elements vertically, such as physical books on a bookshelf. Each book takes up a single slot in the column.
- **Consistency:** With a `VBox`, all book elements are aligned neatly in a single vertical column, creating a consistent and organized look. This consistency makes it easy for users to scan through the library.
- **Scalability:** As the library grows with more books, the `VBox` layout automatically accommodates new additions by stacking them below the existing ones. Users can scroll through the list of books, just like they would on a real bookshelf.

- **Responsive Design:** If the user resizes the application window or uses it on different devices, the `VBox` layout can adapt by rearranging the books to fit the available vertical space, ensuring that the user experience remains user-friendly and visually appealing.
- **Accessibility:** For users with screen readers or those navigating through keyboard controls, the `VBox` layout helps maintain a logical and linear reading order, making it easier for them to explore and select books.

In this analogy, the `VBox` layout serves as a fundamental building block for creating an efficient and user-friendly digital bookshelf, providing a visually pleasing and organized display of books, such as how books are arranged on physical bookshelves.

Code Snippet 1 illustrates the layout of `VBox`.

Code Snippet 1:

```
VBox vbox = new VBox();
vbox.getChildren().addAll(new Button("Button 1"), new
Button("Button 2"), new Button("Button 3"));
```

- In the code snippet, a `VBox` instance is created named `vbox`.
- Then, three buttons (Button 1, Button 2, and Button 3) are added to the `vbox` using the `getChildren() .addAll(...)` method.
- This code creates a vertical arrangement of these buttons, where each button is placed below the previous one.

➤ **HBox:**

The `HBox` layout can be useful in various user interface design scenarios where you must arrange multiple elements or components side by side horizontally.

The `HBox` layout manager arranges its children in a horizontal row, placing them side by side. This layout is well-suited for creating toolbars, navigation menus, or arranging elements from left to right.

Consider an analogy to understand this better.

Imagine you are designing a graphic design software application and you must create a toolbar for users to access various tools and features. This toolbar contains several buttons and icons for actions such as drawing shapes, selecting colors, and more. You want these toolbar elements to be organized horizontally.

Code Snippet 2 illustrates the layout of `HBox`.

Code Snippet 2:

```
HBox hbox = new HBox();
hbox.getChildren().addAll(new Label("Label 1"), new Label("Label
2"), new Label("Label 3"));
```

- In the code snippet, an `HBox` instance is created named `hbox`.
- Then, three labels (Label 1, Label 2, and Label 3) are added to the `hbox` using the

- getChildren().addAll(...). method.
- This code creates a horizontal arrangement of these labels, where each label is placed to the right of the previous one.

Code Snippet 3 illustrates a complete example of JavaFX HBox Layout.

Code Snippet 3:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class HBoxExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        // Create an HBox
        HBox hbox = new HBox();

        // Create buttons to add to the HBox
        Button button1 = new Button("Button 1");
        Button button2 = new Button("Button 2");
        Button button3 = new Button("Button 3");

        // Add buttons to the HBox
        hbox.getChildren().addAll(button1, button2, button3);

        // Create a scene and set it in the stage
        Scene scene = new Scene(hbox, 300, 200);
        primaryStage.setScene(scene);

        // Set the stage title and show it
        primaryStage.setTitle("HBox Example");
        primaryStage.show();
    }

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

Figure 13.1 depicts the output of Code Snippet 3.

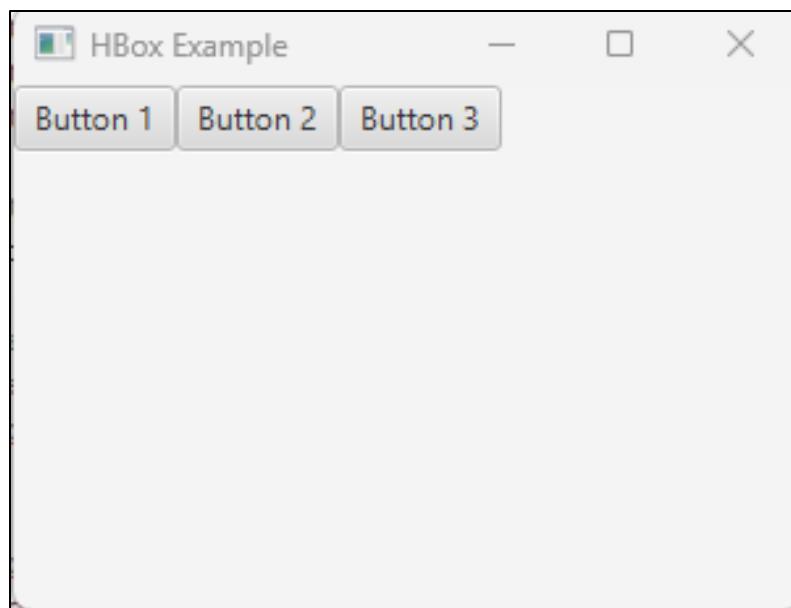


Figure 13.1: Example of JavaFX HBox Layout

Code Snippet 4 illustrates the example of JavaFX VBox layout.

Code Snippet 4:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class VBoxExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        // Create a VBox
        VBox vbox = new VBox(10);
        // 10-pixel spacing between elements

        // Create UI elements to add to the VBox
        Label nameLabel = new Label("Name:");
        TextField nameField = new TextField();
        Label ageLabel = new Label("Age:");
        TextField ageField = new TextField();

        // Add UI elements to the VBox
        vbox.getChildren().addAll(nameLabel, nameField, ageLabel,
ageField);
```

```

        // Create a scene and set it in the stage
        Scene scene = new Scene(vbox, 300, 200);
        primaryStage.setScene(scene);

        // Set the stage title and show it
        primaryStage.setTitle("VBox Example");
        primaryStage.show();
    }

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

```

Figure 13.2 depicts the output of Code Snippet 4.

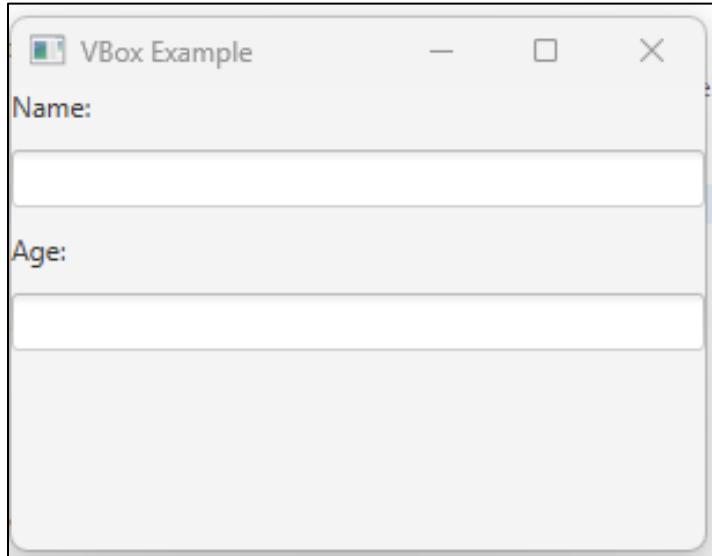


Figure 13.2: Example of JavaFX **VBox** layout

Explanation for Code Snippets 3 and 4 are given as follows:

- In the **HBox** example, a **hbox** and three **Button** elements are created. This results in horizontal arrangement of buttons.
- In the **VBox** example, a **Label** and **TextField** elements for name and age input are created. This results in vertical form-like layout.

➤ **BorderPane:**

The **BorderPane** layout manager is a powerful tool for organizing JavaFX GUI components into five distinct regions: top, bottom, left, right, and center. It is a useful layout manager in JavaFX for arranging content in a structured manner, especially when there is a requirement for a top-level container that separates content into distinct regions. Each region can contain various UI elements, allowing to create complex and structured layouts.

Following are the breakdown of the regions and their purposes:

Top:	Typically used for headers, titles, or menu bars.
Bottom:	Suitable for footers or status bars.
Left:	Ideal for navigation menus or sidebars.
Right:	Like the left region, useful for additional menus or sidebars.
Center:	The central content area where the main content of the application is placed.

For example: Creating a Simple email Client Application.

Let us assume that you are developing a simple email client application. In this application, you want to have a user interface with different regions for navigation, email content, and additional information. The `BorderPane` layout can help you organize this layout efficiently.

Code Snippet 5 illustrates an example of the `BorderPane` JavaFX Layout.

Code Snippet 5:

```
BorderPane borderPane = new BorderPane();
borderPane.setTop(new Label("Top"));
borderPane.setCenter(new Button("Center"));
borderPane.setLeft(new Button("Left"));
borderPane.setRight(new Button("Right"));
borderPane.setBottom(new Label("Bottom"));
```

- In the code snippet, a `BorderPane` is created named `borderPane`.
- To set different UI elements (a label in the top region, a button in the center, buttons on the left and right, and a label in the bottom) using the `setTop(...)`, `setCenter(...)`, `setLeft(...)`, `setRight(...)`, and `setBottom(...)` methods.
- This code creates a structured layout with UI elements in specific regions of the `BorderPane`.

Code Snippet 6 illustrates an example of the JavaFX BorderPane.

Code Snippet 6:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
```

```
public class BorderPaneExample extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        // Create a BorderPane  
        BorderPane borderPane = new BorderPane();  
  
        // Create UI elements for each region  
        Button topButton = new Button("Top");  
        Button bottomButton = new Button("Bottom");  
        Button leftButton = new Button("Left");  
        Button rightButton = new Button("Right");  
        Button centerButton = new Button("Center");  
  
        // Add UI elements to their respective regions  
        borderPane.setTop(topButton);  
        borderPane.setBottom(bottomButton);  
        borderPane.setLeft(leftButton);  
        borderPane.setRight(rightButton);  
        borderPane.setCenter(centerButton);  
  
        // Create a scene and set it in the stage  
        Scene scene = new Scene(borderPane, 400, 300);  
        primaryStage.setScene(scene);  
  
        // Set the stage title and show it  
        primaryStage.setTitle("BorderPane Example");  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

Figure 13.3 depicts the output of Code Snippet 6.

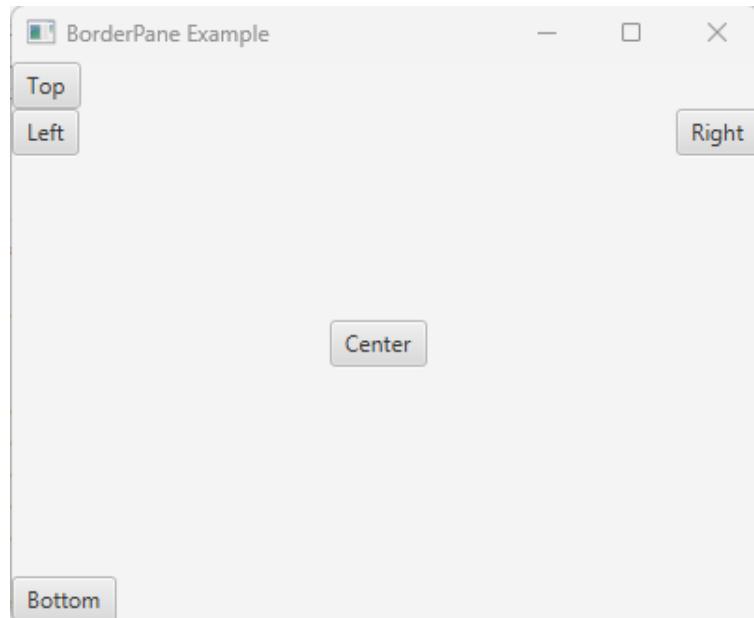


Figure 13.3: Example of the JavaFX BorderPane

Create a JavaFX application by extending the `Application` class and implementing the `start` method.

Inside the `start` method:

- Create a `BorderPane` named `borderPane` to serve as the main layout container.
- Create five `Button` elements, each representing a region (top, bottom, left, right, and center) of the `BorderPane`.
- Using the `setTop`, `setBottom`, `setLeft`, `setRight`, and `setCenter` methods, assign the buttons to their respective regions in the `BorderPane`.
- Create a `Scene` that contains the `borderPane` and set it in the `primaryStage`.
- Set the title of the stage and display it.

➤ **GridPane:**

The `GridPane` layout arranges children in a grid, making it suitable for creating complex layouts with rows and columns. For instance, when designing a calculator-like UI, one could use this type of layout. The `GridPane` layout is particularly useful when you must create complex layouts with rows and columns, such as designing a calculator-like user interface. In this scenario, you can use a `GridPane` to arrange the numeric keys and operators in a grid-like fashion.

Code Snippet 7 illustrates an example of the `GridPane` JavaFX Layout.

Code Snippet 7:

```
GridPane gridPane = new GridPane();
gridPane.add(new Button("Button 1"), 0, 0);
gridPane.add(new Button("Button 2"), 1, 0);
gridPane.add(new Button("Button 3"), 0, 1);
```

- In the code snippet, a `GridPane` is created named `gridPane`.

- Use the `add(...)` method to add three buttons to specific rows and columns within the `gridPane`.
- This code creates a grid-like layout where buttons are positioned in rows and columns based on their coordinates.

➤ **StackPane:**

The `StackPane` layout stacks children on top of each other. It is useful for layering UI elements. The `StackPane` layout is particularly useful when you want to layer UI elements on top of each other, creating a visually stacked effect. One common use case for `StackPane` is when you must display elements with a sense of depth or overlaying information.

Code Snippet 8 illustrates an example of the `StackPane` JavaFX Layout.

Code Snippet 8:

```
StackPane stackPane = new StackPane();
stackPane.getChildren().addAll(new Button("Button 1"), new
Button("Button 2"));
```

- In the code snippet, a `StackPane` is created named `stackPane`.
- Add two buttons to the `stackPane` using the `getChildren().addAll(...)` method.
- This code places the buttons on top of each other in the `StackPane`, creating a layered effect.

13.2 JavaFX CSS

JavaFX provides a robust styling mechanism that allows developers to enhance the visual appearance of their Java applications through Cascading Style Sheets (CSS). Using CSS with JavaFX, it is easy to define and apply styles to UI components, achieving a consistent and polished look. JavaFX CSS is an extension of standard CSS with some JavaFX-specific features.

Selectors:

JavaFX CSS uses selectors to target specific UI elements for styling. For example, to style all buttons in your application, you can use the selector: `Button`.

Properties:

Properties define the visual characteristics of UI components. Developers can set properties such as `background-color`, `font-size`, `text-fill`, and more.

Code Snippet 9 illustrates an example of JavaFX CSS UI components.

Code Snippet 9:

```
.my-button {
    -fx-background-color: #3498db;
    -fx-text-fill: white;
    -fx-font-size: 14px;
}
```

Pseudo-classes:

Pseudo-classes allow to apply styles based on component state or user interaction.

Common pseudo-classes include: `:hover`, `: focused`, `: disabled`, and more.

Code Snippet 10 illustrates an example of the JavaFX CSS Pseudo-classes.

Code Snippet 10:

```
Button:hover {  
    -fx-background-color: #2980b9;  
}
```

13.2.1 Applying CSS to JavaFX Components

There are two ways in which developers can apply CSS to JavaFX components. They are described as follows:

1. Inline CSS:

Inline CSS allows to apply styles directly to JavaFX components within the Java code. This method is useful when you want to apply specific styles to individual components and it is often used for quick and simple styling requirements. To apply inline CSS, you typically use the `setStyle` method on JavaFX nodes (components).

While inline CSS is a quick way to apply styles directly to components, it is not as maintainable as using external CSS, especially for larger applications. It can lead to code clutter and may not promote consistency across your application.

It is used to apply CSS styles directly to a JavaFX component in the Java code.

Code Snippet 11 illustrates an example of Inline CSS.

Code Snippet 11:

```
Button button = new Button("Click Me");  
button.setStyle("-fx-background-color: #3498db; -fx-text-fill:  
white;");
```

Explanation of the code:

- Start by creating a `Button` named `button` with the label "Click Me."
- To apply styles, use the `setStyle(...)` method on the `button` object.
- Inside the `setStyle(...)` method, provide a CSS string with the desired styling properties.
- Set the background color of the button to `#3498db`, the text color to white, and the font size to 14 pixels.
- Use the `setStyle(...)` method which allows developers to define styles for a specific component without the necessity for external CSS files.

2. External CSS File:

Applying CSS to JavaFX components using an external CSS file is a more organized and maintainable approach. In this method, you define your styles in a separate CSS file and then, apply those styles to JavaFX components using CSS classes or IDs.

Create an External CSS File: First, you are required to create a CSS file that contains the styles. You can use a simple text editor to create this file with a ".css" extension. For example, you might create a file named "styles.css."

Define Styles: In your CSS file, you can define styles for different JavaFX components. For example, you can style buttons, labels, and other UI elements.

Load the CSS File in Your JavaFX Application: In your JavaFX application, you must load the external CSS file and apply styles to the components using the CSS classes or IDs you defined in the CSS file.

Code Snippet 12 illustrates an example of the External CSS.

Code Snippet 12:

```
Scene scene = new Scene(root);
scene.getStylesheets().add("styles.css");
```

Explanation of the code:

- Create an instance of Scene (assuming root is the root node of application's scene).
- Then, use the getStylesheets().add(...) method on the scene object to specify the CSS file to be used.
- In this example, assume that there is an external CSS file named 'styles.css' with predefined styles. The JavaFX application will load and apply the styles defined in this file to the UI components within the specified scene.

Using external CSS files is a common practice for maintaining separation between code and styling.

Applying Styles by Selector:

Applying styles by selector in JavaFX allows you to target and style multiple components based on predefined criteria using CSS selectors. This method provides a flexible way to apply styles consistently across components that share a common trait. JavaFX CSS supports various selectors, including element selectors, class selectors, and ID selectors.

- **Create or Identify the Components:** First, you are required to create or identify the JavaFX components you want to style. These components should share some common characteristics that you can use as a selector criterion. For example, you may want to style all buttons or labels with a specific style.
- **Define the CSS Selector:** In your CSS file, define a CSS selector that matches the components you want to style. You can use one of following types of selectors:
 - **Element Selector:** To style all instances of a specific component type, use the component's name.
 - **Class Selector:** To style components with a specific CSS class applied to them, use the dot notation.
 - **ID Selector:** To style a single unique component identified by its ID, use the pound (#) notation.
- **Apply the Selector to Components:** In your JavaFX application code, apply the CSS selector to the relevant components using the `getStyleClass` or `setId` method, depending on

whether you are using class or ID selectors. For element selectors, no additional method is required.

Code Snippet 13 illustrates an example of CSS Selector.

Code Snippet 13:

```
Button button1 = new Button("Button 1");
Button button2 = new Button("Button 2");
button1.getStyleClass().add("my-button");
button2.getStyleClass().add("my-button");
```

Create two Button objects, button1 and button2, with labels 'Button 1' and 'Button 2.'

To apply a common style to both buttons, use the `getStyleClass().add(...)` method on each button object.

In this example, assume that there is a CSS class called `.my-button` defined in CSS file. The `getStyleClass().add(...)` method assigns this class to the buttons.

In CSS file, define the styles associated with the `.my-button` class, and those styles will be applied to both button1 and button2.

13.3 JavaFX UI Controls

JavaFX provides a rich set of User Interface (UI) controls that enable developers to create interactive and visually appealing GUIs for their Java applications. These UI controls range from basic components such as buttons and labels to advanced elements such as tables and charts.

Some examples are as follows:

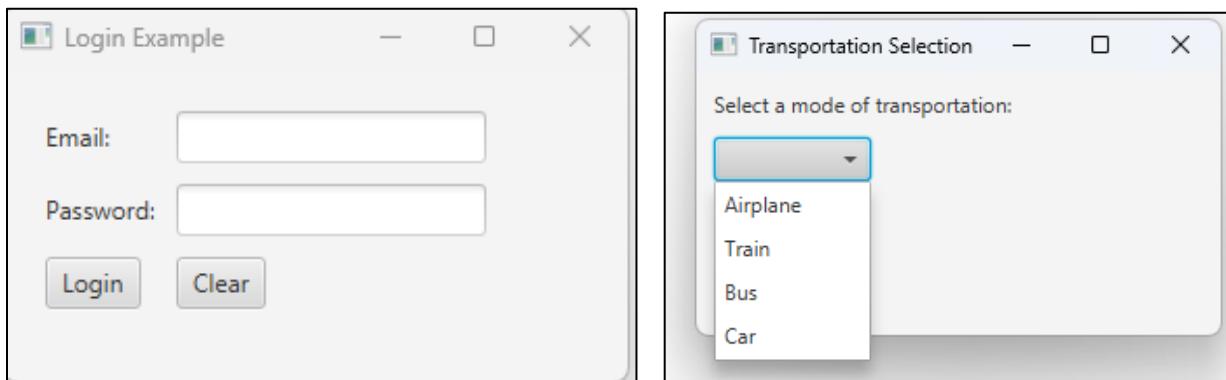


Table 13.1 shows commonly used JavaFX UI Controls.

Control	Description	Example
Button	The Button control represents a clickable button that can trigger actions.	Button button = new Button("Click Me");
Label	The Label class is used to create text-based labels that	Label label = new Label("Hello, JavaFX!");

Control	Description	Example
	can display information, instructions, or identifiers to users.	
TextField	A TextField is a single-line text input field that allows users to input or edit a single line of text.	TextField textField = new TextField();
TextArea	A TextArea is a multi-line text input field that allows users to input or edit multiple lines of text.	TextArea textArea = new TextArea();
CheckBox	The CheckBox control is a toggleable checkbox for binary choices.	CheckBox checkBox = new CheckBox("Enable Feature");
RadioButton	The RadioButton control is used for selecting a single option from a group of options.	RadioButton radioButton1 = new RadioButton("Option 1"); RadioButton radioButton2 = new RadioButton("Option 2");
ComboBox	The ComboBox control provides a dropdown list of options for selection.	ComboBox<String> comboBox = new ComboBox<>(); comboBox.getItems().addAll("Option 1", "Option 2", "Option 3");
ListView	The ListView control displays a list of items that users can select.	ListView<String> listView = new ListView<>(); listView.getItems().addAll("Item 1", "Item 2", "Item 3");

Table 13.1: Commonly Used JavaFX UI Controls

Code Snippet 14 illustrates creating and customizing a JavaFX Label control.

Code Snippet 14:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class LabelExample extends Application {

    @Override
    public void start(Stage primaryStage) {

```

```

// Create a Label with text
Label label = new Label("Hello, JavaFX!");

// Customize the Label's appearance
label.setStyle("-fx-font-size: 24px; -fx-font-weight: bold;
               -fx-text-fill: #3498db;");

// Create a layout to hold the Label
StackPane root = new StackPane();
root.getChildren().add(label);

// Create a Scene and set it in the Stage
Scene scene = new Scene(root, 300, 200);
primaryStage.setScene(scene);

// Set the Stage title and show it
primaryStage.setTitle("Label Example");
primaryStage.show();
}

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

```

Figure 13.4 depicts the output of Code Snippet 14.

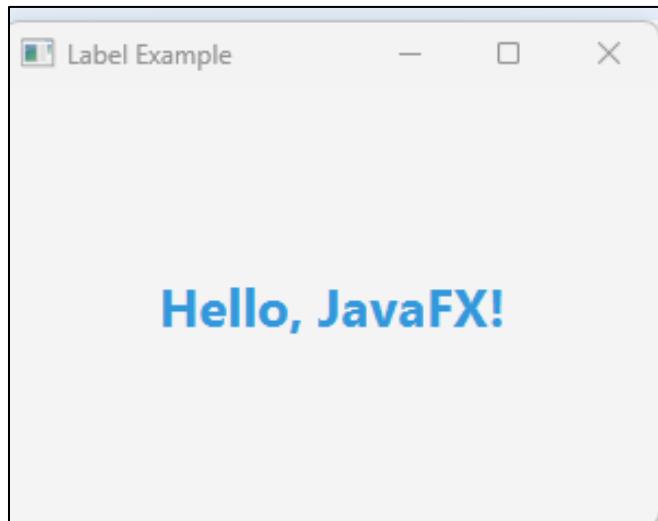


Figure 13.4: Creating and Customizing a JavaFX Label Control

Create a JavaFX application by extending the `Application` class and implementing the `start` method.

Inside the `start` method:

- Create a `Label` named `label` and set its text content to 'Hello, JavaFX!'.

- To customize the appearance of the label, use the `setStyle(...)` method to apply CSS styling. In this example, set the font size, font weight, and text color of the label.
- Create a `StackPane` layout named `root` to hold the label. `StackPane` is a simple layout manager that centers its content.
- Add the label to the `StackPane` using `root.getChildren().add(label)`.
- Create a `Scene` that contains the `StackPane` and set it on the `Stage`.

Table 13.2 shows difference between `TextField`, `TextArea`, and `Text`.

Control Type	Purpose	Input Type	Editable	Example Usage
<code>TextField</code>	Single-line text input	Single line	Yes	Usernames, search queries, short text
<code>TextArea</code>	Multi-line text input	Multiple lines	Yes	Paragraphs, comments, longer text
<code>Text</code>	Displaying static text	Static, non-editable	No	Labels, headings, displaying text content

Table 13.2: Difference between `TextField`, `TextArea`, and `Text`

Code Snippet 15 illustrates an example for JavaFX Button.

Code Snippet 15:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class ButtonExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me");
        button.setOnAction(event -> System.out.println("Button Clicked"));

        Scene scene = new Scene(button, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Button Example");
        primaryStage.show();
    }

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

Figure 13.5 depicts the output of Code Snippet 15.

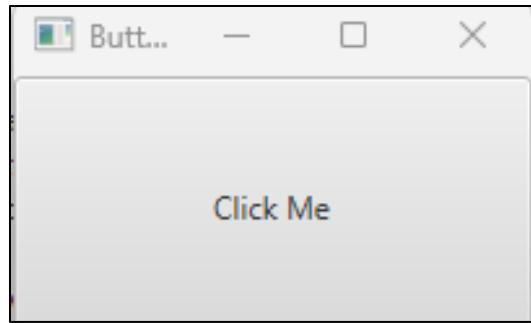


Figure 13.5: Button Example

This code creates a simple JavaFX application with a button. When the button is clicked, it prints a message to the console.

Code Snippet 16 illustrates ComboBox example.

Code Snippet 16:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ComboBoxExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        ComboBox<String> comboBox = new ComboBox<>();
        comboBox.getItems().addAll("Option 1", "Option 2", "Option
            3");
        comboBox.setOnAction(event -> {
            String selectedOption = comboBox.getValue();
            System.out.println("Selected option: " +
                selectedOption);
        });

        VBox vbox = new VBox(comboBox);
        Scene scene = new Scene(vbox, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("ComboBox Example");
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Code Snippet 16 creates a ComboBox that displays a list of options. When an option is selected from the drop-down, it prints the selected option to the console.

Figure 13.6 depicts the output of Code Snippet 16.

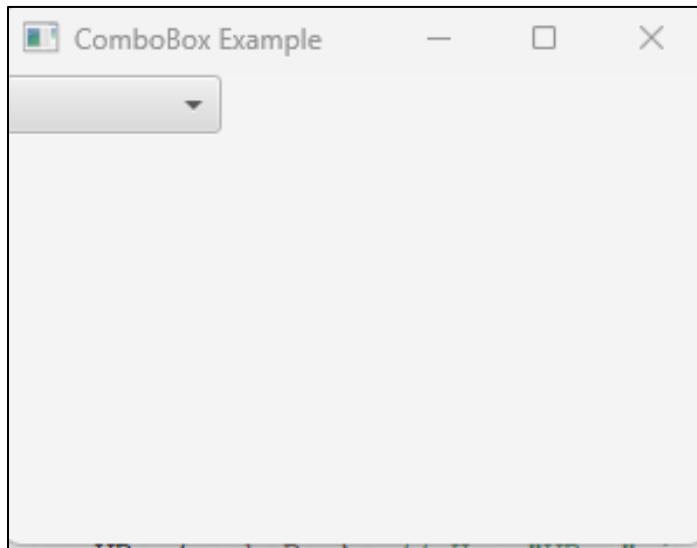


Figure 13.6: ComboBox Example

Code Snippet 17 illustrates an example of using several JavaFX UI controls together.

Code Snippet 17:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class GenericJavaFXApp extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        // Create UI controls
        Label label = new Label("Enter your name:");
        TextField textField = new TextField();
        Button button = new Button("Submit");
        CheckBox checkBox = new CheckBox("I agree to the terms and
            conditions");

        // Add event handling for the button
        button.setOnAction(e -> {
```

```

        String name = textField.getText();
        if (checkBox.isSelected()) {
            System.out.println("Hello, " + name + "! Agreement
                accepted.");
        } else {
            System.out.println("Hello, " + name + "! Agreement
                not accepted.");
        }
    });

    // Create a layout for the UI controls
    VBox vbox = new VBox(10); // Vertical layout with spacing
    vbox.getChildren().addAll(label, textField, checkBox,
        button);

    // Create a scene
    Scene scene = new Scene(vbox, 300, 200);

    // Set the scene in the stage
    primaryStage.setScene(scene);

    // Set the stage title and show it
    primaryStage.setTitle("Generic JavaFX GUI");
    primaryStage.show();
}
}

```

Figure 13.7 depicts the output of Code Snippet 17.

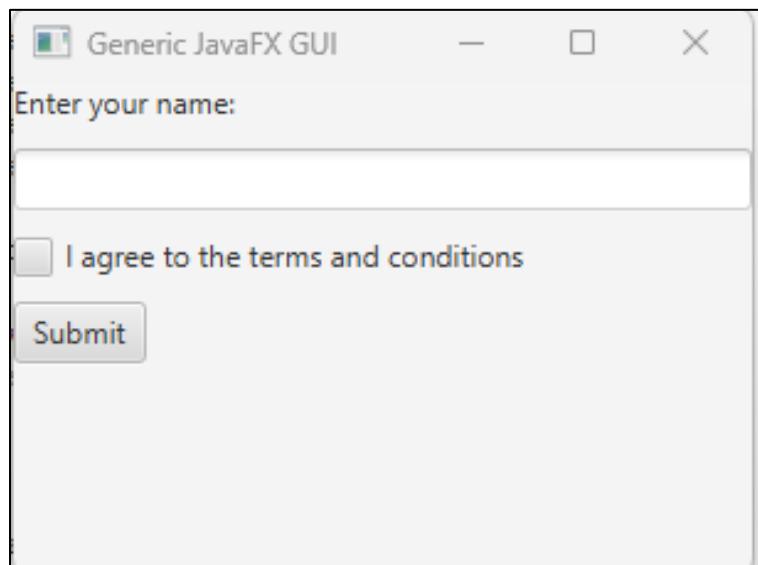


Figure 13.7: JavaFX GUI Controls

13.4 JavaFX UI Properties

One of the key features in JavaFX is the use of UI properties, which are observable and bindable attributes of JavaFX UI components. In simple words, JavaFX UI properties are attributes associated with UI components that can be observed and modified. UI properties enable developers to create dynamic and responsive user interfaces by reacting to changes in these properties. These properties allow developers to build responsive applications by listening for changes and updating the UI accordingly.

There are four commonly used properties of UI controls, which are described as follows:

Text Property:

The `textproperty` represents the text content of UI components such as labels, buttons, and text fields.

Code Snippet 18 illustrates how to use JavaFX `textproperty`.

Code Snippet 18:

```
Label label = new Label("Hello, JavaFX!");  
label.textProperty().addListener((observable, oldValue, newValue) -> {  
    System.out.println("Label text changed to: " + newValue);  
});
```

Value Property:

The `value` property is often associated with components such as sliders, progress bars, and spinner controls.

Code Snippet 19 illustrates how to use JavaFX `value` property.

Code Snippet 19:

```
Slider slider = new Slider(0, 100, 50);  
slider.valueProperty().addListener((observable, oldValue, newValue) -> {  
    System.out.println("Slider value changed to: " + newValue);  
});
```

Selected Property:

The `selected` property is used in check boxes, radio buttons, and toggle buttons to represent their checked/unchecked or selected/unselected state.

Code Snippet 20 illustrates how to use JavaFX `selected` property.

Code Snippet 20:

```
CheckBox checkBox = new CheckBox("Enable Feature");  
checkBox.selectedProperty().addListener((observable, oldValue, newValue) -> {
```

```
        System.out.println("CheckBox selected state changed to: " +  
newValue);  
});
```

Editable Property:

The `editable` property is associated with text input fields, indicating whether the field is editable or read-only.

Code Snippet 21 illustrates how to use JavaFX `editable` property.

Code Snippet 21:

```
TextField textField = new TextField("Editable Text");  
textField.editableProperty().addListener((observable, oldValue,  
newValue) -> {  
    System.out.printn("TextField editable state changed": " +  
newValue);  
});
```

Code Snippet 22 shows the code for a complete example showing various JavaFX UI properties.

Code Snippet 22:

```
import javafx.application.Application;  
import javafx.scene.Scene;  
import javafx.scene.control.Label;  
import javafx.scene.layout.VBox;  
import javafx.scene.control.Button;  
import javafx.stage.Stage;  
  
public class UIPropertiesExample extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        Label label = new Label("Initial Text");  
        Button button = new Button("Click Me");  
  
        // Add event handling for the button. When the button is  
        // clicked, the label's text should change  
  
        button.setOnAction(e -> {  
            // Change the text property  
            label.setText("New Text");  
        });  
  
        // Add a listener to the text property  
        label.textProperty().addListener((observable, oldValue,  
            newValue) -> {  
            button.setText("Button Clicked");  
            button.setDisable(true);  
        });  
    }  
}
```

```

// Create a layout for the UI controls
VBox vbox = new VBox(10); // Vertical layout with spacing
vbox.getChildren().addAll(label, button);
// Create a scene
Scene scene = new Scene(vbox, 300, 200);
primaryStage.setScene(scene);
primaryStage.setTitle("UI Properties Example");
primaryStage.show();
}

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

```

Figure 13.8 depicts the output of Code Snippet 22 before and after button click. A Label is displayed initially with text "Initial Text". When the button is clicked, the label text is changed and at the same time, the button is disabled so that it cannot be clicked again. This is done through use of the `setDisable()` method.

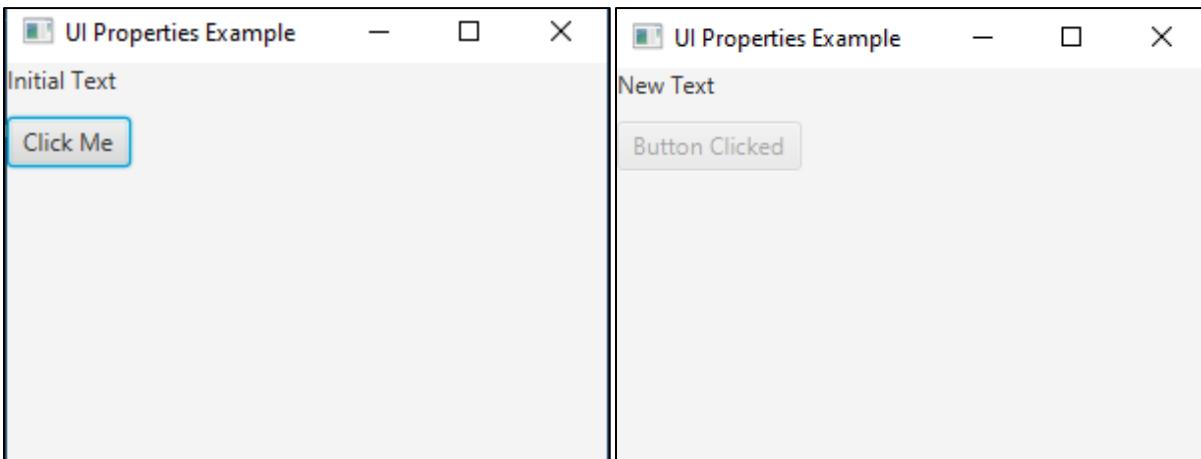


Figure 13.8: Example Showing JavaFX UI Properties in Action

13.5 JavaFX Charts and their Types

JavaFX provides a robust charting library that allows developers to create a wide variety of charts and graphs to visualize data in their Java applications. These charts are highly customizable and can be used for data analysis, reporting, and presenting information in a visually appealing manner.

Figures 13.9 to 13.11 depict various types of charts.

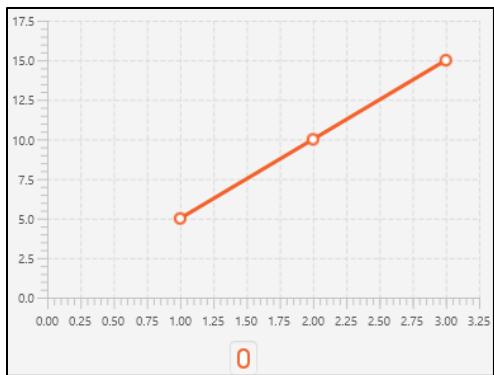


Figure 13.9: Line Chart

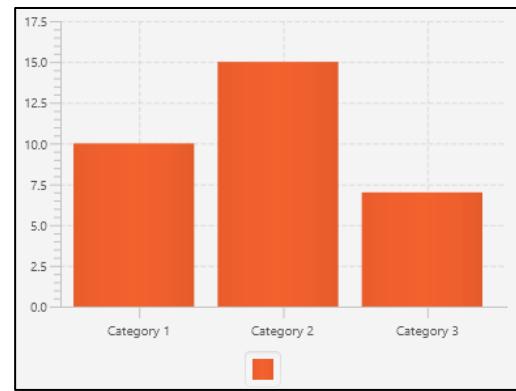


Figure 13.10: Bar Chart

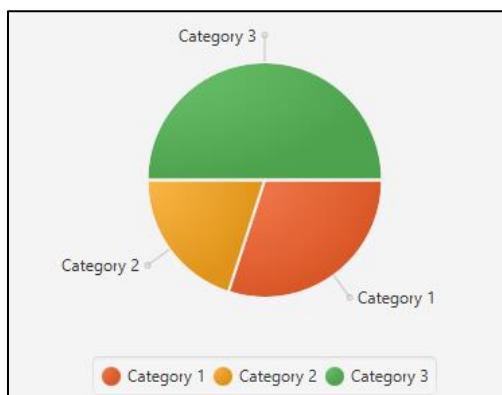


Figure 13.11: Pie Chart

JavaFX offers several types of charts, each designed for different data visualization requirements. Some of the most used JavaFX chart types are as follows:

Line Chart:

Creating a line chart in JavaFX allows you to visualize data with a series of data points connected by lines. Line charts are useful for showing trends and changes in data over time. To create a line chart in JavaFX, you can use the `LineChart` class from the JavaFX library.

A line chart is used to display data points connected by lines. It is ideal for visualizing trends and data that change over time.

Code Snippet 23 illustrates creating a line chart.

Code Snippet 23:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.stage.Stage;
```

```

public class LineChartExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        NumberAxis xAxis = new NumberAxis();
        NumberAxis yAxis = new NumberAxis();

        LineChart<Number, Number> lineChart = new
                LineChart<>(xAxis, yAxis);

        XYChart.Series<Number, Number> series = new
                XYChart.Series<>();
        series.getData().addAll(new XYChart.Data<>(1, 5),
                               new XYChart.Data<>(2, 10),
                               new XYChart.Data<>(3, 15));

        lineChart.getData().add(series);

        Scene scene = new Scene(lineChart, 400, 300);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Line Chart Example");
        primaryStage.show();
    }

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

```

Figure 13.12 depicts the output for Code Snippet 23.

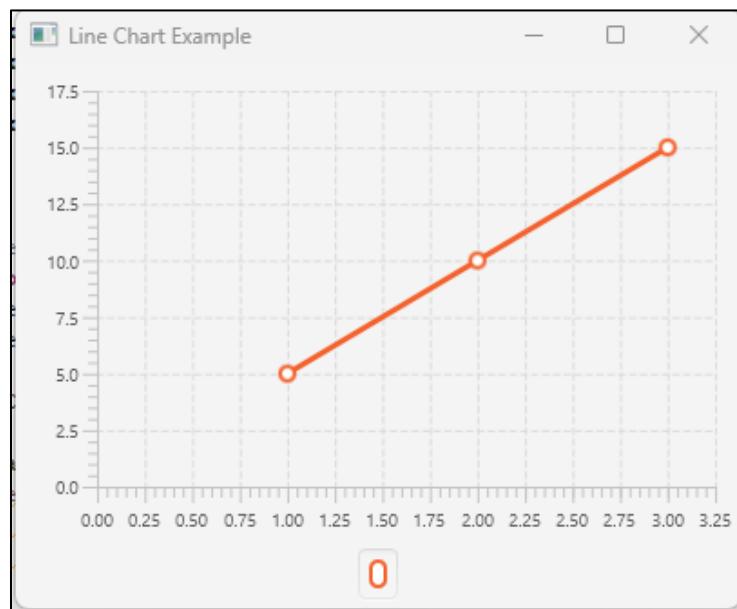


Figure 13.12: Creating a Line Chart

Bar Chart:

Creating a bar chart in JavaFX is a powerful way to visualize data as a series of bars, where the length or height of each bar represents a data value.

A bar chart displays data as horizontal or vertical bars. It is commonly used for comparing data between different categories or groups.

Code Snippet 24 illustrates creating a vertical bar chart.

Code Snippet 24:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.stage.Stage;

public class BarChartExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        CategoryAxis xAxis = new CategoryAxis();
        NumberAxis yAxis = new NumberAxis();

        BarChart<String, Number> barChart = new BarChart<>(xAxis,
                yAxis);

        XYChart.Series<String, Number> series = new
                XYChart.Series<>();
        series.getData().addAll(new XYChart.Data<>("Category 1",
                10), new XYChart.Data<>("Category 2", 15),
                new XYChart.Data<>("Category 3", 7));

        barChart.getData().add(series);

        Scene scene = new Scene(barChart, 400, 300);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Bar Chart Example");
        primaryStage.show();
    }

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

Figure 13.13 depicts the output of Code Snippet 24.

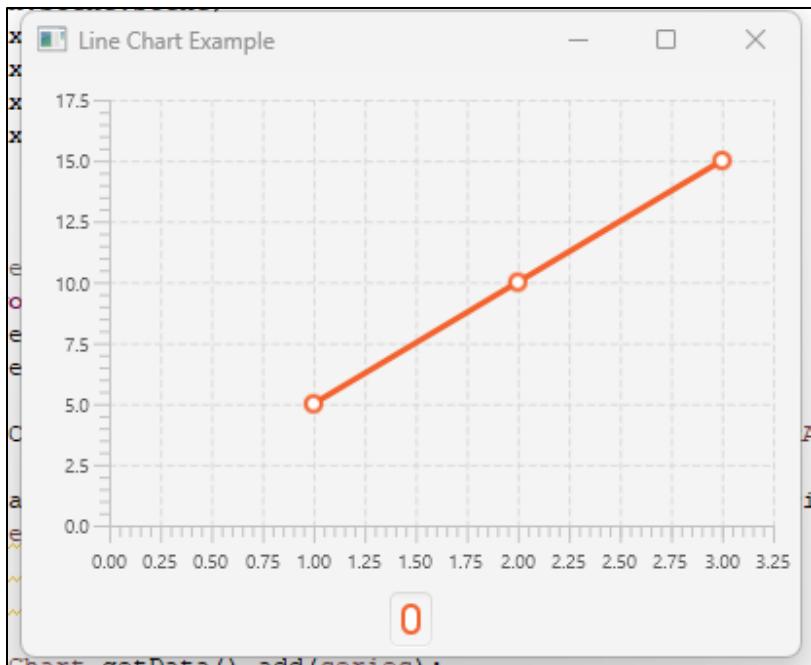


Figure 13.13: Creating a Vertical Bar Chart

Pie Chart:

Creating a pie chart in JavaFX is a great way to visualize data as a circular chart with segments (slices) where each segment represents a data category.

A pie chart is used to display data as a circular chart divided into slices. It is effective for showing the proportion of different parts to a whole.

Code Snippet 25 illustrates creating a pie chart.

Code Snippet 25:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.stage.Stage;

public class PieChartExample extends Application {

    @Override
    public void start(Stage primaryStage) {
        PieChart pieChart = new PieChart();

        PieChart.Data slice1 = new PieChart.Data("Category 1", 30);
        PieChart.Data slice2 = new PieChart.Data("Category 2", 20);
        PieChart.Data slice3 = new PieChart.Data("Category 3", 50);

        pieChart.getData().addAll(slice1, slice2, slice3);

        Scene scene = new Scene(pieChart, 400, 300);
        primaryStage.setScene(scene);
    }
}
```

```
        primaryStage.setTitle("Pie Chart Example");
        primaryStage.show();
    }

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

Figure 13.14 shows a pie chart generated using code given in Code Snippet 25.

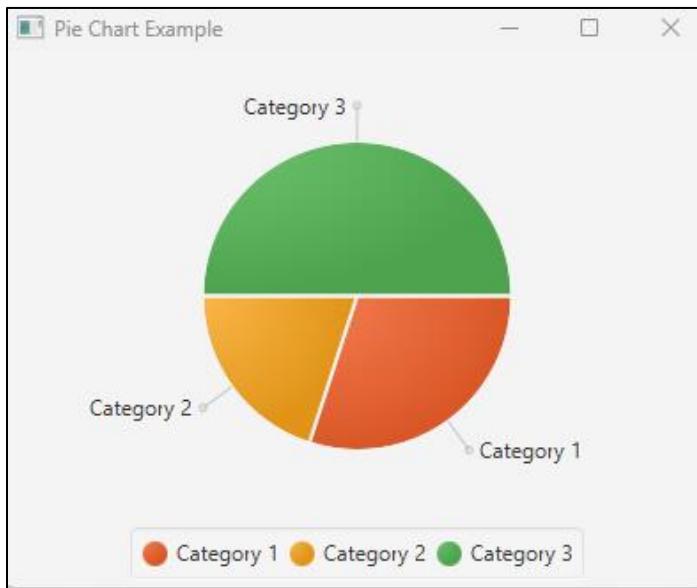


Figure 13.14: Creating a Pie Chart

13.6 Summary

- JavaFX offers various layout managers, including `BorderPane`, `HBox`, and `VBox`, to structure user interfaces.
- JavaFX allows styling of UI components using CSS, promoting separation of design and logic.
- JavaFX Labels display non-editable text or information in a graphical user interface.
- JavaFX UI properties are observable and bindable attributes of UI components.
- JavaFX offers a versatile charting library for visualizing data in various forms.
- Common chart types include line charts (for trends), bar charts (for comparisons), and pie charts (for proportions).

13.7 Check Your Progress

1. Which JavaFX layout manager is suitable for organizing components in five regions: top, bottom, left, right, and center?

(A)	HBox
(B)	VBox
(C)	GridPane
(D)	BorderPane

2. What is the primary purpose of using CSS in JavaFX applications?

(A)	Handling user input
(B)	Defining data models
(C)	Styling user interface components
(D)	Performing mathematical calculations

3. What is the primary purpose of a JavaFX Label component?

(A)	Collecting user input
(B)	Displaying text or non-editable information
(C)	Managing layouts and alignment
(D)	Creating interactive charts

4. What are JavaFX UI properties?

(A)	Visual effects applied to UI components
(B)	Observable and bindable attributes of UI components
(C)	External resources used in UI design
(D)	User interface layout managers

5. When is a line chart in JavaFX typically used?

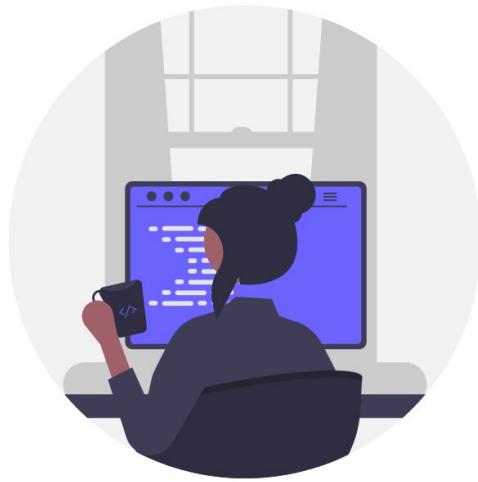
(A)	To display data as horizontal bars
(B)	To show proportions within a whole
(C)	To visualize trends and changes over time
(D)	To create pie-shaped data representations

13.7.1 Answers

1. D
2. C
3. B
4. B
5. C

Try It Yourself

1. Create a JavaFX application that uses a `BorderPane` layout to arrange UI components in following regions: top (for a title), left (for navigation links), center (for content), and bottom (for footer information).
2. Develop a JavaFX program that uses both `HBox` and `VBox` layouts to create a simple form with labels and text fields. Use `HBox` for arranging labels and `VBox` for organizing the label-text field pairs vertically.
3. Create a JavaFX application with multiple UI components (For Example buttons, labels, and text fields) and apply CSS styles to customize their appearance. Experiment with different font sizes, colors, and background styles.
4. Modify the previous example to load CSS styles from an external CSS file. Apply different styles to various UI components using class selectors.
5. Develop a JavaFX program with a label that displays a greeting message. Customize the label's appearance by setting its font size, color, and style.
6. Create a JavaFX application that includes a label and a button. When the button is clicked, change the label's text to display a new message.
7. Create a JavaFX program that includes a slider and a label. Bind the label's text property to the slider's value property so that the label dynamically displays the slider's current value.
8. Build a JavaFX application with a checkbox and a label. When the check box is selected (checked), change the label's text to indicate the selection state.
9. Develop a JavaFX application that displays a line chart representing monthly sales data for a business. Use random data for demonstration purposes.
10. Create a JavaFX program that shows a vertical bar chart illustrating the sales figures of different products. Customize the chart by adding labels and different colors for each bar.



Session 14

JavaFX Event Handling

Welcome to the Session, **JavaFX Event Handling**.

In simplest terms, JavaFX provides a robust event handling framework that allows developers to create interactive and responsive user interfaces. Events can be triggered by various user interactions, such as mouse clicks, keyboard input, or changes in UI components. In this session, we will explore JavaFX event handling, including convenience methods, event filters, and event handlers, and provide code snippets to demonstrate these concepts. The session explains JavaFX Event Handling in detail.

In this Session, you will learn to:

- Explain JavaFX Event Handling
- Identify JavaFX Convenience methods
- List and describe JavaFX Event Filters
- Elaborate on JavaFX Event Handlers

14.1 JavaFX Event Handling

JavaFX provides a robust and flexible event handling mechanism to manage user interactions, such as mouse clicks, keyboard input, and other user-triggered actions within a GUI application. Event handling is crucial in creating responsive and interactive JavaFX applications.

1. Event and EventHandler:

- An event in JavaFX represents a specific user interaction or system action, such as a mouse click or a button press.
- An **EventHandler** is a functional interface used to handle events. It defines a single method, `handle()`, which is invoked when the event occurs.

2. Event Source:

- The component or node that generates an event is referred to as the event source.
- Common event sources include buttons, text fields, and graphical elements.

14.2 JavaFX Convenience Methods

JavaFX provides convenient methods to simplify event handling. These methods are typically used with components such as buttons, allowing you to specify an action to be performed when the button is clicked.

Code Snippet 1 shows the use of JavaFX Convenience Methods.

Code Snippet 1:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFXConvenienceMethodsExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Convenience Methods
Example");

        Button clickButton = new Button("Click Me");

        // Before click (Using convenience method)
        clickButton.setOnAction(event -> {
            clickButton.setText("Clicked!");
        });

        StackPane root = new StackPane();
        root.getChildren().add(clickButton);
        primaryStage.setScene(new Scene(root, 300, 200));
        primaryStage.show();
    }

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

Figures 14.1 and 14.2 show the output of Code Snippet 1 before and after Click Me button click event respectively.

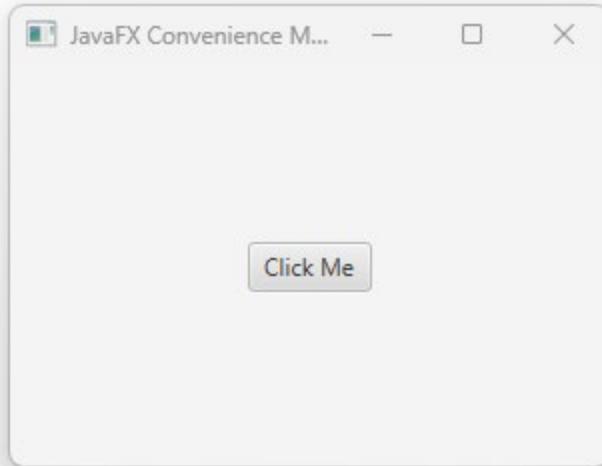


Figure 14.1: JavaFX Convenience Methods (before button click)

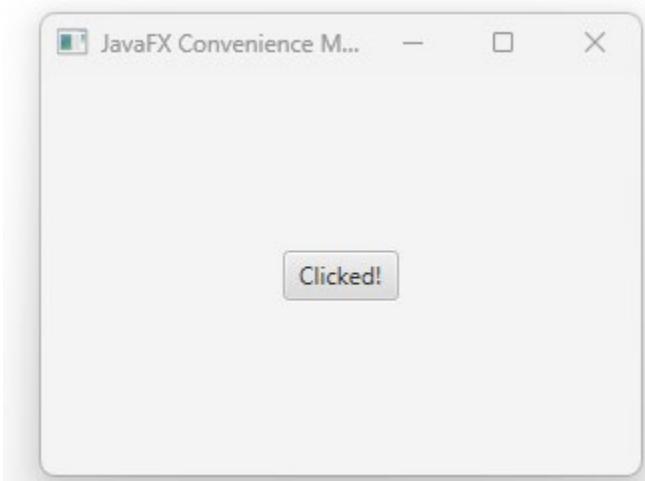


Figure 14.2: JavaFX Convenience Methods (after button click)

The explanation of Code Snippet 1 is as follows:

1. Import Statements:

- The code begins with importing necessary JavaFX classes and components, including Application, Scene, Button, StackPane, and Stage.

2. Extending Application:

- The EventHandlingExample class extends the Application class, which is a fundamental starting point for JavaFX applications.

3. Overriding the start Method:

- The start method is the entry point for JavaFX applications. It is overridden to define the initial UI setup and behavior.

4. Creating a Button:

- Inside the start method, a Button named button is created with the label Click Me. This button represents the interactive UI element in the application.

5. Handling Button Clicks Using a Convenience Method:

- JavaFX provides a convenient method named setOnAction for handling button clicks and other action events easily.

- In Code Snippet 1, the button's `setOnAction` method is called with a lambda expression as its argument. The lambda expression defines what should happen when the button is clicked.
- The lambda expression (`event -> { ... }`) serves as an event handler. When the button is clicked, the code inside the lambda expression is executed.
- In this case, it simply prints the message "Clicked!" to the console.

6. Creating a Root Node (StackPane):

- A StackPane named `root` is created. StackPane is a layout container in JavaFX that allows stacking UI elements on top of each other.

7. Adding the Button to the Root Node:

- The button is added to the `root` using the `getChildren().add(button)` method. This action places the button at the center of the StackPane.

8. Creating a Scene:

- A Scene is created, specifying the `root` as its content, and setting the dimensions to 300 pixels in width and 200 pixels in height. The Scene represents the visual content of the application's window.

9. Setting the Scene and Stage:

- The scene is set as the content of the `primaryStage`, which represents the main window of the application.
- The title of the `primaryStage` is set to "Event Handling Example."

10. Launching the Application:

- Finally, the `main` method calls `launch(args)` to start the JavaFX application.

11. Event Handling Output:

- On running this JavaFX application, it displays a window with a button labeled Click Me.
- Clicking the button triggers the event handler attached to it, which prints "Clicked!" to the console.

14.3 JavaFX Event Filters

Event filters allow intercepting and handling events during the event capturing phase before they reach the event target. In JavaFX, event filters are a mechanism for handling events in a specific order before they reach their target node. Event filters can be useful for tasks such as event logging, event manipulation, or event interception.

Code Snippet 2 illustrates JavaFX Event Filters.

Code Snippet 2:

```
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFXEventFiltersExample extends Application {
```

```

public static void main(String[] args) {
    launch(args);
}
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("JavaFX Event Filters Example");

    Button clickButton = new Button("Click Me");

    // Add an event filter before the click
    clickButton.addEventFilter(MouseEvent.MOUSE_CLICKED, new
EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
            System.out.println("Event filter before click");
        }
    });
    // Add an event handler after the click
    clickButton.setOnMouseClicked(event -> {
        System.out.println("Event handler after click");
    });

    StackPane root = new StackPane();
    root.getChildren().add(clickButton);
    primaryStage.setScene(new Scene(root, 300, 200));
    primaryStage.show();
}
}

```

Figures 14.3 and 14.4 show the output of Code Snippet 2 before and after button click respectively.

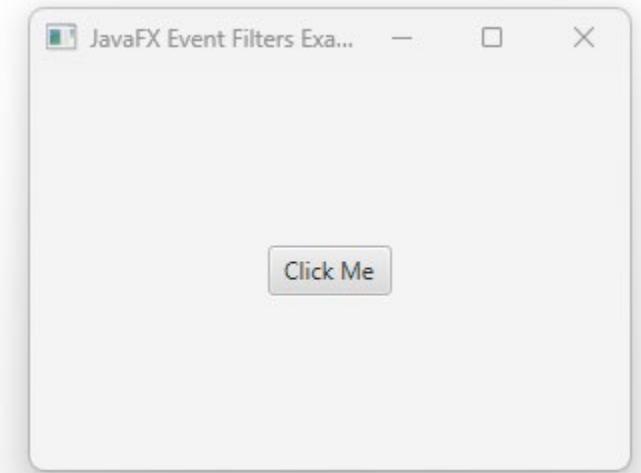


Figure 14.3: JavaFX Event Filters (before button click)

```

Event filter before click
Event handler after click

```

Figure 14.4: Console Window 5or JavaFX Event Filters (after button click)

The explanation of Code Snippet 2 is as follows:

1. Adding an Event Filter:

- Event filters in JavaFX are used to intercept and handle events during the event capturing phase before they reach the event target.
- In this code snippet, an event filter is added to the button using the `addEventFilter` method.
- The first argument to `addEventFilter` specifies the type of event to capture. In this case, it captures mouse click events.
(`javafx.scene.input.MouseEvent.MOUSE_CLICKED`).
- The second argument is a lambda expression that defines the action to be taken when the event filter captures the specified event. In this case, it prints "Event filter before click Event handler after click" to the console.

14.4 JavaFX Event Handlers

Event handlers are used to handle events after they have reached their target node. You can attach event handlers directly to specific UI components to respond to events generated by those components. JavaFX event handlers are used to respond to user interactions and other events in JavaFX applications.

Code Snippet 3 illustrates JavaFX Event Handlers.

Code Snippet 3:

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFXEventHandlersExample extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Event Handlers Example");

        Button clickButton = new Button("Click Me");

        // Event handler before the click
        clickButton.addEventHandler(ActionEvent.ACTION, new
EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Event handler before click");
            }
        });
    }
}
```

```

        }
    });
    // Event handler after the click
    clickButton.setOnAction(event -> {
        System.out.println("Event handler after click");
    });
    StackPane root = new StackPane();
    root.getChildren().add(clickButton);
    primaryStage.setScene(new Scene(root, 300, 200));
    primaryStage.show();
}
}

```

Figures 14.5 and 14.6 show the output of Code Snippet 3 before and after the button is clicked respectively.

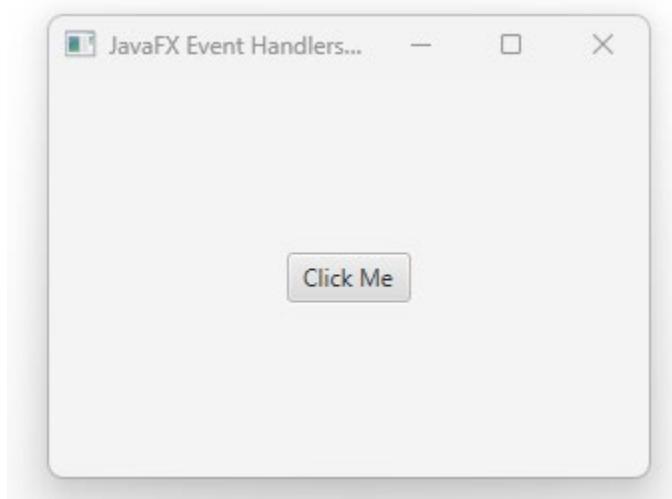


Figure 14.5: JavaFX Event Handlers

Event handler before click
 Event handler after click

Figure 14.6: Console Window for JavaFX Event Handlers (after button click)

The explanation of Code Snippet 3 is as follows:

1. Class Definition:

- The class is named `EventHandlersExample` and extends the `Application` class. In JavaFX, an application starts by defining a class that extends `Application`.

2. Event Handling Using `setOnMouseClicked`:

- The `button` object has an event handler attached to it using the `setOnMouseClicked` method. This event handler is a lambda expression that defines what should happen when the button is clicked.
- When the button is clicked, it prints the message "Event filter before click Event handler after click" to the console. This demonstrates how you can use event handling to respond to user interactions.

14.5 Summary

- JavaFX events are used for user interactions and system actions and are handled using the `EventHandler` interface's `handle()` method.
- The component that triggers an event is the event source, including elements such as buttons and text fields.
- JavaFX simplifies event handling with methods such as `setOnAction` to specify actions when elements like buttons are clicked.
- Event filters intercept and handle events during the capturing phase before they reach the target, often used with parent nodes.
- Event handlers respond to events after they reach the target node and can be attached to specific UI components, such as using `setOnMouseClicked` for button clicks.

14.6 Check Your Progress

1. What is an EventHandler in JavaFX used for?

(A)	To create UI components
(B)	To define event sources
(C)	To handle events when they occur
(D)	To specify event filters

2. Which method in JavaFX is typically used for handling button clicks and other action events?

(A)	setOnClick()
(B)	addEventFilter()
(C)	setOnAction()
(D)	setOnEventHandler()

3. When are event filters in JavaFX applied?

(A)	After events have reached their target node
(B)	During event capturing, before events reach their target node
(C)	Only when there is a mouse event
(D)	When the event source is a text field

4. Event handlers in JavaFX are used to:

(A)	Intercept events during the event capturing phase
(B)	Create event sources
(C)	Define convenience methods for event handling
(D)	Handle events after they have reached their target node

5. What is the purpose of the `setOnMouseClicked` method in JavaFX?

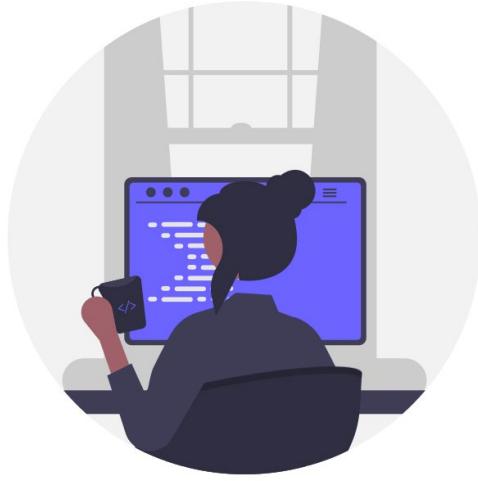
(A)	To set the text of a button
(B)	To create a new scene
(C)	To specify an event handler for mouse clicks
(D)	To add a filter for mouse events

14.6.1 Answers

1. C
2. C
3. B
4. D
5. C

Try It Yourself

1. Create a JavaFX application with a button that, when clicked, displays an alert saying "Hello, JavaFX!"
2. Build a JavaFX program that detects and displays the coordinates of the mouse when the user clicks on a canvas. Implement event handlers for onMouseClicked and onMouseMoved events.
3. Develop a simple calculator application that performs basic arithmetic operations (addition, subtraction, multiplication, and division) using JavaFX buttons. Handle button click events for numbers and operators and display the result in a text field.
4. Create a JavaFX application with a password input field that dynamically assesses the strength of a password as it is typed. Use event handling to evaluate the password's complexity (for example, length, character types) and provide visual feedback to the user.
5. Design a custom event in JavaFX for a scenario of your choice. For example, create an application that simulates a game and fire a custom GameOverEvent when the game is over. Implement event listeners to respond to this custom event.
6. Develop a simple JavaFX email client that uses event filtering to confirm the deletion of important emails. When the user clicks the Delete button, trigger an event filter to check if the email is marked as important and display a confirmation dialog box before deleting it.
7. Create a JavaFX shopping cart application. Use event handling to allow users to add items to the cart, remove items, and automatically update the total price when items are added or removed from the cart.
8. Build a JavaFX application that enables the user to dynamically customize the user interface. Use event handling to change the background color, font size, and other visual aspects in real-time based on user selections.
9. Design a registration form with various input fields (name, email, and password) and a Submit button. Implement event handling to ensure that the Submit button is enabled only when all required fields are filled and display an error message otherwise.
10. Develop a JavaFX application where the user can drag and drop elements (for example, images or labels) within a canvas. Implement event handling for onMousePressed, onMouseDragged, and onMouseReleased events to achieve this functionality.



Session 15

Media with JavaFX

Welcome to the Session, **Media with JavaFX**.

In this session, we dive into the domain of media integration with JavaFX, a versatile framework for creating dynamic and engaging GUIs in Java. This session will provide knowledge to discover the capabilities of JavaFX in handling various media types, including images, audio, and video.

In this Session, you will learn to:

- Illustrate an overview of using media with JavaFX
- Explain properties and constructors of Media class
- Explain the process to play audio with JavaFX
- Explain the process to play video with JavaFX

15.1 Overview of Media with JavaFX

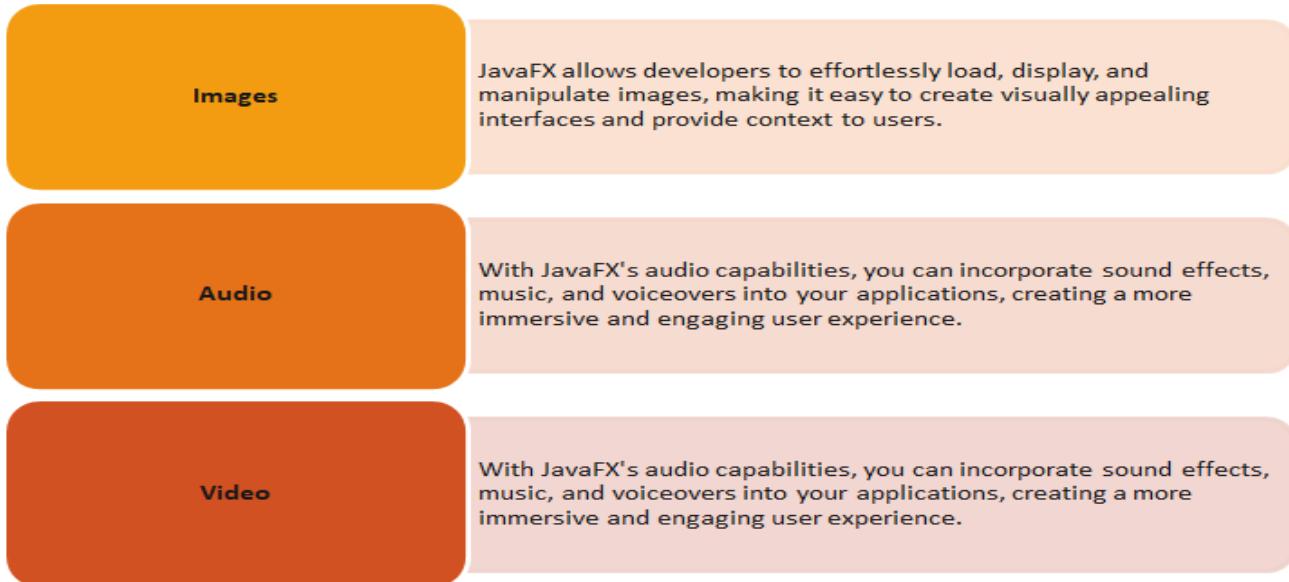
In the digital age, multimedia elements such as images, audio, and video play an integral role in enhancing user experiences and conveying information effectively. JavaFX, a robust and versatile framework for building graphical user interfaces in Java, empowers developers to seamlessly integrate these multimedia components into their applications.

Multimedia elements serve as powerful tools for engaging and captivating users. Whether one is developing a dynamic e-learning platform, an interactive entertainment application, or a data visualization tool, the integration of multimedia can significantly enhance the overall user experience. JavaFX equips developers with the tools and capabilities to harness the full potential of multimedia in their applications.



15.1.1 Key Components of Media Integration

Media integration with JavaFX revolves around three fundamental components:



JavaFX facilitates media handling through the `javafx.scene.media` package, which contains essential classes for this purpose. Within `javafx.scene.media`, there are following key classes and components:

- `javafx.scene.media.Media`
- `javafx.scene.media.MediaPlayer`
- `javafx.scene.media.MediaStatus`
- `javafx.scene.media.MediaView`

15.1.2 Media Events

The JavaFX media API is designed to be event-driven, allowing developers to handle media events using callback behaviors. Instead of manually writing code to respond to button clicks using an

`EventHandler`, JavaFX provides a mechanism to respond to media player events, such as `OnReady`, where `XXXX` represents the specific event name.

In JavaFX, one can utilize the `java.lang.Runnable` functional interfaces as callbacks, which are invoked when a media event occurs. When working with media content in JavaFX, one can create lambda expressions (implementing `java.lang.Runnable` interfaces) to be set on the `onReady` event.

Code Snippet 1 demonstrates the use of JavaFX's media capabilities to play audio or video content. It begins by importing the necessary JavaFX media classes. The code defines a string variable `mediaURL` to hold the location of the media file to be played. This URL should be set to the path of the media file to play. Next, a `Media` object named `media` is created by passing the `mediaURL` to it. This `Media` object represents the media source. A `MediaPlayer` named `mediaPlayer` is created, associated with the `media` object. The `MediaPlayer` is responsible for controlling the playback of the media.

A `Runnable` instance named `playMusic` is defined using a lambda expression. This `Runnable` instance contains the code to start playing the media using `mediaPlayer.play()`. Finally, the `playMusic` lambda expression is set as an event handler for the `onReady` event of the `MediaPlayer`. This means that when the media is ready to be played, the lambda expression will be executed, and the media playback will begin.

Code Snippet 1:

```
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;

// Define the media source URL
String mediaURL = "d:/media.mp4";

// Create a Media object
Media media = new Media(mediaURL);

// Create a MediaPlayer for the media
MediaPlayer mediaPlayer = new MediaPlayer(media);

// Create a Runnable using a lambda expression to play the media
Runnable playMusic = () -> mediaPlayer.play();

// Set the Runnable as an event handler for the onReady event
mediaPlayer.setOnReady(playMusic);
. . .
```

Table 15.1 lists various potential media and media player methods that are triggered by specific events.

Class	Set On Method	Description
Media	<code>setOnError()</code>	This method is triggered when an error is encountered.
MediaPlayer	<code>setOnEndOfMedia()</code>	The method is triggered when end of the media play is reached.
	<code>setOnError()</code>	This method is triggered when an error occurs.
	<code>setOnHalted()</code>	This method is triggered when the status of media changes to halted.
	<code>setOnMarker()</code>	This method is invoked when the Marker event is triggered.
	<code>setOnPaused()</code>	This method is invoked when a pause event occurs.
	<code>setOnPlaying()</code>	This method is invoked when the play event occurs.
	<code>setOnReady()</code>	This method is invoked when the media is in ready state.
	<code>setOnRepeat()</code>	This method is invoked when the repeat property is set.
	<code>setOnStalled()</code>	This method is invoked when the media player is stalled.
MediaView	<code>setOnError()</code>	This method is invoked when an error occurs in the media view.

Table 15.1: Media Player Events

15.2 Properties of Media with JavaFX

The properties of `Media` class are listed in Table 15.2. All the properties are read-only except for `onError`.

Property	Description
duration	The duration of the source media is expressed in seconds and this property is of the <code>Duration</code> class object type.
error	This property is assigned the value of a <code>MediaException</code> when an error occurs. It is of the <code>MediaException</code> class object type.

Property	Description
height	The height of the source media is measured in pixels, and it is represented as an integer-type property.
onError	The event handler that is invoked when an error occurs is set using the <code>setOnError()</code> method.
width	The width of the source media is expressed in pixels, and it is categorized as an integer-type property.

Table 15.2: Properties of Media Class

15.3 Constructors of Media with JavaFX

In JavaFX, the `Media` class provides several constructors to create a `Media` object, allowing developers to specify the media source and associated attributes. Table 15.3 lists all the primary constructors for the `Media` class.

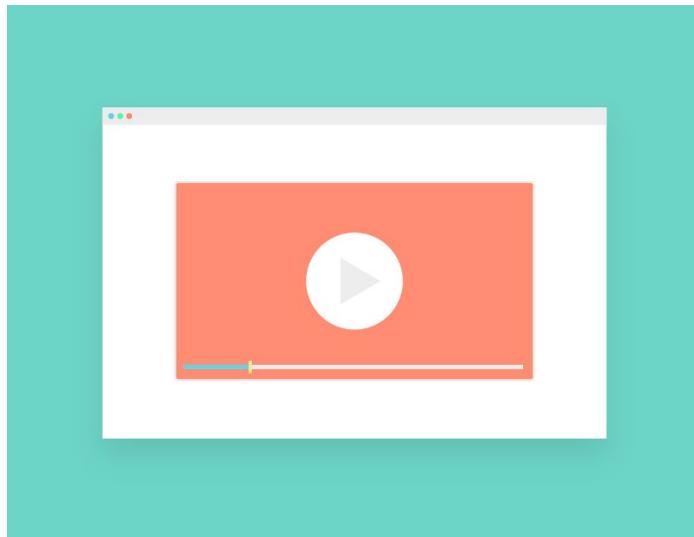
Constructor	Description	Example
Media(String source)	This constructor takes a <code>String</code> argument, which should be a valid URL or file path pointing to the media source. It creates a <code>Media</code> object based on the provided source location.	<code>Media media = new Media("file:/path/to/media.mp4");</code>
Media(URI uri)	This constructor accepts a <code>URI</code> object representing the media source location. It is an alternative way to create a <code>Media</code> object using a <code>URI</code> .	<code>URI mediaURI = new URI("file:/path/to/media.mp4");</code> <code>Media media = new Media(mediaURI.toString());</code>
Media(InputStream inputStream)	This constructor takes an <code>InputStream</code> as an argument, allowing to create a <code>Media</code> object from media data provided through the input stream. This is useful when you want to work with media content from sources such as databases or network streams.	<code>InputStream mediaInputStream = ...;</code> <code>// Initialize with //media data</code> <code>Media media = new Media(mediaInputStream);</code>

Table 15.3: Constructors of Media Class

These constructors enable developers to create `Media` objects based on different sources, whether they are local files, remote URLs, or data streams. Once there is a `Media` object, it can be used to create a `MediaPlayer` and manage media playback in a JavaFX application.

15.4 Playing Audio with JavaFX

JavaFX provides a convenient way to integrate audio playback into applications. Whether one is building a media player, a game, or any application that requires sound, JavaFX offers a set of classes and methods to manage audio efficiently.



Key Classes for Audio Playback

➤ Media Class:

The `Media` class is the starting point for playing audio in JavaFX. It represents the media source, which can be an audio file located on the local filesystem, a URL, or even an input stream. One can create a `Media` object by specifying the source location as specified in Code Snippet 2.

Code Snippet 2:

```
String audioFile = "file:/D:/audio.mp3";
Media media = new Media(audioFile);
```

➤ MediaPlayer Class:

Once developers have a `Media` object, they can create a `MediaPlayer` to control audio playback. The `MediaPlayer` class offers methods to play, pause, stop, and seek within the media. It also provides options for volume control and handling media events. Here is how they can create a `MediaPlayer` as specified in Code Snippet 3.

Code Snippet 3:

```
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

➤ MediaView Class (Optional):

To display video along with audio or simply visualize audio playback, one can use the `MediaView` class. It allows to embed audio and video content within a JavaFX application.

Playing Audio

To start audio playback, call the `play()` method on `MediaPlayer` instance as specified in Code Snippet 4.

Code Snippet 4:

```
MediaPlayer.play();
```

15.4.1 Step by Step Guide on How to Play Audio in JavaFX

Before beginning these steps, ensure the required JavaFX environment has been set up. Then, follow these steps to create an Audio Player application:

➤ Import Required Packages

Make sure to import the necessary JavaFX packages at the beginning of the Java file. Code Snippet 5 depicts this.

Code Snippet 5:

```
import javafx.application.Application;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.stage.Stage;
```

➤ Extend Application Class

Code Snippet 6 depicts this.

Code Snippet 6:

```
public class AudioPlayerApp extends Application {
    // Application code goes here
}
```

➤ Override the `start` Method

Inside `AudioPlayerApp` class, override the `start()` method. This is where one can set up audio playback. Code Snippet 7 depicts this.

Code Snippet 7:

```
@Override
public void start(Stage primaryStage) {
    // Audio playback code goes here
}
```

➤ Create a Media Object

Define the source of the audio using the `Media` class. One can specify a file path, URL, or input stream. Code Snippet 8 depicts this.

Code Snippet 8:

```
String audioFile = "file:/D:/audio.mp3";
Media media = new Media(audioFile);
```

➤ Create a MediaPlayer Object

Create a `MediaPlayer` object that utilizes the `Media` object that was just created. Code Snippet 9 depicts this.

Code Snippet 9:

```
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

➤ Play the Audio

Use the `play()` method to start audio playback. Code Snippet 10 depicts this.

Code Snippet 10:

```
mediaPlayer.play();
```

➤ Set the Stage Title and Show

Optionally, one can set the title for the application window and display it as shown in Code Snippet 11.

Code Snippet 11:

```
primaryStage.setTitle("Audio Player");
primaryStage.show();
```

➤ Add a Main Method

Finally, add a `main` method to launch the JavaFX application as shown in Code Snippet 12.

Code Snippet 12:

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

Code Snippet 13 contains the complete example of a simple JavaFX audio player application.

Code Snippet 13:

```
import javafx.application.Application;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.stage.Stage;

public class AudioPlayerApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Define the audio source
        String audioFile = "file:/D:/audio.mp3";
```

```

Media media = new Media(audioFile);

// Create a MediaPlayer
MediaPlayer mediaPlayer = new MediaPlayer(media);

// Play the audio
mediaPlayer.play();

// Set the stage title
primaryStage.setTitle("Audio Player");

// Show the stage
primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
}
}

```

15.5 Playing Video with JavaFX

JavaFX provides a straightforward way to incorporate video playback into applications. Whether one is building a video player, a multimedia presentation tool, or any application that involves video content, JavaFX offers a set of classes and methods to seamlessly manage video playback.



Key Classes for Video Playback

➤ **Media Class:**

The `Media` class serves as the starting point for video playback in JavaFX. It represents the video source, which can be a video file on the local file system, a URL, or even an input stream. One can create a `Media` object by specifying the source location. Code Snippet 14 depicts this.

Code Snippet 14:

```
String videoFile = "file:/D:/MyVideo.mp4";
```

```
Media media = new Media(videoFile);
```

➤ MediaPlayer Class:

Once developers have a `Media` object, they can create a `MediaPlayer` to manage video playback. The `MediaPlayer` class provides methods for playing, pausing, stopping, and seeking operations within the video. It also allows for volume control and handling video-related events. Here is how one can create a `MediaPlayer`. Code Snippet 15 depicts this.

Code Snippet 15:

```
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

➤ MediaView Class (Optional):

To display video content within the JavaFX application, one can utilize the `MediaView` class. It enables to embed video playback into the user interface. Code Snippet 16 depicts calling of `play()` method on a `MediaPlayer` instance.

Playing Video

To initiate video playback, one can simply call the `play()` method on `MediaPlayer` instance.

Code Snippet 16:

```
mediaPlayer.play();
```

15.5.1 Event-Driven Video Playback

JavaFX follows an event-driven model for video playback, similar to audio. One can set up event handlers to respond to various video-related events, such as when the video is ready to play (`onReady`), when an error occurs (`onError`), or when the video reaches its end (`onEndOfMedia`). Code Snippet 17 demonstrates example of event driven playback.

Code Snippet 17:

```
mediaPlayer.setOnReady(() -> {
    // Code to execute when the video is ready to play
});

mediaPlayer.setOnError(() -> {
    // Code to handle video playback errors
});
```

These event handlers allow to create interactive and responsive video applications.

15.5.2 Step by Step Guide on How to Play Video in JavaFX

Before beginning these steps, ensure the required JavaFX environment has been set up. Then, follow these steps to create a Video Player application:

➤ Import Required Packages

Make sure to import the necessary JavaFX packages at the beginning of the Java file. Code Snippet 18 depicts this.

Code Snippet 18:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;
```

➤ Extend Application Class

Code Snippet 19 creates a Java class that extends the Application class.

Code Snippet 19:

```
public class VideoPlayerApp extends Application {
    // Application code goes here
}
```

➤ Override the start Method

Inside the VideoPlayerApp class, override the start() method. This is where one will set up video playback. Code Snippet 20 depicts this.

Code Snippet 20:

```
@Override
public void start(Stage primaryStage) {
    // Video playback code goes here
}
```

➤ Create a Media Object

Define the source of audio using the Media class. One can specify a file path, URL, or input stream. Code Snippet 21 depicts this.

Code Snippet 21:

```
String videoFile = "file:/D:/MyVideo.mp4";
Media media = new Media(videoFile);
```

➤ Create a MediaPlayer Object

Code Snippet 22 depicts this.

Create a `MediaPlayer` object that utilizes the `Media` object that was just created.

Code Snippet 22:

```
MediaPlayer mediaPlayer = new MediaPlayer(media);
```

➤ **Create a `MediaView` (Optional)**

To display the video within the JavaFX application, create a `MediaView` as shown in Code Snippet 23.

Code Snippet 23:

```
MediaView mediaView = new MediaView(mediaPlayer);
```

➤ **Set Up the Scene**

Create a scene and add the `MediaView` to it (if it was created) as shown in Code Snippet 24.

Code Snippet 24:

```
Scene scene = new Scene(mediaView, 800, 600);
```

➤ **Set the Stage Title and Show**

Optionally, one can set the title for the application window and display it. Code Snippet 25 depicts this.

Code Snippet 25:

```
primaryStage.setTitle("Video Player");
primaryStage.setScene(scene);
primaryStage.show();
```

➤ **Play the Video**

Start video playback by calling the `play()` method on `MediaPlayer` instance as shown in Code Snippet 26.

Code Snippet 26:

```
mediaPlayer.play();
```

➤ **Add a Main Method**

Finally, add a main method to launch the JavaFX application as shown in Code Snippet 27.

Code Snippet 27:

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

Code Snippet 28 contains the complete example of a simple JavaFX video player application.

Code Snippet 28:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;
import javafx.scene.Group;
public class VideoPlayerApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Define the video source
        String videoFile = "file:/D:/MyVideo.mp4";
        Media media = new Media(videoFile);

        // Create a MediaPlayer
        MediaPlayer mediaPlayer = new MediaPlayer(media);

        // Create a MediaView (optional)
        MediaView mediaView = new MediaView(mediaPlayer);
        mediaPlayer.setAutoPlay(true);
        Group root = new Group();
        root.getChildren().add(mediaView);
        // Set up the scene
        Scene scene = new Scene(root, 800, 600);

        // Set the stage title
        primaryStage.setTitle("Video Player");

        // Set the scene and show the stage
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

When executed, the code will launch the application and autoplay the video.

15.6 Summary

- JavaFX equips developers with tools and capabilities to harness the full potential of multimedia in their applications.
- JavaFX packages (such as javafx.scene.media) must be imported to work with multimedia components effectively.
- Within javafx.scene.media, key classes and components include javafx.scene.media.Media, javafx.scene.media.MediaPlayer, javafx.scene.media.MediaStatus, and javafx.scene.media.MediaView.
- Media, MediaPlayer, and MediaView define many methods that are triggered by specific events.
- One can utilize the Media class and specify the source of an audio or video, specifying a file path, URL, or input stream.
- One can instantiate a MediaPlayer object with the designated Media source to manage audio or video playback.

15.7 Check Your Progress

1. What is the primary JavaFX class used for managing media content?
 - A) MediaView
 - B) MediaPlayer
 - C) MediaContent
 - D) MediaStream
2. Which JavaFX class represents the source of media, such as audio or video?
 - A) MediaSource
 - B) MediaStream
 - C) MediaFile
 - D) Media
3. How can you play media in JavaFX using a `MediaPlayer` object?
 - A) `media.play();`
 - B) `mediaPlayer.start();`
 - C) `media.playMedia();`
 - D) `mediaPlayer.playMedia();`
4. Which method is used to set an event handler when the media is ready to play?
 - A) `setOnPlay`
 - B) `setOnStart`
 - C) `setOnReady`
 - D) `setOnPause`
5. In JavaFX, what is the purpose of the `MediaView` class?
 - A) To represent media content
 - B) To control media playback
 - C) To display media content in the UI
 - D) To handle media events

15.7.1 Answers

1. B
2. D
3. A
4. C
5. C

Try It Yourself

1. Create a JavaFX application that plays a video file. Allow users to control playback (play, pause, and stop) through UI buttons.
2. Build an audio player application that can load and play multiple audio files. Implement a playlist feature to switch between tracks.
3. Develop a JavaFX program that demonstrates handling media events. Implement event handlers for various media events such as ready, error, and end of media.

Appendix

Sr. No.	Case Studies
1.	<p>Exercise 1: Online Bookstore Application using JavaFX</p> <p>Scenario: You are tasked with developing an online bookstore application using JavaFX. The application should provide users with a platform to browse, search, and purchase books. As a Java developer, you must create a user-friendly and responsive JavaFX application to facilitate these tasks.</p> <p>Part 1: Designing the User Interface</p> <p>a. Using JavaFX, design a user interface for the online bookstore. The UI should consist of following components:</p> <ul style="list-style-type: none"> • A search bar to search for books. • A book listing section that displays book covers, titles, authors, and prices. • A shopping cart to hold selected books. • A checkout page for completing the purchase. • Implement navigation between these sections using JavaFX layouts, buttons, and menus. <p>Part 2: Managing Book Data</p> <p>a. Create a data structure, such as a Java collection, to store book information. Each book should have attributes such as title, author, price, and a unique identifier.</p> <p>b. Populate the data structure with a selection of books for users to browse. You can hardcode the data initially.</p> <p>Part 3: Implementing Book Browsing and Searching</p> <p>a. Develop functionality to display books in the book listing section, including book covers, titles, authors, and prices.</p> <p>b. Implement a search feature that allows users to search for books by title or author. Display search results dynamically as users' type.</p> <p>Part 4: Managing the Shopping Cart</p> <p>a. Allow users to add books to their shopping cart. Create an interactive cart section that displays selected books with titles, quantities, and total prices.</p> <p>b. Implement the ability to adjust the quantity of items in the shopping cart and remove items if desired.</p>

	<p>Part 5: Checkout and Payment</p> <ul style="list-style-type: none"> a. Create a checkout page that collects user information such as name, address, and payment details (you can use mock data for this exercise). b. Simulate the payment process by showing a confirmation message and deducting the total from the user's account (use mock data). <p>Part 6: Additional Features</p> <ul style="list-style-type: none"> a. Allow users to view book details when they click a book in the listing section. Display information such as a book's description, genre, and reviews. b. Implement a user profile section where users can view their purchase history and manage their account details (name, address, password, and so on). <p>Part 7: Styling and Responsiveness</p> <ul style="list-style-type: none"> a. Use JavaFX CSS to style the application and make it visually appealing. b. Ensure that the application is responsive, adapting to different screen sizes and orientations. <p>Part 8: Error Handling and Validation</p> <ul style="list-style-type: none"> a. Implement error handling and validation for scenarios such as empty search queries, invalid payment information, and out-of-stock items. b. Display user-friendly error messages to guide users when issues arise.
2.	<p>Exercise 2: Weather Forecast Application using JavaFX</p> <p>Scenario: You have been tasked with creating a weather forecast application using JavaFX. The application should provide users with real-time weather information for various locations. As a Java developer, your goal is to design a user-friendly and informative JavaFX application for users to check weather conditions.</p> <p>Part 1: Designing the User Interface</p> <ul style="list-style-type: none"> a. Use JavaFX to design a user interface for a weather forecast application.

	<p>The UI should include following components:</p> <ul style="list-style-type: none">• A search bar for users to enter the location they want to check the weather for.• A display section that shows the current temperature, weather condition, humidity, wind speed, and an icon representing the weather.• A forecast section to display the weather forecast for the next few days. <p>b. Implement responsive design to ensure the application works well on different screen sizes.</p> <p>Part 2: Integrating Weather Data</p> <ol style="list-style-type: none">a. Choose a weather API (for example, OpenWeatherMap, WeatherAPI) to fetch real-time weather data for the application.b. Implement the API calls to retrieve weather information for a specified location. Use Java libraries such as HttpURLConnection or third-party libraries for making API requests.c. Parse the JSON or XML data received from the weather API to extract the required weather information. <p>Part 3: Displaying Weather Information</p> <ol style="list-style-type: none">a. Develop the functionality to display weather information in the application's display section. Include details such as temperature, weather condition, humidity, and wind speed.b. Use icons or images to represent the weather condition (for example sun, clouds, and rain). <p>Part 4: Weather Forecast</p> <ol style="list-style-type: none">a. Create a forecast section that displays the weather forecast for the next few days. Include details such as high and low temperatures, weather conditions, and dates.b. Implement the ability for users to view weather forecasts for multiple days by navigating through the forecast data. <p>Part 5: User Interaction</p> <ol style="list-style-type: none">a. Allow users to enter a location in the search bar and fetch weather information by pressing a button or pressing Enter key.
--	---

	<p>b. Ensure the application updates the weather information in real-time when the user enters a new location or refreshes the data.</p> <p>Part 6: Error Handling and Validation</p> <ul style="list-style-type: none"> a. Implement error handling for situations where the user enters an invalid location or there are issues with the weather API request. b. Display user-friendly error messages to guide users when issues arise. <p>Part 7: Styling and Responsiveness</p> <ul style="list-style-type: none"> a. Use JavaFX CSS to style the application and make it visually appealing. b. Ensure that the application is responsive, adapting to different screen sizes and orientations.
3.	<p>Project 3: Task Manager Dashboard using JavaFX</p> <p>Scenario: You are assigned to develop a task manager dashboard using JavaFX. The application should provide users with a platform to create, organize, and track their tasks. As a Java developer, your goal is to create a user-friendly and efficient JavaFX application for task management.</p> <p>Part 1: Designing the User Interface</p> <ul style="list-style-type: none"> a. Use JavaFX to design a user interface for the task manager dashboard. The UI should include following components: <ul style="list-style-type: none"> • A section for creating and adding new tasks with a task name and description. • A task list that displays existing tasks with details including task name, due date, status, and priority. • Buttons for adding tasks, updating tasks, deleting tasks, and filtering tasks. b. Implement responsive design to ensure the application works well on different screen sizes. <p>Part 2: Data Model for Tasks</p> <ul style="list-style-type: none"> a. Define a data model for tasks. Create a Java class to represent a task, including attributes such as task name, description, due date, status (for example, incomplete/completed), and priority. b. Use a Java Collection (for example, ArrayList) to store the task objects and manage the task data.

	<p>Part 3: Adding and Managing Tasks</p> <ul style="list-style-type: none"> a. Implement functionality to add new tasks. Users should be able to enter a task name, description, due date, and select a priority level when adding a task. b. Develop the ability to update existing tasks. Users should be able to modify task details, including task name, description, due date, status, and priority. c. Allow users to delete tasks from the list. <p>Part 4: Task List and Filtering</p> <ul style="list-style-type: none"> a. Display the list of tasks in the task list section. Each task should include details such as task name, due date, status, and priority. b. Implement filtering options to allow users to filter tasks by status (for example, incomplete/completed) or priority level. <p>Part 5: User Interaction and Validation</p> <ul style="list-style-type: none"> a. Enable users to interact with the application by clicking a task to view or edit its details. b. Implement validation checks to ensure that no essential information is missing when adding or updating a task. Display appropriate error messages when validation fails. <p>Part 6: Styling and Responsiveness</p> <ul style="list-style-type: none"> a. Use JavaFX CSS to style the application and make it visually appealing. b. Ensure that the application is responsive, adapting to different screen sizes and orientations.
c.	<p>Project 4: Online Food Ordering System using JavaFX</p> <p>Scenario: You have been assigned to develop an online food ordering system using JavaFX. The application should provide users with the ability to browse restaurant menus, add items to their order, and place food orders. As a Java developer, your task is to create an intuitive and efficient JavaFX application for online food ordering.</p> <p>Part 1: Designing the User Interface</p> <ul style="list-style-type: none"> a. Using JavaFX, design a user interface for the online food ordering system.

The UI should include following components:

- A list of restaurants available for ordering.
 - A menu section that displays restaurant menus with food items, descriptions, prices, and images.
 - A shopping cart to hold selected food items.
 - A checkout page for reviewing and placing orders.
- b. Implement responsive design to ensure the application works well on different screen sizes.

Part 2: Managing Restaurant Data

- a. Create a data model for restaurant information, including details such as restaurant name, cuisine type, menu items, and images.
- b. Populate the data structure with a selection of restaurants and their menus for users to browse. You can hardcode the data initially.

Part 3: Browsing Menus and Placing Orders

- a. Develop functionality to display restaurant menus, including food items, descriptions, prices, and images.
- b. Allow users to browse through the menu items and add items to their shopping cart. Include the ability to adjust item quantities and remove items.

Part 4: Managing the Shopping Cart

- a. Create a shopping cart that displays the selected food items with names, quantities, and total prices.
- b. Implement the ability for users to adjust item quantities and remove items from the cart.

Part 5: Checkout and Payment

- a. Design a checkout page where users can review the food items in their cart and place the order.
- b. Simulate the payment process by showing a confirmation message and deducting the total amount from the user's account (use mock data for payment).

Part 6: User Interaction and Validation

- a. Allow users to select a restaurant from the list and view their menus.

- b. Implement validation checks to ensure that users do not proceed to checkout without selecting food items.

Part 7: Error Handling and Validation

- a. Implement error handling for scenarios where users enter incorrect or incomplete information.
- b. Display user-friendly error messages to guide users when issues arise.

Part 8: Styling and Responsiveness

- a. Use JavaFX CSS to style the application and make it visually appealing.
- b. Ensure that the application is responsive, adapting to different screen sizes and orientations.