Programming in Java

Student Guide

Trademark Acknowledgements

All products are registered trademarks of their respective organizations.
All software is used for educational purposes only.

Programming in Java/SG/13-M07-V1.0

Printed at Repro India Ltd. Mahape, Navi Mumbai - 400 710.

COURSE DESIGN-STUDENT GUIDE

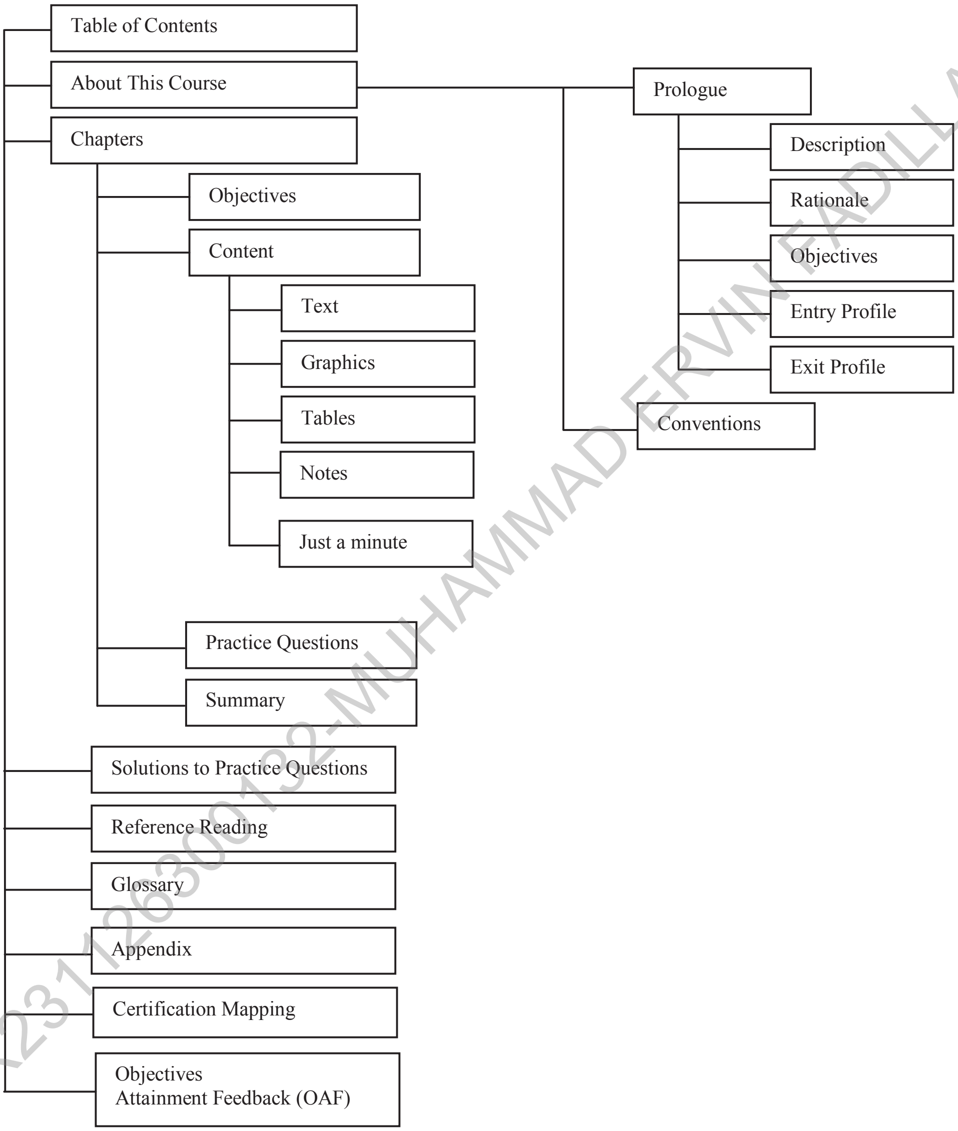


Table of Contents

About This Course

Chapter 1 – Implementing Inner Classes and Type Casting

Chapter 2 – Working with Regular Expressions and Localization

Chapter 3 – Working with Generics

Chapter 4 – Working with Collections

Chapter 5 – Working with Threads

Chapter 6 – Implementing Thread Synchronization and Concurrency

Chapter 7 – Working with Streams

Chapter 8 – Working with NIO Classes and Interfaces

Chapter 9 – Introduction to JDBC

Chapter 10 – Creating Applications Using Advanced Features of JDBC

Solutions to Practice Questions

Reference Reading

Glossary

Appendix

Certification Mapping

About This Course

Prologue

Description

The Programming in Java course helps the students to develop efficient and robust applications by using the Java programming language. It also describes how to create inner classes and generic classes. In addition, this course discusses the implementation of type casting, localization, threads, thread synchronization, and concurrency. Further, it discusses the various classes of the `java.util`, `java.io` and `java.nio` packages.

The course also explains how to work with regular expressions and how to create multithread applications. Further, it discusses the Java Database Connectivity architecture and implementation of database connectivity.

Rationale

Today, there are varied electronic devices available in the market. To work with these electronic devices, different applications are used. These applications are developed by using different programming languages, such as C, C++, Java, and C#. However, the applications developed by using programming languages like C and C++ do not support cross-platform portability.

Java is an object oriented programming language that helps to develop real-life portable applications. We can create both, CUI-based application and GUI-based application, by using Java. The code reusability feature of Java enables software developers to upgrade the existing applications without rewriting the entire code of the application.

Objectives

After completing this course, the students will be able to:

- Implement inner classes and type casting
- Work with regular expressions and localization
- Work with generics
- Work with collections
- Work with threads
- Implement thread synchronization and concurrency
- Work with streams
- Work with NIO classes and interfaces
- Get familiar with JDBC
- Create applications using advanced features of JDBC

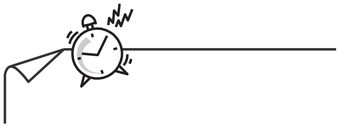
Entry Profile

The students who want to take this course should have basic knowledge of logic building and effective problem solving.

Exit Profile

After completing this course, the students will be able to develop object-based applications in Java.

Conventions

Convention	*Indicates…*
	Note
	*Just a minute*
	Placeholder for an activity

Chapter 1

Implementing Inner Classes and Type Casting

At times, you need to develop a helper class that should provide a very specific functionality to another class in the program. However, at the same time, you also want to ensure that the helper class is not accessible to other class. This can be achieved by defining the helper class inside the class that uses the functionality. A class defined within another class is called an inner class. The inner class increases encapsulation. In addition, it makes the code readable and maintainable.

There are times when you need to convert the data from one type into another. For this, Java supports type casting.

This chapter focuses on creating inner classes and implementing type casting.

Objectives

In this chapter, you will learn to:

- Create inner classes
- Implement type casting

Creating Inner Class

Consider a scenario where you are developing a Java application to process credit card payments for clients, such as retail stores. In this application, you have created a `CreditCard` class that models the credit card used for payment by a client. The `CreditCard` class defines private fields for sensitive information, such as the credit card number, expiry date, and CVV number. You need to create a `CreditCardValidator` class that contains the business logic to validate a credit card using an object of the `CreditCard` class. However, you cannot create a separate `CreditCardValidator` class as it will not be able to access the private fields of the `CreditCard` class required for performing the validation. In such a scenario, you can use an inner class. *Inner classes* are the classes defined inside another class. Java provides the following four types of inner classes:

- Regular inner class
- Static inner class
- Method-local inner class
- Anonymous inner class

Regular Inner Class

Consider the scenario of a Bill Desk application used in the departmental store. This application requires login credentials to login. Therefore, you need to create a class named `Login`, which will have the username and password fields. As these details should not be accessible to other classes, you need to declare the username and password as private fields. However, the login credentials provided by the user need to be validated. Therefore, you need to define a class named `ValidateCredentials` to implement the validation logic. However, if this class is declared as an outer class, you cannot access the private fields. Therefore, to implement the preceding functionality, you need to create a regular inner class.

A *regular inner class* is a class whose definition appears inside the definition of another class. The regular inner class is similar to a member of an outer class. Therefore, the regular inner class shares all the features of a member of a class. The following code snippet demonstrates an example for the regular inner class:

```
public class Login
{
    private String username;
    private String password;
    class ValidateCredentials
    {
        public void validate()
        {
            // the private fields username and password are accessible here
        }
    }
}
```

In the preceding code snippet, a class, `ValidateCredential`, is created inside the class, `Login`. The `validate()` method of `ValidateCredential` can access all the members of the `Login` class. Therefore, the private fields, `username` and `password`, are accessible to the `validate()` method.

In order to instantiate an inner class outside the outer class, you first need to get the reference of the outer class object and create an object of the inner class, as shown in the following code snippet:

```
Login obj=new Login ();
Login.ValidateCredentials cobj= obj.new ValidateCredentials ();
```

Static Inner Class

Consider a scenario of the Payment Gateway application. In this application, you need to implement the logic to validate various details, such as credit card, debit card, and net banking account login details. In order to meet the preceding requirement, you can use a regular inner class. However, to access a regular inner class outside the class, you need to instantiate the outer class. Thereafter, by using the outer class object, you need to instantiate the inner class. This means that for each request for a validation, two objects, one for the outer class and another for the inner class, will be created in the heap.

Consider an example of an enterprise application where a large number of users are making numerous validation requests. Therefore, the amount of objects that will get assigned to the heap will increase. This will adversely affect the performance of the application.

Therefore, to address the preceding challenge, you can create a static inner class. As a result, you will be able to minimize the amount of objects, which are required to access the inner class functions. This is because in order to access a static inner class, you do not need its enclosing outer class object.

Static inner classes are inner classes marked with the static modifier. The following code snippet demonstrates an example for static inner classes:

```
class PaymentDetails
{
    private static String fieldToValidate;
    static class ValidateCreditCards
    {
        void validate()
        {
           /* code be added */
        }
    }
    static class ValidateDebitCards
    {
        void validate()
        {
           /* code be added */
        }
    }
    static class ValidateNetBankingAccount
    {
        void validate()
        {
           /* code be added */
        }
    }
}
```

In the preceding code snippet, `ValidateCreditCards`, `ValidateDebitCards` and `ValidateNetBankingAccount` are static inner classes defined inside the `PaymentDetails` class.

You need to use the following code snippet to instantiate the `ValidateNetBankingAccount` static inner class outside the `PaymentDetails` class:

```
PaymentDetails.ValidateNetBankingAccount validator=new
PaymentDetails.ValidateNetBankingAccount();
```

Note

A static inner class cannot directly access the member variables of the outer class. Therefore, to access the member variables, you need to create the instance of the outer class inside the inner class.

Method-local Inner Class

A *method-local inner* class is defined inside the method of the enclosing class. Because the class is defined inside a method, it needs to be instantiated within the same method.

Consider the following code snippet of the method-local inner class:

```
public class MethodLocalInner
{
    private String x = "MyOuterClass";
    void display()
    {
        final String z = "local variable";
        System.out.println(x);
        class Inner
        {
            public void print()
            {
                System.out.println("Outer x: " + x);
                System.out.println("Local variable z: " + z);
            }
        }
        Inner obj = new Inner();
        obj.print();
    }
    public static void main(String args[])
    {
        MethodLocalInner m = new MethodLocalInners();
        m.display();
    }
}
```

In the preceding code snippet, a `MethoLocalInner` class with a method named `display()` is created. In the `display()` method, a class named `Inner` is created with a method named `print()`.

Anonymous Inner Class

Consider a scenario where you need to develop a gaming application by using the Swing components. In this application, you need to design a startup screen interface with a start button. In addition, you want to implement the functionality wherein the game should begin when the start button is clicked. To implement this functionality, you need to override the `actionPerformed()` method of the `ActionListener` interface, as shown in the following code snippet:

```
class StartUpAction implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        /* code to be added here */
    }
}
public class StartUpScreen
{
   JFrame jf;
   JButton start;
   StartUpScreen()
   {
       jf=new JFrame("Startup Screen");
       start=new JButton("Start");
       jf.setLayout(new FlowLayout());
       jf.add(start);
       jf.setSize(300,300);
       jf.setVisible(true);
       start.addActionListener(new StartUpAction());
   }
}
```

The `StartUpAction` class is accessible to other classes in the application. However, the functionality defined in the `StartUpAction` class is not required by other classes in the application. Therefore, you need not define an outer class. In addition, the instance of this class is used only once. Therefore, with the help of an anonymous inner class, the functionality provided by the `StartUpAction` class can be implemented. An *anonymous inner class* has no name, and it is either a subclass of a class or an implementer of an interface. In case of the anonymous class, a curly brace is followed by the semicolon.

Consider the following code snippet to implement an anonymous inner class:

```
start.addActionListener(new ActionListener()
{
           public void actionPerformed(ActionEvent e)
           {
                /* code to be added here */
           }
});
```

In the preceding code snippet, an anonymous inner class of the type, `ActionListener`, is created and the `actionPerformed()` method is overridden.

Just a minute:

Which one of the following inner classes can be defined without the name of the class?

1. *Regular inner class*
2. *Static inner class*
3. *Method-local inner class*
4. *Anonymous inner class*

Answer:

4. *Anonymous inner class*

Activity 1.1: Creating Inner Classes

Implementing Type Casting

You have learned that Java supports inheritance and polymorphism. Therefore, it is possible to assign a sub class object to the base class. For example, Employee is a base class that has two child classes, Manager and Supervisor. The Employee class reference can hold the object of child classes. However, by default, you cannot assign the base class reference to a sub class reference. Therefore, it is necessary to convert from one type into other. For this, Java supports the following types of type casting:

- Type casting primitive data types
- Type casting objects

Type Casting Primitive Data Types

A primitive data type supports the following types of type casting:

- Implicit casting
- Explicit casting

Implicit Casting

Implicit Casting refers to an automatic conversion of one data type into another. It occurs if both the data types are compatible with each other. For example, you can assign a value of the `int` type to a variable of the `long` data type.

Consider the following code snippet of the implicit casting:

```
int a = 100;
long b = a;
```

In the preceding code snippet, implicit casting occurs automatically. This type of casting usually occurs when we put the smaller data type, such as `byte`, into the larger data type, such as `int`. However, if we try to assign a `double` value to an `integer` type, we will get an error as we are trying to put a larger value into a smaller container.

Explicit Casting

Explicit Casting occurs when one data type cannot be implicitly converted into another data type. In case of explicit conversion, you must convert the data type into the compatible type.

For example, the `int` type is large enough to store a `byte` value. Therefore, it does not require any explicit conversion. However, assigning an `int` value to a `byte` type would require explicit conversion as the range of the `byte` type is smaller as compared to the `int` type.

Consider the following code snippet that casts the integer number to the byte type:

```
int a = 10;
byte b = (byte) a;
System.out.println(b);
```

In the preceding code snippet, explicit casting has occurred. This type of casting usually occurs when we put the larger data type, such as `int`, into the smaller data type, such as `byte`.

Type Casting Object

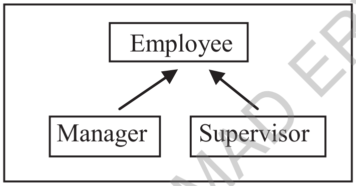
The casting of object references depends on the relationship of the classes involved in the same hierarchy. Any object reference can be assigned to a reference variable of the type, `Object`, as the `Object` class is a superclass of every Java class.

An object supports the following types of type casting:

- Upcasting
- Downcasting

Upcasting

Upcasting is usually done along the class hierarchy in a direction from the derived class to the base class. For example, the `Manager` and `Supervisor` classes extend the `Employee` class, as shown in the following figure.



The Process of Upcasting

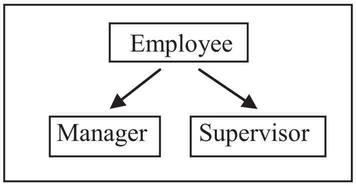
Then, a `Manager` class object can be treated as if it were an `Employee` object, as shown in following code snippet:

```
Employee emp1 = new Employee();
Employee emp2 = new Employee();
Manager mgr = new Manager();
Supervisor spr = new Supervisor();
emp1 = mgr;
emp2=spr;
```

In the preceding code snippet, an object of the `Employee`, `Manager`, and `Supervisor` classes are created. The reference of the `Manager` class is assigned to the `Employee` class. Similarly, the reference of the `Supervisor` class is assigned to the `Employee` class.

Downcasting

Downcasting is usually done along the class hierarchy in a direction from the base class towards the derived classes, as shown in the following figure.



The Process of Downcasting

Consider the following code to implement downcasting:

```
class Employee
{
    public Employee()
    {
        System.out.println("Employee ");
    }
    public void print()
    {
        System.out.println("Employee Details");
    }
}
class Manager extends Employee
{
    public Manager()
    {
        System.out.println("Manager");
    }
    public void print()
    {
        System.out.println("Manager Details");
    }
}
class Supervisor extends Employee
{
    public Supervisor()
    {
        System.out.println("Supervisor");
    }
    public void print()
    {
        System.out.println("Supervisor Details");
    }
}
class Test
{
    public static void main(String agrs[])
    {
        Employee emp1, emp2;
```

```
           Manager m1;
            Supervisor s1;
            emp1 = new Manager();
            emp2 = new Supervisor();
            Employee emp3=new Manager();
            m1 = (Manager) emp1;
            s1 = (Supervisor) emp2;
            emp3.print();
        }
    }
```

In the preceding code, the objects, `emp1` and `emp2`, are downcasted to `Manager` and `Supervisor` class objects, respectively. In addition, the statement, `Employee emp3=new Manager();`, creates the reference variable of `Employee` class which stores the object of the `Manager` class. This type of object creation is called virtual method invocation. The statement, `emp3.print();`, will invoke the print method of the `Manager` class.

Just a minute:

Which one of the following casts allows automatic conversion from one data type into another?

1. *Upcasting*
2. *Downcasting*
3. *Implicit casting*
4. *Explicit casting*

Answer:

3. *Implicit casting*

Practice Questions

1. State whether the following statement is true or false.

   Upcasting is done along the class hierarchy in a direction from a derived class to the base class.

2. Which one of the following inner classes is defined inside the method of the enclosing class?

   a. Regular inner class

   b. Static inner class

   c. Method-local inner class

3. Which one of the following options correctly denotes an anonymous inner class?

   a. It is defined inside the method of the enclosing class.

   b. It is marked with a static modifier.

   c. It is similar to the member of the outer class.

   d. It does not have a name and is either a subclass of a class or an implementer of an interface.

4. State whether the following statement is true or false.

   Downcasting is usually done along the class hierarchy in a direction from the base class towards the derived classes.

5. State whether the following statement is true or false.

   Method-local inner class is defined inside the method of the enclosing class.

Summary

In this chapter, you learned that:

- A class defined within another class is called an inner class.
- Java provides the following four types of inner classes:
  - Regular inner class
  - Static inner class
  - Method-local inner class
  - Anonymous inner class
- A regular inner class is a class whose definition appears inside the definition of another class. The regular inner class is similar to a member of an outer class.
- Static inner classes are inner classes marked with the static modifier.
- A method-local inner class is defined inside the method of the enclosing class. Because the class is defined inside a method, it needs to be instantiated within the same method.
- Java supports the following types of type casting:
  - Type casting primitive data types
  - Type casting objects
- A primitive data type supports the following types of type casting:
  - Implicit casting
  - Explicit casting
- Implicit conversion refers to an automatic conversion of one data type into another. It occurs if both the data types are compatible with each other.
- Explicit conversion occurs when one data type cannot be implicitly converted into another data type.
- In case of explicit conversion, you must convert the data type into the compatible type.
- The casting of object references depends on the relationship of the classes involved in the same hierarchy.
- An object supports the following types of type casting:
  - Upcasting
  - Downcasting
- Upcasting is usually done along the class hierarchy in a direction from the derived class to the base class.
- Downcasting is usually done along the class hierarchy in a direction from the base class towards the derived classes.

Chapter 2

Working with Regular Expressions and Localization

At times, you need to develop applications that provide a functionality, such as search or filter, which requires processing of strings. To develop such applications, you need to write lengthy code, which is a time-consuming task. To solve this issue, Java supports regular expressions also known as regex. *Regular expressions* allow you to match a character sequence with another string.

At times, you want to develop applications that can be used worldwide. These applications should be able to display various pieces of information, such as text, currency, and date, according to the region where they are deployed. To meet this requirement, Java supports localization.

In this chapter, you will learn about processing strings using regular expressions and implementing localization.

Objectives

In this chapter, you will learn to:

- Process strings using regex
- Implement localization

Processing Strings Using Regex

Consider a scenario where you need to develop a Java application that provides the functionality to search the text, hello, in a particular file. To accomplish this task, Java provides a support for regular expressions. A regular expression is a string that defines a character sequence called pattern. This pattern can be matched against another string. To work with regular expressions, Java provides two classes, `Pattern` and `Matcher`, in the `java.util.regex` package.

In addition, Java provides character classes to work with a set of characters. Further, to identify the number of occurrences of a pattern in a string, Java supports quantifiers.

Note
In regular expressions, a group of characters enclosed within square brackets is called a character class.

Working with the Pattern and Matcher Classes

The `Pattern` class represents a compiled regular expression. As the `Pattern` class does not define any constructor, the `Pattern` class reference is obtained by using the static method, `compile()`, of the `Pattern` class.

The following code snippet is used to create a reference of the `Pattern` class:

```
Pattern myPattern = Pattern.compile("Expression");
```

In the preceding code snippet, the regular expression, `Expression`, is passed to the `compile()` method, which returns the object of the `Pattern` class. Then, the `Pattern` object is used to create a `Matcher` object, as shown in the following code snippet:

```
Matcher myMatcher = myPattern.matcher("Expression");
```

To compare the two expressions, the `Matcher` class provides the `matches()` method. The `matches()` method returns the value, `true`, if the expression in the matcher object matches the pattern specified in the pattern object. Otherwise, it returns the value, `false`. The following code snippet shows the usage of the `matches()` method:

```
boolean myBoolean = myMatcher.matches();
```

The following table lists the most commonly used methods of the `Pattern` class.

Method	***Description***	***Example***

`String pattern()`	*Is used to return the compiled regular expression.*	`String s1 = "pattern";`
		`Pattern myPattern = Pattern.compile(s1);`
		`String s2 = myPattern.pattern();`
		`System.out.println(s2);`
		Output:
		`pattern`

`String[] split(CharSequence input, int limit)`	*Is used to split the given input sequence based on the pattern and the limit.*	`Pattern pattern = Pattern.compile(":");`
		`String[] split = pattern.split("One:two:three",2);`
		`for (String element : split)`
		`{`
		`System.out.println("element = " + element);`
		`}`
		Output:
		`element = One`
		`element = two:three`

`static boolean matches(String regex,CharSequence input)`	*Is used to compile the given regular expression and attempt to match the given input against it.*	`boolean matches = Pattern.matches("is", "is");`
		`System.out.println("matches = " + matches);`
		Output:
		`matches = true`

The Methods of the Pattern Class

The following table lists the most commonly used methods of the `Matcher` class.

Method	***Description***	***Example***

`Matcher appendReplacement(StringBuffer sb, String replacement)`	*Is used to process the input character sequence specified in a matcher by adding the input character sequence to the* `StringBuffer` *variable. If the match specified in* `Pattern` *is found in* `Matcher`*, then it is replaced by the replacement string and is appended to the* `StringBuffer` *variable. Further, the remaining character sequence is truncated.*	`Pattern pattern=`
		`Pattern.compile("John");`
		`Matcher matcher=`
		`pattern.matcher("John does this, and John does that");`
		`StringBuffer s3 = new StringBuffer();`
		`while(matcher.find())`
		`{`
		`matcher.appendReplacement(s3, "sam");`
		`System.out.println(s3.toString());`
		`}`
		Output:
		`sam`
		`sam does this, and sam`

`StringBuffer appendTail(StringBuffer sb)`	*Is used to process the input character sequence specified in* `Matcher` *by adding the input character sequence to the* `StringBuffer` *variable. If the match specified in* `Pattern` *is found in* `Matcher`*, then it is replaced by the replacement string and is appended to the* `StringBuffer` *variable. Further, the remaining character sequence is appended.*	`Pattern pattern=`
		`Pattern.compile("John");`
		`Matcher matcher=`
		`pattern.matcher("John does this, and John does that");`
		`StringBuffer s3 = new StringBuffer();`
		`while(matcher.find())`
		`{`
		`matcher.appendReplacement(s3, "sam");`
		`System.out.println(s3.toString());`
		`}`
		`matcher.appendTail(s3);`
		`System.out.println(s3.toString());`
		Output:
		`sam`
		`sam does this, and sam`
		`sam does this, and sam does that`

Method	*Description*	*Example*

`String replaceAll(String replacement)`	*Is used to replace every subsequence of the input sequence that matches the pattern with the given replacement string.*	`Pattern pattern= Pattern.compile("John");`
		`Matcher matcher= pattern.matcher("John does this, and John does that");`
		`String s2 = matcher.replaceAll("sam");`
		`System.out.println("replaceAll = " + s2);`
		Output:
		`replaceAll = sam does this, and sam does that`

`String replaceFirst(String replacement)`	*Is used to replace the first subsequence of the input sequence that matches the pattern with the given replacement string.*	`Pattern pattern= Pattern.compile("John");`
		`Matcher matcher= pattern.matcher("John does this, and John does that");`
		`String s2 = matcher.replaceFirst("sam");`
		`System.out.println("replaceFirst = " + s2);`
		Output:
		`replaceFirst = sam does this, and John does that`

`int start(int group)`	*Is used to return the start index of the subsequence captured by the given group during the match operation.*	`Pattern pattern = Pattern.compile("a");`
		`Matcher matcher = pattern.matcher("This is a text");`
		`while(matcher.find())`
		`{`
		`System.out.println("Match started at:"+ matcher.start(0));`
		`}`
		Output:
		`Match started at:8`

Method	*Description*	*Example*

int end(int group)	*Is used to return the offset after the last character of the subsequence is captured by the given group during the match operation.*	Pattern pattern =
		Pattern.compile("a");
		Matcher matcher =
		pattern.matcher("This is a
		text");
		while(matcher.find())
		{
		System.out.println("Match ended
		at:"+ matcher.end(0));
		}
		Output:
		Match ended at:9

The Methods of the Matcher Class

Working with Character Classes

Consider a scenario where you need to search certain words in a file that can have the prefix “ a”, “b”, or “c” and suffix “at”, such as, bat and cat. For this, regular expressions provide character classes. The character class specifies the characters that will successfully match a single character from a given input string.

The following table lists the constructs of the character class.

Construct	*Description*
[def]	*Match succeeds if the given input string starts with the characters: d, e, or f.*
[^def]	*Match succeeds if the given input string starts with any character except d, e, or f (negation).*
[a-zA-Z]	*Match succeeds if the given input string starts with any character between a to z or A to Z, inclusive (range).*
[b-e[n-q]]	*Match succeeds if the given input string starts with any character between b to e or n to q.*
[a-z&&[abc]]	*Match succeeds if the given input string starts with characters: a, b, or c (intersection).*
[a-z&&[^bcd]]	*Match succeeds if the given input string starts with any character between a to z, except b, c, and d :* [ae-z] *(subtraction).*

Construct	*Description*
[a-z&&[^n-p]]	*Match succeeds if the given input string starts with any character between a to z and not n to p: [a-mq-z] (subtraction).*

The Constructs of the Character Class

Consider the following code that demonstrates the use of the character class:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class TestRegx
{
    public static void main(String[] args)
    {
        Pattern myPattern = Pattern.compile("[abc]at");
        Matcher myMatcher = myPattern.matcher("bat");
        boolean myBoolean = myMatcher.matches();
        if(myBoolean)
        System.out.println("Expression Matched");
        else
        System.out.println("Expression Not Matched");
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Expression Matched
```

In the preceding code, `myBoolean`, will have the value, `true`, only when the first letter matches one of the characters defined by the character class.

Working with Quantifiers

Consider a scenario where you want to specify the number of times a character or a sequence of characters appears in an expression. For example, you want to find the occurrence of the characters, Ro, in the expression, “When in Rome, do as the Romans”. To accomplish this task, a regular expression provides quantifiers that help you specify the number of occurrences of a string matched with the specified string. In addition, quantifiers enable you to easily select a range of characters in the files. Quantifiers are of the following three types:

- **Greedy**: Is used to match with the longest possible string that matches the pattern.
- **Reluctant**: Is used to match with the shortest possible string that matches the pattern.
- **Possessive**: Is used to match the regular expression with the entire string. It matches only when the whole string satisfies the criteria.

The following table describes the different types of quantifiers.

Greedy	***Reluctant***	***Possessive***	***Description***
X?	*X??*	*X?+*	*X, once or not at all*
*X**	*X*?*	*X*+*	*X, zero or more times*
X+	*X+?*	*X++*	*X, one or more times*
X{n}	*X{n}?*	*X{n}+*	*X, exactly n times*
X{n,}	*X{n,}?*	*X{n,}+*	*X, at least n times*
X{n,m}	*X{n,m}?*	*X{n,m}+*	*X, at least n times but not more than m times*

The Types of Quantifiers

Consider the following code snippet to test the occurrence of the word beginning with Ro in the expression, When in Rome, do as the Romans:

```
String text="When in Rome, do as the Romans ";
String textSplit[]=text.split(" ");
Pattern myPattern=Pattern.compile("Ro.+");
for(int i=0;i<textSplit.length;i++)
{
    Matcher myMatcher=myPattern.matcher(textSplit[i]);
    boolean myBoolean =myMatcher.matches();
    System.out.println(myBoolean);
}
```

In the preceding code snippet, the `split()` method splits the expression "`When in Rome, do as the Romans`". Thereafter, the pattern, `Ro`, is compared with each split expression. The `System.out.println(myBoolean);` statement prints the value, `true`, if the first two characters of the word are "`Ro`".

Consider the following code that accepts the pattern and matcher from the user:

```
import java.util.regex.*;
import java.util.*;
public class PatternMethods
{
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the desired pattern: ");
        String pattern = input.nextLine();
        System.out.println("Enter the text: ");
        String matcher = input.nextLine();
        Pattern myPattern2 = Pattern.compile(pattern);
        Matcher myMatcher2 = myPattern2.matcher(matcher);
        Boolean myBoolean2 = myMatcher2.matches();
        boolean b = myBoolean2;
        if (b == true)
        {

           System.out.println("I found the text:" + myMatcher2.group() + "\n"
                    + "Starting at:" + myMatcher2.start() + "\n"
                    + "Ending at index:" + myMatcher2.end());

        }
        else if (b == false)
        {
            System.out.println("No match found");
        }

   }
}
```

While executing the preceding code, if you provide the pattern as `.*Friday` and matcher as `XFriday`, then the following output will be displayed:

```
I found the text:XFriday
Starting at:0
Ending at index:7
```

The preceding output is generated as the pattern uses the greedy quantifier. Since the quantifier is greedy, initially, the entire input string is consumed. Because the entire input string does not match with the pattern, the matcher traverses back by one letter at a time, until the rightmost occurrence of "`Friday`" has been found. At this point, the match succeeds and the search ends.

While executing the preceding code, if you provide the pattern as `.*?Friday` and matcher as `XFriday`, then the following output will be displayed:

```
I found the text:XFriday
Starting at:0
Ending at index:7
```

The preceding output is generated as the pattern uses the reluctant quantifier. Since the quantifier is reluctant, initially, no character of the input string is consumed. Thereafter, the characters of the string are consumed one by one.

While executing the preceding code, if you provide the pattern as `.*+Friday` and matcher as `XFriday`, then the following output will be displayed:

```
No match found
```

The preceding output is generated as the pattern uses the possessive quantifier. Since the quantifier is possessive, initially, the entire input string is consumed. Because this quantifier does not traverse back, it fails to find a match.

Just a minute:

Which one the following methods is used for compiling the given regular expression and attempting to match the given input against the specified pattern?

1. `static boolean matches(String regex,CharSequence input)`
2. `Pattern pattern()`
3. `static Pattern compile(String regex,int flags)`
4. `Matcher appendReplacement(StringBuffer sb,String replacement)`

Answer:

1. `static boolean matches(String regex,CharSequence input)`

Activity 2.1: Processing Strings Using Regex

Implementing Localization

Consider a scenario where you want to develop a hotel management application that can be used worldwide. You want that the text in the application should be displayed according to the region. For example, if the application is used in France, the text should be displayed in French and the currency should be displayed in Euros. To develop such applications, you need to implement localization, which is a process of customizing the application to a specific locale and culture. Localization can be implemented on different types of data, such as date, currency, and text. To localize different types of data, it is necessary to determine the language and country. To determine the language, Java provides a predefined set of language codes, such as `zh` for Chinese and `en` for English. In addition, it provides a predefined set of country codes, such as `AU` for Australia and `CN` for China. Further, to work with localization, the `Locale` class of the `java.util` package is used. The instance of the `Locale` class can be created, as shown in the following code snippet:

```
Locale l=new Locale("de","DE");
```

In the preceding code snippet, `de` specifies the language code and `DE` specifies the country code.

Localizing Date

To localize the date, you need to use the various date formats. To determine the date format according to the locale, you can use the `java.text.DateFormat` class. To display the date in German, the following code is used:

```
import java.text.DateFormat;
import java.util.*;
public class DateDemo {
    public static void main(String args[]){
        DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, new
Locale("de","DE"));
        String date = df.format(new Date());
        System.out.println(date);
    }
}
```

The preceding code generates the following output:

```
4. Juni 2013
```

Note
The preceding output may vary based on the current system's date.

In the preceding code, `DateFormat.getDateInstance` is used to specify a formatting style for the German locale. `DateFormat.Long` specifies the description of the date according to the locale. The `String date = df.format(new Date());` statement formats the current date into the date string.

Similarly, to display the time in German, the following code is used:

```
import java.text.*;
import java.util.*;
public class TimeDemo {
    public static void main(String args[]){
        DateFormat df = DateFormat.getTimeInstance(DateFormat.LONG, new
Locale("de","DE"));
        String time = df.format(new Date());
        System.out.println(time);
    }
}
```

The preceding code generates the following output:

```
13:36:28 IST
```

Note
The preceding output may vary based on the current system's time.

In the preceding code, the `getTimeInstance()` method is used to specify a formatting style for the German locale. `DateFormat.Long` specifies the description of the date according to the locale. The `String time = df.format(new Date());` statement formats the current time into the string format.

Localizing Currency

To localize the currency, you need to use various currency formats. To determine the currency format according to the locale, you can use the `java.text.NumberFormat` class. To display the currency in German, the following code is used:

```
import java.text.*;
import java.util.*;
public class CurrencyDemo {
    public static void main(String args[]){
        NumberFormat nft = NumberFormat.getCurrencyInstance(new
Locale("de","DE"));
        String formatted = nft.format(1000000);
        System.out.println(formatted);
    }
}
```

The preceding code generates the following output:

```
1.000.000,00 €
```

In the preceding code, `NumberFormat.getCurrencyinstance` returns a currency format for the locale. The `String formatted = nft.format(1000000);` statement formats the currency into a string.

Localizing Text

To localize text, you need to use the resource bundles. A resource bundle is a property file that contains the locale specific data. To work with data stored in a resource bundle, you need to use the `ResourceBundle` class. `ResourceBundle` is a class that encapsulates a set of resources. To access a particular resource, you need to obtain a reference of the `ResourceBundle` class and call its `getObject()` method.

To have a locale-specific object, a program needs to load the `ResourceBundle` object by using the `getBundle` method, as shown in the following code snippet:

```
ResourceBundle myResourceBundle = ResourceBundle.getBundle("MessagesBundle");
```

In the preceding code snippet, a `MessagesBundle` resource bundle has been loaded that contains the localized resource. When a the `getBundle()` method is called, the `ResourceBundle` tries to load a resource file with a variation of the name that was specified. It searches the resource bundle with the explicitly specified locale values appended to the name. It then searches the resource bundle with the default Locale's values. For example, if the default Locale is `Locale.US`, then `getBundle()` will load the resource bundle file in the US locale.

Consider a scenario where you need to develop a Java application that displays the message, Welcome, in German or Chinese. For this, you need to create the German and Chinese resource bundles.

You can create the German resource bundle file, such as `MessageBundle_de.properties`, and add the following code snippet inside the file to localize the message, Welcome, in German:

```
message=willkommen
```

You can create the Chinese resource bundle file, such as `MessageBundle_zn.properties`, and add the following code snippet inside the file to localize the message, Welcome, in Chinese:

```
message=歡迎
```

Note

To create a language specific resource bundle file, you need to specify the properties filename as `<resourcebundle name>_<language code>.properties`.

The following code demonstrates how the text is localized:

```
import java.util.Locale;
import java.util.ResourceBundle;
public class TestLocale {
    public static void main(String args[]){
        Locale l1=new Locale("de","DE");
        ResourceBundle rb1 = ResourceBundle.getBundle("MessageBundle",l1);
        System.out.println(rb1.getString("message"));
        Locale l2=new Locale("zn","ZN");
        ResourceBundle rb2 = ResourceBundle.getBundle("MessageBundle",l2);
        System.out.println(rb2.getString("message"));
        }

    }
```

In the preceding code, the `getBundle()` method is used to load the `ResourceBundle` class when the program needs a locale-specific object. To retrieve an object from the resource bundle, the `getString()` method is used.

Just a minute:

Which one of the following statements denotes the correct way to create an instance of the `Locale` class?

1. `Locale l=new Locale(“de”, “DE”);`
2. `Locale l=new Locale(“DE”, “de”);`
3. `Locale l=new Locale(‘de’, ‘DE’);`
4. `Locale l=new Locale(‘DE’, ‘de’);`

Answer:

1. `Locale l=new Locale(“de”, “DE”);`

Activity 2.2: Implementing Localization

Practice Questions

1. State whether the following statement is true or false.

   Localization is the process of customizing the application to a specific locale and culture.

2. Which one of the following pattern class methods is used to return the regular expression from which the pattern was compiled?

   a. `String pattern()`
   b. `static boolean matches(String regex,CharSequence input)`
   c. `static Pattern compile (String regex)`

3. Which one of the following quantifiers is used to match with the shortest possible string that matches the pattern?

   a. Greedy
   b. Reluctant
   c. Possessive

4. Identify the correct option by using which the match succeeds if the given input string starts with any character between a and z, except b, c, and d.

   a. `[a-z&&[abc]]`
   b. `[b-z&&[^bcd]]`
   c. `[a-z&&[^n-p]]`
   d. `[abc]`

5. Identify the correct option that is used to match the pattern with the longest possible string.

   a. Greedy
   b. Reluctant
   c. Possessive

Summary

In this chapter, you learned that:

- Regular expressions allow you to match a character sequence with another string.
- A regular expression is a string that defines a character sequence called pattern.
- To work with regular expressions, Java provides two classes, `Pattern` and `Matcher`, in the `java.util.regex` package.
- Java provides character classes to work with a set of characters.
- To identify the number of occurrences of a pattern in a string, Java supports quantifiers.
- The `Pattern` class represents a compiled regular expression.
- As the `Pattern` class does not define any constructor, the `Pattern` class reference is obtained by using the static method, `compile()`, of the `Pattern` class.
- The character class specifies the characters that will successfully match a single character from a given input string.
- A regular expression provides quantifiers that help you specify the number of occurrences of a string matched with the specified string.
- Quantifiers are of the following three types:
  - Greedy
  - Reluctant
  - Possessive
- Localization is a process of customizing the application to a specific locale and culture.
- Localization can be implemented on different types of data, such as date, currency, and text.
- To localize different types of data, it is necessary to determine the language and country.
- To determine the language, Java provides a predefined set of language codes, such as `zh` for Chinese and `en` for English.
- To work with localization, the `Locale` class of the `java.util` package is used.
- To localize the date, you need to use the various date formats. To determine the date format according to the locale, you can use the `java.text.DateFormat` class.
- To localize the currency, you need to use various currency formats. To determine the currency format according to the locale, you can use the `java.text.NumberFormat` class.
- To localize text, you need to use the resource bundles.
- A resource bundle is a property file that contains the locale specific data. To work with data stored in a resource bundle, you need to use the `ResourceBundle` class.
- To have a locale-specific object, a program needs to load the `ResourceBundle` object by using the `getBundle()` method.

Working with Generics

At times, you need to create interfaces, classes, and methods that may work with different types of data. For this, you need to write a robust and type-safe code, which is a tedious and error-prone task. To overcome the overhead of writing the type-safe code, Java provides the *generics* feature. Generics allow you to create generalized interfaces, classes, and methods.

This chapter focuses on the creation of the user-defined generic classes and methods and implementation of type-safety.

Objectives

In this chapter, you will learn to:

- Create user-defined generic classes and methods
- Implement type-safety

Creating User-defined Generic Classes and Methods

Consider a scenario where you need to develop a class with a method, which can return various objects, such as `Integer`, `String`, and `Double`. To implement the functionality, you need to create generic classes and methods. Generics mean parameterized types that enable you to write the code that can work with many types of object. In addition, it allows you to provide the enhanced functionality to classes and methods by making them generalized. The generic classes and methods are created so that the references and methods can accept any type of object.

Creating Generic Classes

Generics enable you to generalize classes. This means that a reference variable, an argument of a method, and a return type can be of any type. In the declaration of a generic class, the name of the class is followed by a *type parameter* section. The type parameter section is represented by angular brackets, <>, that can have one or more types of parameters separated by commas. The following syntax is used to create generic classes:

```
class [ClassName]<T>
{

}
```

You can create a generic class by using the following code snippet:

```
class GenericClassDemo<T>
{

}
```

Consider an example where you want to create a generic class to set and get the `Integer` and `String` values. For this, you can use the following code:

```
public class GenericClassDemo<T>
{
    private T t;

    public void setValue(T t)
    {
        this.t = t;
    }

    public T getValue()
    {
        return t;
    }

    public static void main(String[] args)
    {
        GenericClassDemo<Integer> iobj = new GenericClassDemo<Integer>();
        iobj.setValue(10);
        System.out.println(iobj.getValue());
        GenericClassDemo<String> sobj = new GenericClassDemo<String>();
        sobj.setValue("Ten");
        System.out.println(sobj.getValue());
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
10
Ten
```

In the preceding code, the `GenericClassDemo` class is created with two methods, `setValue()` and `getValue()`. `T` represents the type of data, which can accept any object, such as `Integer` and `String`. The `T` parameter in the preceding code is used as a reference type, return type, and method argument.

Creating a Generic Method

In the generic class, a method can use the type parameter of the class, which automatically makes the method generic. Consider the following code of the generic method inside the generic class:

```
public class GenericClassDemo<T>
{
    private T t;

    public void setValue(T t)
    {
        this.t = t;
    }

    public T getValue()
    {
        return t;
    }
}
```

In the preceding code, the `setValue()` and `getValue()` methods are using the class's type parameter.

However, you can declare a generic method, which contains its own one or more type parameters. The declaration of a generic method contains the type parameter that is represented by angular brackets, <>. The following syntax is used to create a generic method:

```
public <Type Parameter> [Return Type] [MethodName](Argument list…)
{
}
```

You can create a generic method, as shown in the following code snippet:

```
public <T> T showValue(T val)
{
}
```

The type parameter section can have one or more type of parameters separated by commas. Further, it is also possible to create a generic method inside a non generic class.

Consider the following code of the generic method inside the non generic class:

```
public class GenericMethodDemo
{
    public <M> M display(M val)
    {
        return val;
    }

    public static void main(String[] args)
    {
        GenericMethodDemo obj = new GenericMethodDemo();

        System.out.println("The generic method is called with String value: "
+ obj.display("Test"));
        System.out.println("The generic method is called with Double value: "
+ obj.display(7.5));
        System.out.println("The generic method is called with Boolean value:
" + obj.display(true));
        System.out.println("The generic method is called with Integer value:
" + obj.display(10));

    }
}
```

Once the preceding code is executed, the following output is displayed:

```
The generic method is called with String value: Test
The generic method is called with Double value: 7.5
The generic method is called with Boolean value: true
The generic method is called with Integer value: 10
```

In the preceding code, the `GenericMethodDemo` class contains the `display()` method that is used to display the values. In addition, inside the `main()` method, the display method is called four times with four different types of values.

Implementing Type-safety

Consider an example where you want to create an object of a generic class. To create the object, you need to invoke the constructor of a generic class with the required type of parameters, such as `Integer` or `String`, which increases the length of the code. However, to reduce the coding, Java provides the flexibility to leave the type parameters empty as long as the compiler can guess or judge the type of the argument from the context. In addition, Java provides wildcards that allow you to achieve inheritance in a type parameter.

Using a Generic Type Inference

You have learned how to create generic classes and methods. In order to use those classes or methods, you need to understand how Java infers the arguments that are provided as the type parameters. While using a generic type, you must provide the type argument for every type parameter declared for the generic type. The type argument list is a comma-separated list that is delimited by angular brackets and follows the type name. For example, consider the following code of a generic type with two type arguments:

```
class Pair<X, Y>
{

    private X first;
    private Y second;

    public Pair(X a1, Y a2)
    {
        first = a1;
        second = a2;
    }

    public X getFirst()
    {
        return first;
    }

    public Y getSecond() {
        return second;
    }

    public static void main(String[] args)
    {
        Pair<String, Integer> obj1 = new Pair<String, Integer>("Test", 1);
        System.out.println("String is " + obj1.getFirst());
        System.out.println("Integer is " + obj1.getSecond());

        Pair<Integer, String> obj2 = new Pair<Integer, String>(2, "Demo");
        System.out.println("Integer is " + obj2.getFirst());
        System.out.println("String is " + obj2.getSecond());

    }
}
```

Once the preceding code is executed, the following output is displayed:

```
String is Test
Integer is 1
Integer is 2
String is Demo
```

In the preceding code, the `Pair` class contains two methods, `getFirst()` and `getSecond()`, which return the value of the `first` and `second` variables. In the `main()` method, two objects have been created that are of the `<String, Integer>` and `<Integer, String>` types.

Java provides a new feature, *type inference*, which enables you to invoke the constructor of a generic class with an empty set of type parameters, <>. The empty set of type parameters can be used as long as the compiler can infer the type arguments from the context.

Consider the preceding example in which you can use the empty set of type parameters while creating the objects of the `Pair` class, as shown in the following code snippet:

```
Pair<String, Integer> obj1 = new Pair<>("Test", 1);

Pair<Integer, String> obj2 = new Pair<>(2, "Demo");
```

Note
The pair of angular brackets, <>, is informally called the diamond operator.

Using Wildcards

Generics follow a simple rule of declaration, which says the type of reference variable declaration must match the type passed to the actual object. This means that in generics, a subclass cannot be passed as a subtype of a superclass, as shown in the following code snippet:

```
WildCardDemo<Number> obj = new WildCardDemo<Integer>();
```

Generics only allow the following type of declaration:

```
WildCardDemo<Integer> obj = new WildCardDemo<Integer>();
```

Or

```
WildCardDemo<Number> obj = new WildCardDemo<Number>();
```

However, Java provides wildcard arguments to pass a subclass as a subtype of a superclass in generics. The wildcard arguments are the special constructs that are used as a type argument in a generic type instance. There are various types of wildcards.

The following table lists the most commonly used wildcard arguments.

Wildcard	***Description***
`? extends`	*It specifies that the type can be anything that extends from the class specified after the* `extends` *keyword. It is a bounded wildcard.*
`? super`	*It specifies that the type can be anything that is the base class of the class specified after the* `super` *keyword. It is a bounded wildcard.*
`?`	*It specifies that the type can be anything. It doesn't need to be a subtype or super type of any class. It is an unbounded wildcard.*

The Wildcard Arguments

Note

A bounded wildcard is used to bound any type of object by using the keywords: `extends`*,* `implements`*, and* `super`*. An unbounded wildcard does not bind any type of object and does not associate with any keyword.*

Consider an example where you want to compare the values of two objects. However, you want to compare only the numeric objects. For this, you can use the following code:

```
public class WildCardDemo<T> {

    private T t;

    public void setValue(T t) {
        this.t = t;
    }

    public T getValue() {
        return t;
    }
    public boolean compare(WildCardDemo<? extends Number> wcd)
    {
        if(t == wcd.t)
        {
            return true;

        }
        else
        {
            return false;
        }

    }
```

```
        public static void main(String[] args)
        {
            WildCardDemo<Integer> obj1 = new WildCardDemo<Integer>();
            obj1.setValue(10);

            WildCardDemo<String> obj2 = new WildCardDemo<String>();
            obj2.setValue("Test");

            System.out.println("Value of first object: " +obj1.getValue());
            System.out.println("Value of second object: " +obj2.getValue());


            System.out.println("Are both equal? " + obj1.compare(obj2));
    //Compilation Error

        }
    }
```

In the preceding code, the `WildCardDemo` class is created with three methods: `setValue()`, `getValue()`, and `compare()`. The `compare()` method uses wildcards so that an object of `Number` or one of its subtype can be passed to it at runtime. When the `compare()` method is called, a compilation error is raised because `obj2` is of type, `String`, and `obj1` is of type, `Integer`.

In the preceding example, the `<? extends Number>` wildcard is used, which tells that the specified object must inherit the `Number` class.

Just a minute:

Which one of the following wildcards is used to specify the type that is inherited from the specified class?

1. `<?>`
2. `<? extends >`
3. `<extends ?>`
4. `<? extends ?>`

Answer:

2. `<? extends >`

Activity 3.1: Working with Generics

Practice Questions

1. Which one of the following symbols is used to define type parameters?

   a. `{ }`

   b. `[ ]`

   c. `( )`

   d. < >

2. State whether the following statement is true or false.
   A non generic class can contain a generic method.

3. Identify the correct syntax of a generic class.

   a. `public <T> class Myclass { }`

   b. `public class <T> Myclass { }`

   c. `public class Myclass <T> { }`

   d. `public class Myclass {<T>}`

4. Identify the correct syntax of a generic method.

   a. `public <T> T myMethod()`

   b. `public myMethod<T> T()`

   c. `public <T> myMethod()`

   d. `public myMethod T()`

5. Which one of the following wildcards specifies that the type can be anything?

   a. `?`

   b. `? extends`

   c. `? super`

   d. `? implements`

Summary

In this chapter, you learned that:

- Generics enable you to generalize classes, which mean that a reference variable, an argument of a method, and a return type can be of any type.
- The type parameter section is represented by angular brackets, <>, that can have one or more types of parameters separated by commas.
- In the generic class, a method can use the type parameter of the class, which automatically makes the method generic.
- You can declare a generic method, which contains its own one or more type parameters.
- The declaration of a generic method contains the type parameter that is represented by angular brackets, <>.
- Java provides wildcards that allow you to achieve inheritance in a type parameter.
- Java provides a new feature, type inference, which enables you to invoke the constructor of a generic class with an empty set of type parameters, <>.
- Java provides wildcard arguments to pass a subclass as a subtype of a superclass in generics.
- Java provides the following types of wildcards:
  - `? extends`
  - `? super`
  - `?`

Working with Collections

At times, you need to handle a collection of objects. This task can be performed with the help of arrays. However, arrays work on a single type of data. Therefore, to work with the collection of different types of object, Java provides the collection framework. This framework provides interfaces, classes, and algorithms for implementing data structures. The collection framework is used for various purposes, such as storing, retrieving, and manipulating objects.

This chapter explains how to use various interfaces, such as `List`, `Set`, `Map`, and `Deque`, to create a collection of objects. In addition, it discuss about the sorting functionalities.

Objectives

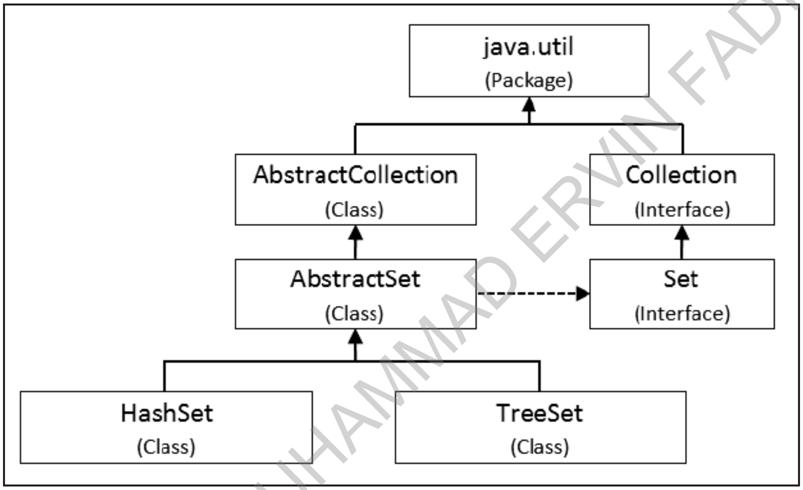
In this chapter, you will learn to:

- Use the Set interface
- Use the List interface
- Use the Map interface
- Use the Deque interface
- Implement sorting

Using the Set Interface

Consider a scenario where you need to develop a chat application. In this application, you can implement a set of functionalities to create a chat group dynamically from the list of available contacts. Further, members can be added and removed from the chat group as per the user preference. To implement this functionality, you can create a collection of contacts. However, to ensure the uniqueness of the contacts in the chat group, you need to create the collection using the `Set` interface.

The `Set` interface is used to create a collection of unique objects. The following figure shows the class hierarchy of the `java.util` package, which implements the `Set` interface.



The Class Hierarchy of the java.util Package

The preceding figure shows that the `java.util` package contains the `AbstractCollection`, `AbstractSet`, `HashSet`, and `TreeSet` classes and the `Collection` and `Set` interfaces. The `AbstractSet` class implements the `Set` interface and extends the `AbstractCollection` class, which is further extended by the `HashSet` and `TreeSet` classes. In addition, the `Set` interface extends the `Collection` interface.

The package, `java.util`, provides the `Iterator` interface to traverse through the set collection. The reference of the `Iterator` interface can be obtained using the `iterator()` method of the `Set` interface. The `Iterator` interface contains various methods, such as `hasNext()` and `next()`, which help to traverse the set collection.

Working with the HashSet Class

The `HashSet` class provides the implementation of the `Set` interface that enables you to create a set in which insertion is faster because it does not sort the elements. The `HashSet` class provides various constructors that can be used to create an instance.

The following table lists the commonly used constructors of the HashSet class.

Constructor	***Description***
HashSet()	*Creates an empty instance of HashSet.*
HashSet(Collection<? extends E> c)	*Creates an instance with the specified collection.*
HashSet(int initialCapacity)	*Creates an empty instance of HashSet with the specified initial capacity. The initial capacity is the size of HashSet that grows or shrinks automatically, whenever objects are added or removed to/ from HashSet.*

The Constructors of the HashSet Class

The following table lists the commonly used methods of the HashSet class.

Method	***Description***
boolean add(E e)	*Is used to add the specified object to HashSet, if it is not present. If present, it leaves HashSet unchanged and returns false.*
void clear()	*Is used to remove all the objects from HashSet.*
boolean remove(Object o)	*Is used to remove the specified object from HashSet.*
int size()	*Is used to get the number of objects in HashSet.*

The Methods of the HashSet Class

Consider the following code snippet to create a collection using a HashSet class:

```
import java.util.*;
class Contacts
{
  /* code to create contacts */
}
public class ChatGroup
{
    public static void main(String args[])
    {
     Contacts a=new Contacts();
     Contacts b=new Contacts();
     Contacts c=new Contacts();
     HashSet<Contacts> hs=new HashSet<Contacts>();
     hs.add(a);
     hs.add(b);
```

```
        hs.add(c);
        }
    }
```

In the preceding code snippet, the generic type object of the HashSet class is created that contains a collection of the Contacts objects. The add() method of the HashSet class is used to add the Contacts objects to the HashSet object.

You can use the following code snippet to remove the Contacts object from the HashSet object:

```
hs.remove(c);
```

You can use the following code snippet to find the number of objects available in the HashSet object:

```
hs.size();
```

You can use the following code snippet to traverse through the HashSet object:

```
Iterator i = hs.iterator();
    while(i.hasNext())
    {
        System.out.println(i.next());
    }
```

In the preceding code snippet, the iterator() method is used to retrieve the Iterator reference. In addition, the hasNext() method is used to check if there are more elements in HashSet, and the next() method is used to retrieve the element.

You have learned how generics resolve type safety issues in the Java applications. In addition, you have also used the generic HashSet collection where you need to specify the type of objects that the collection would store. However, there are several existing applications that have been developed before generics were introduced and such applications should continue running. To ensure interoperability of such applications, Java also provides support for non generic collections. In the previous example, a collection of objects is created by specifying the generic type, <Contact>. In this context, the collection is of the Contact objects only. However, you can also use the non generic HashSet collection that can store any type of objects. For this, you can use the following code snippet:

```
Contacts a=new Contacts();
HashSet hs=new HashSet();
hs.add(a);
hs.add("String object");
hs.add(new Integer(3));
```

In the preceding code snippet, a non generic HashSet collection is created that contains a collection of different objects.

Note

Raw type is the name of the generic class or interface without any type arguments.

Working with the TreeSet Class

The `TreeSet` class provides the implementation of the `Set` interface that enables you to create a sorted set of objects. The insert or add object process is slow as it sorts the objects on every insertion. However, `TreeSet` automatically arranges the objects in a sorted order, which enhances the search of data from a large amount of information. The `TreeSet` class provides various constructors that can be used to create an instance of `TreeSet`. The following table lists the commonly used constructors of the `TreeSet` class.

Constructor	***Description***
`TreeSet()`	*Creates an empty instance of `TreeSet`.*
`TreeSet(Collection<? extends E> c)`	*Creates an instance with the specified collection.*

The Constructors of the TreeSet Class

The following table lists the commonly used methods of the `TreeSet` class.

Method	***Description***
`boolean add(E e)`	*Is used to add the specified object to `TreeSet`, if it is not present.*
`void clear()`	*Is used to remove all the objects from `TreeSet`.*
`boolean remove(Object o)`	*Is used to remove the specified object from `TreeSet`.*
`int size()`	*Is used to get the number of objects in `TreeSet`.*

The Methods of the TreeSet Class

Consider the following code to add and remove the objects to/ from a `TreeSet` collection:

```
import java.util.Iterator;
import java.util.TreeSet;

public class TreeSetDemo
```

```
{
    public static void main(String[] args)
    {
        TreeSet<Integer> obj = new TreeSet<Integer>();
        Integer iobj1 = new Integer(114);
        Integer iobj2 = new Integer(111);
        Integer iobj3 = new Integer(113);
        Integer iobj4 = new Integer(112);

        System.out.println("Size of TreeSet is: " + obj.size());

        obj.add(iobj1);
        obj.add(iobj2);
        obj.add(iobj3);
        obj.add(iobj4);
        obj.add(iobj2);

        System.out.println("\nTreeSet after adding the objects: " + obj);
        System.out.println("Size of TreeSet after adding objects: " +
obj.size());

        obj.remove(iobj3);
        obj.remove(iobj1);

        System.out.println("\nTreeSet after removing the objects: " + obj);
        System.out.println("Size of TreeSet after removing objects: " +
obj.size());

        System.out.println("\nThe final TreeSet: ");
        Iterator i = obj.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

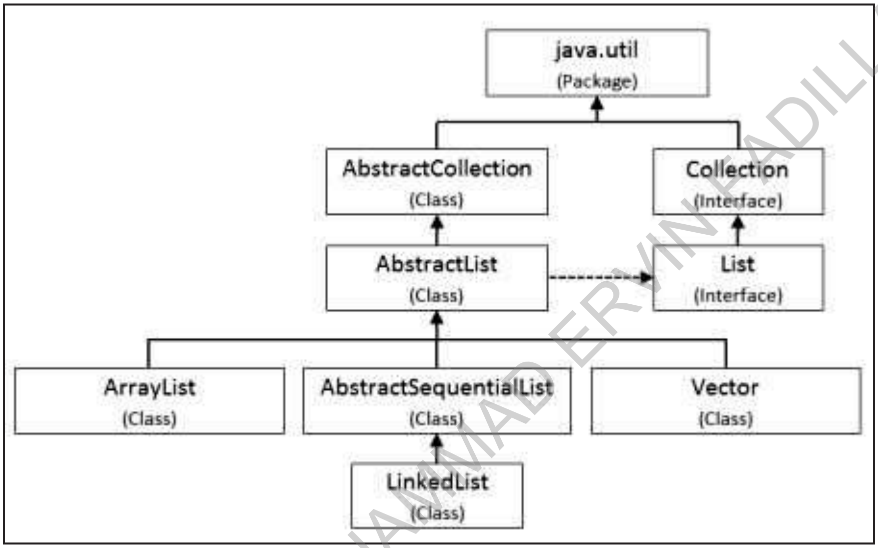
Once the preceding code is executed, the following output is displayed:

```
Size of TreeSet is: 0

TreeSet after adding the objects: [111, 112, 113, 114]
Size of TreeSet after adding objects: 4

TreeSet after removing the objects: [111, 112]
Size of TreeSet after removing objects: 2

The final TreeSet:
111
112
```

In the preceding code, the `TreeSet` object contains a collection of the `Integer` objects. The `add()` method of the `TreeSet` class is used to add the `Integer` objects to the `TreeSet` object. Further, two objects are removed using the `remove()` method. The `Iterator` interface is used to iterate the objects in the set one after the other.

Using the List Interface

The List interface is used to create an ordered collection of objects, which can contain duplicate objects. In this collection, the objects can be inserted one after the other or at the specified position in the list. Similarly, the objects can be accessed one after the other or from the specified position in the list. The following figure shows the class hierarchy of the java.util package that implements the List interface.



The Class Hierarchy of the java.util Package

The preceding figure shows that the java.util package contains the AbstractCollection, AbstractList, ArrayList, AbstractSequentialList, Vector, and LinkedList classes and the Collection and List interfaces. The AbstractCollection class is extended by the AbstractList class, which is then extended by the ArrayList, AbstractSequentialList, and Vector classes. The AbstractList class implements the List interface, which extends the Collection interface. The LinkedList class extends the AbstractSequentialList class.

The package, java.util, provides the ListIterator interface to traverse through the list collection. The reference of the ListIterator interface can be obtained using the listIterator() method of the List interface. The ListIterator interface contains various methods, such as hasNext(), hasPrevious(), next(), and previous(), which help to traverse the list in both directions.

Working with the ArrayList Class

The ArrayList class provides the implementation of the List interface. The ArrayList class enables you to create a resizable array. Its size increases dynamically when more elements are added. ArrayList proves to be efficient in searching an object in the list and inserting the objects at the end of the list.

The ArrayList class provides various constructors that can be used to create an instance. The following table lists the commonly used constructors of the ArrayList class.

Constructor	***Description***
ArrayList()	*Creates an empty instance of ArrayList.*
ArrayList(Collection<? extends E> c)	*Creates an instance with the specified collection.*
ArrayList(int initialCapacity)	*Creates an empty instance of ArrayList with the specified initial capacity. The initial capacity is the size of ArrayList that grows or shrinks automatically, whenever an object is added or removed to/ from ArrayList.*

The Constructors of the ArrayList Class

The following table lists the commonly used methods of the ArrayList class.

Method	***Description***
boolean add(E e)	*Is used to add the specified object at the end of ArrayList.*
void clear()	*Is used to remove all the objects from ArrayList.*
E get(int index)	*Is used to retrieve the object at the specified index from ArrayList.*
E remove(int index)	*Is used to remove the object at the specified index from ArrayList.*
boolean remove(Object o)	*Is used to remove the first occurrence of the specified object from ArrayList.*
int size()	*Is used to get the number of objects in ArrayList.*

The Methods of the ArrayList Class

Consider the following code to add and remove the objects to/ from an ArrayList collection:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> obj = new ArrayList<String>();
        String sobj1 = new String("Element 1");
        String sobj2 = new String("Element 2");
        String sobj3 = new String("Element 3");
        String sobj4 = new String("Element 4");

        System.out.println("Size of ArrayList is: " + obj.size());

        obj.add(sobj1);
        obj.add(sobj2);
        obj.add(sobj3);
        obj.add(sobj4);
        obj.add(sobj1);

        System.out.println("\nArrayList after adding the objects: " + obj);
        System.out.println("Size of ArrayList after adding objects: " +
obj.size());

        obj.remove(2);
        obj.remove(sobj4);

        System.out.println("\nArrayList after removing the objects: " + obj);
        System.out.println("Size of ArrayList after removing objects: " +
obj.size());

        System.out.println("\nThe final ArrayList: ");
        ListIterator i = obj.listIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

In the preceding code, the `ArrayList` object contains a collection of the `String` objects. The `add()` method of the `ArrayList` class is used to add the `String` objects. Further, two objects are removed using the `remove()` method. The `ListIterator` interface is used to iterate the objects in the list one after the other.

Once the preceding code is executed, the following output is displayed:

```
Size of ArrayList is: 0

ArrayList after adding the objects: [Element 1, Element 2, Element 3, Element
4, Element 1]
Size of ArrayList after adding objects: 5

ArrayList after removing the objects: [Element 1, Element 2, Element 1]
Size of ArrayList after removing objects: 3
```

```
The final ArrayList:
Element 1
Element 2
Element 1
```

Working with the LinkedList Class

The `LinkedList` class provides the implementation of the `List` interface. The `LinkedList` class enables you to create a *doubly-linked list*. The doubly-linked list is a set of sequentially linked records called nodes where each node contains two links that are the references of the previous node and the next node in the sequence, respectively. `LinkedList` can be traversed in the forward or backward direction. `LinkedList` proves to be efficient in insertion and deletion of objects from the list.

The `LinkedList` class provides various constructors that can be used to create an instance. The following table lists the commonly used constructor of the `LinkedList` class.

Constructor	***Description***
`LinkedList()`	*Creates an empty instance of `LinkedList`.*
`LinkedList(Collection<? extends E> c)`	*Creates an instance with the specified collection.*

The Constructors of the LinkedList Class

The following table lists the commonly used methods of the `LinkedList` class.

Method	***Description***
`boolean add(E e)`	*Is used to add the specified object at the end of `LinkedList`.*
`void clear()`	*Is used to remove all the objects from `LinkedList`.*
`E get(int index)`	*Is used to retrieve the object at the specified index from `LinkedList`.*
`E remove(int index)`	*Is used to remove the object at the specified index from `LinkedList`.*
`boolean remove(Object o)`	*Is used to remove the first occurrence of the specified object from `LinkedList`.*
`int size()`	*Is used to get the number of objects in `LinkedList`.*

The Methods of the LinkedList Class

Consider the following code to add and remove the objects to/ from a `LinkedList` collection:

```
import java.util.LinkedList;
import java.util.ListIterator;

public class LinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList<Integer> obj = new LinkedList<Integer>();
        Integer iobj1 = new Integer(101);
        Integer iobj2 = new Integer(102);
        Integer iobj3 = new Integer(103);
        Integer iobj4 = new Integer(104);

        System.out.println("Size of LinkedList is: " + obj.size());

        obj.add(iobj1);
        obj.add(iobj2);
        obj.add(iobj3);
        obj.add(iobj4);
        obj.add(iobj1);

        System.out.println("\nLinkedList after adding the objects: " + obj);
        System.out.println("Size of LinkedList after adding objects: " +
obj.size());

        obj.remove(iobj2);
        obj.remove(iobj3);

        System.out.println("\nLinkedList after removing the objects: " +
obj);
        System.out.println("Size of LinkedList after removing objects: " +
obj.size());

        System.out.println("\nThe final LinkedList: ");
        ListIterator i = obj.listIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Size of LinkedList is: 0

LinkedList after adding the objects: [101, 102, 103, 104, 101]
Size of LinkedList after adding objects: 5

LinkedList after removing the objects: [101, 104, 101]
Size of LinkedList after removing objects: 3
```

```
The final LinkedList:
101
104
101
```

In the preceding code, the LinkedList object contains a collection of the Integer objects. The add() method of the LinkedList class is used to add the Integer objects. Further, two objects are removed using the remove() method. The ListIterator interface is used to iterate the objects in the list one after the other.

Working with the Vector Class

The Vector class is similar to the ArrayList and LinkedList classes. However, the methods of the Vector class are synchronized, which means that only one thread at a given time can access it in a multithreaded environment. On the other hand, the methods of the ArrayList and LinkedList classes are not synchronized.

The Vector class provides various constructors that can be used to create an instance. The following table lists the commonly used constructors of the Vector class.

Constructor	***Description***
Vector()	*Creates an empty instance of Vector.*
Vector(Collection<? extends E> c>	*Creates an instance with the specified collection.*
Vector(int initialCapacity)	*Creates an empty instance of Vector with the specified initial capacity. The initial capacity is the size of Vector that grows or shrinks automatically, whenever an object is added or removed to/ from Vector.*

The Constructors of the Vector Class

The following table lists the commonly used methods of the Vector class.

Method	***Description***
boolean add(E e)	*Is used to add the specified object at the end of Vector.*
void clear()	*Is used to remove all the objects from Vector.*
E get(int index)	*Is used to retrieve the object at the specified index from Vector.*

Method	***Description***
`E remove(int index)`	*Is used to remove the object at the specified index from* `Vector`.
`boolean remove(Object o)`	*Is used to remove the first occurrence of the specified object from* `Vector`.
`int size()`	*Is used to get the number of objects in* `Vector`.

The Methods of the Vector Class

Consider the following code to add and remove the objects to/ from a `Vector` collection:

```
import java.util.ListIterator;
import java.util.Vector;
public class VectorDemo
{
    public static void main(String[] args)
    {
        Vector<Double> obj = new Vector<Double>();
        Double dobj1 = new Double(77.5);
        Double dobj2 = new Double(68.1);
        Double dobj3 = new Double(52.8);
        Double dobj4 = new Double(40.2);

        System.out.println("Size of Vector is: " + obj.size());

        obj.add(dobj1);
        obj.add(dobj2);
        obj.add(dobj3);
        obj.add(dobj4);
        obj.add(dobj1);

        System.out.println("\nVector after adding the objects: " + obj);
        System.out.println("Size of Vector after adding objects: " +
obj.size());

        obj.remove(dobj1);
        obj.remove(dobj3);

        System.out.println("\nVector after removing the objects: " + obj);
        System.out.println("Size of Vector after removing objects: " +
obj.size());

        System.out.println("\nThe final Vector: ");
        ListIterator i = obj.listIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Size of Vector is: 0

Vector after adding the objects: [77.5, 68.1, 52.8, 40.2, 77.5]
Size of Vector after adding objects: 5

Vector after removing the objects: [68.1, 40.2, 77.5]
Size of Vector after removing objects: 3

The final Vector:
68.1
40.2
```

In the preceding code, the `Vector` object contains a collection of the `Double` objects. The `add()` method of the `Vector` class is used to add the `Double` objects. Further, two objects are removed using the `remove()` method. The `ListIterator` interface is used to iterate the objects one after the other.

Just a minute:

Which one of the following classes enables you to create a collection of unique objects?

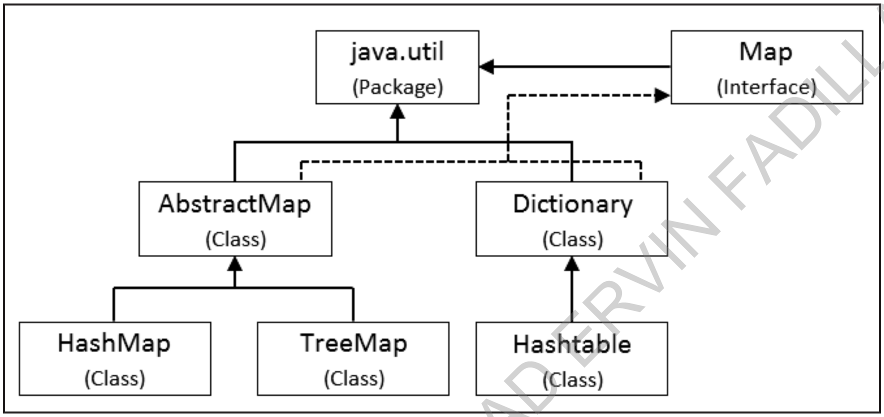
1. *`List`*
2. *`Vector`*
3. *`TreeSet`*
4. *`ArrayList`*

Answer:

3. *`TreeSet`*

Using the Map Interface

The Map interface enables you to create a collection with key-value pair objects. In this collection, both key and value are objects. You need to use the key object to access the corresponding value object. The Map interface allows duplicate value objects but the key object must be unique. The following figure shows the class hierarchy of the java.util package that implements the Map interface.



The Class Hierarchy of the java.util Package

The preceding figure shows that the java.util package contains the AbstractMap, Dictionary, HashMap, TreeMap, and Hashtable classes and the Map interface. The HashMap and TreeMap classes extend the AbstractMap class, which implements the Map interface. In addition, the Hashtable class extends the Dictionary class that also implements the Map interface.

Working with the HashMap Class

The HashMap class enables you to create a collection in which a value is accessible using the key. HashMap stores objects in an unordered form. The HashMap class allows one null value for a key and any number of null values for values.

The HashMap class provides various constructors that can be used to create an instance. The following table lists the commonly used constructors of the HashMap class.

Constructor	***Description***
HashMap()	*Creates an empty instance of HashMap.*
HashMap(int initialCapacity)	*Creates an empty instance of HashMap with the specified initial capacity. The initial capacity is the size of HashMap that grows or shrinks automatically, whenever objects are added or removed to/ from HashMap.*

Constructor	***Description***
`HashMap(Map<? extends K, ? extends V> m)`	*Creates an instance with the same mapping of specified `Map`.*

The Constructors of the HashMap Class

The following table lists the commonly used methods of the `HashMap` class.

Method	***Description***
`void clear()`	*Is used to remove all the mapping from `HashMap`.*
`V get(Object key)`	*Is used to get the value object of the specified key object from `HashMap`.*
`V put(K key, V value)`	*Is used to add the specified key object with the specified value object in `HashMap`.*
`V remove(Object o)`	*Is used to remove the mapping for the specified key object from `HashMap`.*
`int size()`	*Is used to get the number of key-value mappings in `HashMap`.*

The Methods of the HashMap Class

Note

`K` and `V` represent key and value, respectively. Both, `K` and `V`, are used for the types of classes.

Consider the following code to add and remove the objects to/ from a `HashMap` collection:

```
import java.util.HashMap;

public class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> obj = new HashMap<String, Integer>();
        Integer iobj1 = new Integer(11);
        Integer iobj2 = new Integer(22);
        Integer iobj3 = new Integer(33);
        Integer iobj4 = new Integer(44);

        System.out.println("Size of HashMap is: " + obj.size());
        obj.put("L1", iobj1);
        obj.put("L2", iobj2);
        obj.put("L3", iobj3);
        obj.put("L4", iobj4);
        obj.put("L5", iobj1);

        System.out.println("\nHashMap after adding the objects: " + obj);
        System.out.println("Size of HashMap after adding objects: " +
obj.size());

        obj.remove("L3");
        obj.remove("L5");

        System.out.println("\nHashMap after removing the objects: " + obj);
        System.out.println("Size of the HashMap after removing objects: " +
obj.size());
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Size of HashMap is: 0

HashMap after adding the objects: {L1=11, L2=22, L3=33, L4=44, L5=11}
Size of HashMap after adding objects: 5

HashMap after removing the objects: {L1=11, L2=22, L4=44}
Size of the HashMap after removing objects: 3
```

In the preceding code, the `HashMap` object contains a collection of key-value pair objects. The `put()` method of the `HashMap` class is used to add the key-value pair objects. Further, the mapping of two objects are removed using the `remove()` method.

Working with the TreeMap Class

The `TreeMap` class enables you to create a collection of objects in the sorted order with unique keys. The sorted collection improves the retrieval process of objects.

The `TreeMap` class provides various constructors that can be used to create an instance. The following table lists commonly used constructors of the `TreeMap` class.

Constructor	***Description***
`TreeMap()`	*Creates an instance with empty `TreeMap`.*
`TreeMap(Map<? extends K, ? extends V> m)`	*Creates an instance with the same mapping of specified `Map`.*

The Constructors of the TreeMap Class

The following table lists the commonly used methods of the `TreeMap` class.

Method	***Description***
`void clear()`	*Is used to remove all the mapping from `TreeMap`.*
`V get(Object key)`	*Is used to get the value object of the specified key object from `TreeMap`.*
`V put(K key, V value)`	*Is used to add the specified key object with the specified value object in `TreeMap`.*
`V remove(Object o)`	*Is used to remove the mapping for the specified key object from `TreeMap`.*
`int size()`	*Is used to get the number of key-value mapping in `TreeMap`.*

The Methods of the TreeMap Class

Consider the following code to add and remove the objects to/ from a `TreeMap` collection:

```
import java.util.TreeMap;

public class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap<Integer, String> obj = new TreeMap<Integer, String>();
        String sobj1 = new String("Value 1");
        String sobj2 = new String("Value 2");
        String sobj3 = new String("Value 3");
        String sobj4 = new String("Value 4");

        System.out.println("Size of TreeMap is: " + obj.size());
        obj.put(101, sobj1);
        obj.put(102, sobj2);
        obj.put(103, sobj3);
        obj.put(104, sobj4);
        obj.put(105, sobj1);
        System.out.println("\nTreeMap after adding the objects: " + obj);
        System.out.println("Size of TreeMap after adding objects: " +
obj.size());
        obj.remove(103);
        obj.remove(105);
        System.out.println("\nTreeMap after removing the objects: " + obj);
        System.out.println("Size of the TreeMap after removing objects: " +
obj.size());
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Size of TreeMap is: 0

TreeMap after adding the objects: {101=Value 1, 102=Value 2, 103=Value 3,
104=Value 4, 105=Value 1}
Size of TreeMap after adding objects: 5

TreeMap after removing the objects: {101=Value 1, 102=Value 2, 104=Value 4}
Size of the TreeMap after removing objects: 3
```

In the preceding code, the `TreeMap` object contains a collection of key-value pair objects. The `put()` method of the `TreeMap` class is used to add the key-value pair objects. Further, mapping of two objects are removed using the `remove()` method.

Working with the Hashtable Class

The `Hashtable` class enables you to create an unordered collection of objects, which cannot contain the `null` objects. The methods of the `Hashtable` class are synchronized, which means that only one thread at a given time can access it in a multithreaded environment. On the other hand, the methods of the `HashMap` and `TreeMap` classes are not synchronized.

The `Hashtable` class provides various constructors that can be used to create an instance. The following table lists commonly used constructors of the `Hashtable` class.

Constructor	***Description***
`Hashtable()`	*Creates an instance with empty `Hashtable`.*
`Hashtable(int initialCapacity)`	*Creates an instance with empty `Hashtable` with the specified initial capacity. The initial capacity is the size of `Hashtable` that grows or shrinks automatically, whenever objects are added or removed to/ from `Hashtable`.*
`Hashtable(Map<? extends K, ? extends V> t)`	*Creates an instance with the same mapping of specified `Map`.*

The Constructors of the Hashtable Class

The following table lists the commonly used methods of the `Hashtable` class.

Method	***Description***
`void clear()`	*Is used to remove all objects from `Hashtable`.*
`V get(Object key)`	*Is used to get the value object of the specified key object from `Hashtable` or `null` if it does not contain any value for the specified key object.*
`V put(K key, V value)`	*Is used to add the specified key object with the specified value object in `Hashtable`.*
`V remove(Object o)`	*Is used to remove the mapping for the specified key object from `Hashtable`.*
`int size()`	*Is used to get the number of keys in `Hashtable`.*

The Methods of the Hashtable Class

Consider the following code to add and remove the objects to/ from a `Hashtable` collection:

```
import java.util.Hashtable;

public class HashtableDemo
{
    public static void main(String[] args)
    {
        Hashtable<Integer, Double> obj = new Hashtable<Integer, Double>();
        Double dobj1 = new Double(55.6);
        Double dobj2 = new Double(34.6);
        Double dobj3 = new Double(98.6);
        Double dobj4 = new Double(12.5);

        System.out.println("Size of Hashtable is: " + obj.size());

        obj.put(55, dobj1);
        obj.put(60, dobj2);
        obj.put(65, dobj3);
        obj.put(70, dobj4);
        obj.put(75, dobj3);

        System.out.println("\nHashtable after adding the objects: " + obj);
        System.out.println("Size of Hashtable after adding objects: " +
obj.size());

        obj.remove(65);
        obj.remove(75);

        System.out.println("\nHashtable after removing the objects: " + obj);
        System.out.println("Size of Hashtable after removing objects: " +
obj.size());
```

```
        }
    }
```

Once the preceding code is executed, the following output is displayed:

```
Size of Hashtable is: 0

Hashtable after adding the objects: {65=98.6, 75=98.6, 60=34.6, 70=12.5,
55=55.6}
Size of Hashtable after adding objects: 5

Hashtable after removing the objects: {60=34.6, 70=12.5, 55=55.6}
Size of Hashtable after removing objects: 3
```

In the preceding code, the `Hashtable` object contains a collection of key-value pair objects. The `put()` method of the `Hashtable` class is used to add the key-value pair objects. Further, mapping of two objects are removed using the `remove()` method.

Just a minute:

Which one of the following classes contains the synchronized methods?

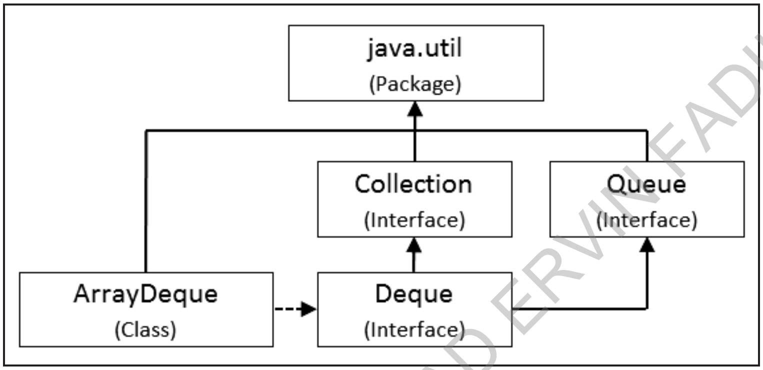
1. *`Hashtable`*
2. *`HashSet`*
3. *`TreeMap`*
4. *`ArrayList`*

Answer:

1. *`Hashtable`*

Using the Deque Interface

The Deque interface allows you to create a collection of objects in which the objects can be inserted or deleted from both ends. Deque is the short form for the double-ended queue. The Deque interface extends the Queue interface that provides the collection in which objects are inserted at the end and deleted from the beginning. The Queue interface mainly works on the First In First Out (FIFO) algorithm. The following figure shows the class hierarchy of the java.util package that implements the Deque interface.



The Class Hierarchy of the java.util Package

The preceding figure shows that the java.util package contains the ArrayDeque class and the Collection, Queue, and Deque interfaces. The ArrayDeque class implements the Deque interface that is extended from the Collection and Queue interfaces.

Working with the ArrayDeque Class

The ArrayDeque class provides an implementation of the Deque interface that enables you to create a resizable array, which allows insertion and deletion from both the ends. However, the methods of the ArrayDeque class are not synchronized, which means more than one thread at a given time can access it in a multithreaded environment.

The ArrayDeque class provides various constructors that can be used to create an instance. The following table lists the commonly used constructors of the ArrayDeque class.

Constructor	***Description***
ArrayDeque()	*Creates an empty instance of ArrayDeque.*
ArrayDeque(Collection<? extends E> c)	*Creates an instance with the specified collection.*
ArrayDeque(int numElement)	*Creates an empty instance of ArrayDeque with the initial capacity that can hold the specified number of elements.*

The Constructors of the ArrayDeque Class

The following table lists the commonly used methods of the `ArrayDeque` class.

Method	***Description***
`boolean add(E e)`	*Is used to add the specified object at the end of `ArrayDeque`.*
`void addFirst(E e)`	*Is used to add the specified object as the first element in `ArrayDeque`.*
`void addLast(E e)`	*Is used to add the specified object as the last element in `ArrayDeque`.*
`void clear()`	*Is used to remove all the objects from `ArrayDeque`.*
`E getFirst()`	*Is used to retrieve the first object of `ArrayDeque`.*
`E getLast()`	*Is used to retrieve the last element of `ArrayDeque`.*
`E removeFirst()`	*Is used to retrieve and remove the first object from `ArrayDeque`.*
`E removeLast()`	*Is used to retrieve and remove the last object from `ArrayDeque`.*
`int size()`	*Is used to retrieve the number of objects in `ArrayDeque`.*

The Methods of the ArrayDeque Class

Consider the following code to add and remove the objects to/ from an `ArrayDeque` collection:

```
import java.util.ArrayDeque;

public class ArrayDequeDemo
{
    public static void main(String[] args)
    {
        ArrayDeque<Double> obj = new ArrayDeque<Double>();
        Double dobj1 = new Double(7.5);
        Double dobj2 = new Double(8.5);
        Double dobj3 = new Double(9.5);
        Double dobj4 = new Double(10.5);

        System.out.println("Size of ArrayDeque is: " + obj.size());

        obj.add(dobj1);
        obj.add(dobj2);
        System.out.println("\nArrayDeque after adding the objects: " + obj);
        System.out.println("Size of ArrayDeque after adding objects: " +
obj.size());

        obj.addFirst(dobj3);
        obj.addLast(dobj4);

        System.out.println("\nArrayDeque after adding the objects at first
and last: " + obj);
        System.out.println("Size of ArrayDeque after adding objects at first
and last: " + obj.size());

        obj.removeFirst();

        System.out.println("\nArrayDeque after removing the first object: " +
obj);
        System.out.println("Size of ArrayDeque after removing the first
objects: " + obj.size());
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Size of ArrayDeque is: 0

ArrayDeque after adding the objects: [7.5, 8.5]
Size of ArrayDeque after adding objects: 2

ArrayDeque after adding the objects at first and last: [9.5, 7.5, 8.5, 10.5]
Size of ArrayDeque after adding objects at first and last: 4

ArrayDeque after removing the first object: [7.5, 8.5, 10.5]
Size of ArrayDeque after removing the first objects: 3
```

In the preceding code, the `ArrayDeque` object contains a collection of the `Double` objects. The `add()` method of the `ArrayDeque` class is used to add the `Double` objects to the `Deque` object. Two objects are added, one at the beginning and the other at the end of the queue. Further, an object is removed using the `remove()` method, and the `removeFirst()` method is used to remove the element stored at the first position.

Implementing Sorting

Consider a scenario of Address Book, which contains the contact details, such as name, contact number, and address. Now, you need to create a list of contacts living in Asia. In addition, you need to ensure that the list should have the flexibility to add or remove any number of contacts. For this, you have decided to create a collection of contacts. Thereafter, you need to sort the list according to the contact name. For this, Java provides the following interfaces to sort the objects in a collection:

- The `Comparable` interface
- The `Comparator` interface

The `Comparable` and `Comparator` interfaces provide their own methods to achieve the sorting on the collection. However, the `Comparable` and `Comparator` interfaces are used with the `sort()` method of the `Collections` and `Arrays` classes to sort the collection of objects.

Using the Comparable Interface

The `Comparable` interface is defined in the `java.lang` package and is used to sort and compare a collection of objects. This interface provides the `compareTo()` method that compares the references of the objects. The `compareTo()` method returns a value of the `int` type with the following characteristics:

- **Negative**: If the current object is less than the object being compared
- **Zero**: If the current object is equal to the object being compared
- **Positive**: If the current object is greater than the object being compared

Consider the example where you want to store the name and marks of students. Thereafter, you want to sort the details of each student based on marks. For this, you have decided to create the `Student` class, as shown in the following code:

```
public class Student implements Comparable
{
    private String name;
    private int marks;

    public Student(String nm, int mk)
    {
        this.name = nm;
        this.marks = mk;
    }

    public int getMarks()
    {
        return marks;
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    public int compareTo(Object obj)
    {
        Student s = (Student) obj;

        if (this.marks > s.getMarks())
            return 1;
        else if (this.marks < s.getMarks())
            return -1;
        else
            return 0;

    }

    public String toString()
    {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Name: "+ name +"\n");
        buffer.append("Marks: " + marks + "\n");
        return buffer.toString();
    }

}
```

In the preceding code, the `Student` class is created, which overrides the `compareTo()` method of the `Comparable` interface. The `compareTo()` method compares the `Student` object on the basis of marks. In addition, the `toString()` method of the `Object` class is overridden so that the name and marks of the student are displayed when the object of the `Student` class is used inside the `println()` method.

Thereafter, you can use the following code to create the `ComparableDemo` class:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.ListIterator;

public class ComparableDemo
{
    public static void main(String[] args)
    {
        Student s1 = new Student("Alex", 88);
        Student s2 = new Student("Bob", 90);
        Student s3 = new Student("Joe", 78);

        ArrayList<Student> obj = new ArrayList<>();
        obj.add(s1);
        obj.add(s2);
        obj.add(s3);

        System.out.println("Student details are:");

        ListIterator li = obj.listIterator();
        while(li.hasNext())
        {
            System.out.println(li.next());
        }
```

```
            Collections.sort(obj);

            System.out.println("Student details after sorting are:");

            ListIterator li2 = obj.listIterator();
            while(li2.hasNext())
            {
                System.out.println(li2.next());
            }
        }
    }
```

Once the preceding code is executed, the following output is displayed:

```
Student details after sorting are:
Name: Joe
Marks: 78

Name: Alex
Marks: 88

Name: Bob
Marks: 90
```

In the preceding code, the `ComparableDemo` class is created that creates an array list of three `Student` objects by adding the objects through the `add()` method. In addition, the array list is sorted using the `sort()` method of the `Collections` class.

Using the Comparator Interface

You have learned that the `Comparable` interface is used to sort and compare a collection of objects. However, by using this interface, you cannot define the logic for multiple sorting. For example, in the previous scenario, you want to create a sorting logic based on the student marks and names. For this, you can use the `Comparator` interface that is defined in the `java.util` package. This interface provides the `compare()` method that is used to compare two objects in a collection. The `compare()` method returns the value of `int` type with the following characteristics:

- **Negative**: If the current object is less than the object being compared
- **Zero**: If the current object is equal to the object being compared
- **Positive**: If the current object is greater than the object being compared

Consider the following code to define the `Student` class:

```
public class Student
{
    private String name;
    private int marks;

    public Student(String nm, int mk)
    {
        this.name = nm;
        this.marks = mk;
    }
    public int getMarks()
    {
        return marks;
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString()
    {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Name: "+ name +"\n");
        buffer.append("Marks: " + marks + "\n");
        return buffer.toString();
    }

}
```

In the preceding code, `name` and `marks` are the attributes of the `Student` class. In addition, the setter and getter methods for the attributes are defined. Further, the `toString()` method of the `Object` class is overridden.

Consider the following code that defines a sorting logic based on the marks of a student:

```
import java.util.Comparator;
public class MarkCompare implements Comparator{
    public int compare(Object a, Object b)
    {
        Student x = (Student) a;
        Student y = (Student) b;

        if (x.getMarks() > y.getMarks())
            return 1;
        else if (x.getMarks() < y.getMarks())
            return -1;
        else
            return 0;
    }
}
```

In the preceding code, the `compare()` method of the `Comparator` interface is overridden and the logic to sort the student objects based on the student marks is defined.

Consider the following code to define a sorting logic based on the names of students:

```
import java.util.Comparator;
public class NameCompare implements Comparator{
    public int compare(Object a, Object b)
    {
        Student x = (Student) a;
        Student y = (Student) b;
        return x.getName().compareTo(y.getName());
    }
}
```

In the preceding code, the `compare()` method of the `Comparator` interface is overridden and the logic to sort student objects based on the student names is defined. Because the name of the students are the `String` objects, the `compare()` method of the `String` class is used to implement the sorting logic.

Thereafter, you can use the following code to create a `ComparatorDemo` class:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.ListIterator;

public class ComparatorDemo
{
    public static void main(String[] args)
    {
        Student s1 = new Student("Alex", 88);
        Student s2 = new Student("Bob", 90);
        Student s3 = new Student("Joe", 78);

        ArrayList<Student> obj = new ArrayList<>();
        obj.add(s1);
        obj.add(s2);
        obj.add(s3);

        System.out.println("Student details are:");

        ListIterator li = obj.listIterator();
        while(li.hasNext())
        {
            System.out.println(li.next());
        }

        Collections.sort(obj, new MarkCompare());

        System.out.println("Mark wise sort:");

        ListIterator li2 = obj.listIterator();
        while(li2.hasNext())
        {
            System.out.println(li2.next());
        }
        Collections.sort(obj, new NameCompare());

        System.out.println("Name wise sort:");

        ListIterator li3 = obj.listIterator();
        while(li3.hasNext())
        {
            System.out.println(li3.next());
        }
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Student details are:
Name: Alex
Marks: 88

Name: Bob
Marks: 90

Name: Joe
Marks: 78

Mark wise sort:
Name: Joe
Marks: 78

Name: Alex
Marks: 88

Name: Bob
Marks: 90

Name wise sort:
Name: Alex
Marks: 88

Name: Bob
Marks: 90

Name: Joe
Marks: 78
```

In the preceding code, the `ComparatorDemo` class is created that creates an array list of three `Student` objects by adding the objects through the `add()` method. In addition, the array list is sorted using the `sort()` method of the `Collections` class.

Activity 4.1: Working with Collections

Practice Questions

1. Which one of the following classes implements the `List` interface?

   a. `HashSet`
   b. `TreeSet`
   c. `Hashtable`
   d. `Vector`

2. Which one of the following classes is inherited from the `Dictionary` class?

   a. `HashSet`
   b. `TreeSet`
   c. `Hashtable`
   d. `HashMap`

3. Which one of the following interfaces enables you to traverse the list of objects?

   a. `Iterator`
   b. `List`
   c. `Collection`
   d. `Comparable`

4. Which one of the following methods is used to add the objects in `TreeMap`?

   a. `put()`
   b. `add()`
   c. `addObject()`
   d. `putObject()`

5. Which one of the following interfaces is used to create a collection of unique objects?

   a. `Set`
   b. `Map`
   c. `Deque`
   d. `List`

Summary

In this chapter, you learned that:

- The Set interface is used to create a collection of unique objects.
- The package, java.util, provides the Iterator interface to traverse through the set collection.
- The HashSet class provides the implementation of the Set interface that enables you to create a set in which insertion is faster because it does not sort the elements.
- The TreeSet class provides the implementation of the Set interface that enables you to create a sorted set of objects.
- The List interface is used to create an ordered collection of objects, which can contain duplicate objects.
- The package, java.util, provides the ListIterator interface to traverse through the list collection.
- The ArrayList class provides the implementation of the List interface. The ArrayList class enables you to create a resizable array.
- The LinkedList class provides the implementation of the List interface. The LinkedList class enables you to create a doubly-linked list.
- LinkedList can be traversed in the forward or backward direction.
- The Vector class is similar to the ArrayList and LinkedList classes.
- The methods of the Vector class are synchronized, which means that only one thread at a given time can access it in a multithreaded environment.
- The methods of the ArrayList and LinkedList classes are not synchronized.
- The Map interface enables you to create a collection with key-value pair objects.
- The Map interface allows duplicate value objects but the key object must be unique.
- The HashMap class enables you to create a collection in which a value is accessible using the key. HashMap stores objects in an unordered form.
- The TreeMap class enables you to create a collection of objects in the sorted order with unique keys.
- The Hashtable class enables you to create an unordered collection of objects, which cannot contain the null objects.
- The methods of the Hashtable class are synchronized, which means that only one thread at a given time can access it in a multithreaded environment.
- The methods of the HashMap and TreeMap classes are not synchronized.
- The Deque interface allows you to create a collection of objects in which the objects can be inserted or deleted from both ends.
- The ArrayDeque class provides an implementation of the Deque interface that enables you to create a resizable array, which allows insertion and deletion from both the ends.
- The methods of the ArrayDeque class are not synchronized, which means more than one thread at a given time can access it in a multithreaded environment.
- Java provides the following interfaces to sort the objects in a collection:
  - The Comparable interface
  - The Comparator interface

- The Comparable interface is defined in the java.lang package and is used to sort and compare a collection of objects. This interface provides the compareTo() method that compares the references of the objects.
- The Comparator interface is defined in the java.util package. This interface provides the compare() method that is used to compare two objects in a collection.

Chapter 5

Working with Threads

Most computer games use graphics and sounds. The graphics, score, and audio effects are presented simultaneously. Imagine a computer game where you first see the screen changing, then you see your scores that are being updated, and finally, you hear the sound effects. In order to appreciate and enjoy a game, all these elements need to be processed simultaneously. In other words, the program should be able to efficiently perform three tasks simultaneously. In Java, you can achieve the preceding functionality by making use of *thread*s. The game can be divided into three subunits that can be executed in parallel by using individual threads.

This chapter discusses how to create multithreaded applications in Java. In addition, it discusses about the thread priorities.

Objectives

In this chapter, you will learn to:

- Use threads in Java
- Create threads

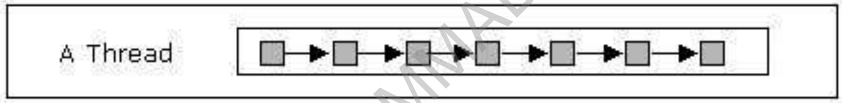
Using Threads in Java

A thread can be defined as a single sequential flow of control within a program. It is a sequence of instructions that is executed to define a unique flow of control. For example, a Central Processing Unit (CPU) performs various processes simultaneously, such as writing and printing a document, installing software, and displaying the date and time. All these processes are handled by separate threads. An application comprises one or more processes that can be performed simultaneously. Further, each process is broken into small chunks known as tasks. Each thread is assigned a unique task to perform, independently. For example, a text editor is an application that allows you to perform two tasks, such as write to a document and print a document, simultaneously.

Further, you can implement multithreading in an application to achieve an efficient performance and utilize the CPU processing capabilities. In order to implement multithreading, Java provides the `Thread` class. In order to work with threads, you need to first identify their life cycle.

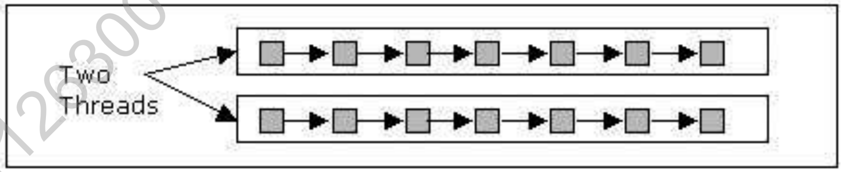
The Basic Concept of Multithreading

A single-threaded application can perform only one task at a time. The following figure shows a single-threaded application.



A Single-threaded Application

A single-threaded application needs to wait for one task to complete before another can start. As a result, the efficiency of the application is hampered. On the other hand, you can apply multithreading when there is a requirement to enable an application to perform several tasks simultaneously. Multithreading helps to perform multiple tasks simultaneously, which saves time and improves the execution speed and efficiency of the application. Every application has at least one thread and you can create more threads, if required. The following figure shows a multi-threaded application.



A Multi-threaded Application

The microprocessor allocates the memory to the processes that are executed by an application and each process occupies its own address space or memory. However, all the threads in a process occupy the same address space.

Advantages and Disadvantages of Multithreading

Most of the computers today utilize software and hardware, such as operating system and multiple processors that support multithreading. Therefore, to achieve efficiency and ensure an optimized performance, most of the applications utilize multhreading. However, there are certain advantages and disadvantages of using multithreading that can be identified. The various advantages of multithreading are:

- **Improved performance**: Provides improvement in the performance of the processor by the simultaneous execution of computation and Input/Output (I/O) operations.
- **Minimized system resource usage**: Minimizes the use of system resources by using threads, which share the same address space and belong to the same process.
- **Program structure simplification**: Simplifies the structure of complex applications, such as multimedia applications. Subprograms can be written for each activity that makes a complex program easy to design and code.

The various disadvantages of multithreading are:

- **Race condition**: When two or more threads simultaneously access the same variable and at least one thread tries to write a value to the variable, it is called the race condition. This condition is caused when synchronization between the two threads is not implemented. For example, in a word processor program, there are two threads, ThreadA and ThreadB. ThreadA wants to read a file, and at the same time, ThreadB wants to write to a file. If ThreadA reads the value before ThreadB performs a write operation, ThreadA will fail to receive the updated value from the file. Such a condition is known as a race condition.
- **Deadlock condition**: The *deadlock* condition arises in a computer system when two threads wait for each other to complete the operations before performing the individual action. As a result, the two threads are locked and the program fails. For example, consider two threads, ThreadA and ThreadB. ThreadA is waiting for a lock to be released by ThreadB, and ThreadB is also waiting for the lock to be released by ThreadA to complete its transaction. This situation is a deadlock condition.
- **Lock starvation**: *Lock starvation* occurs when the execution of a thread is postponed because of its low priority. JRE executes threads based on their priority because the CPU can execute only one thread at a time. The thread with a higher priority is executed before the thread with a lower priority, and therefore, there is a possibility that the execution of the thread that has a lower priority is always postponed.

The Thread Class

The `java.lang.Thread` class is used to construct and access the individual threads in a multi-threaded application. You can create a multi-threaded application by using the `Thread` class or the `Runnable` interface.

The `Thread` class contains various methods that can be used to work with a thread, such as setting and checking the properties of a thread, causing a thread to wait, and being interrupted or destroyed. You can make the classes run in separate threads by extending the `Thread` class. Some of the methods defined in the `Thread` class are:

- `int getPriority()`: Returns the priority of a thread.
- `boolean isAlive()`: Determines whether a thread is running.

- `static void sleep(long milliseconds)`: Causes the thread execution to pause for a period of time as specified by the value of `milliseconds`.
- `String getName()`: Returns the name of a thread.
- `void start()`: Starts a thread execution by calling the `run()` method.
- `static Thread currentThread()`: Returns a reference to the thread object that is currently executing.
- `boolean isAlive()`: Is used to check the existence of a thread. This method returns `true` if the thread is running, and returns `false` if the thread is new or gets terminated.
- `public final void join()`: Allows a thread to wait until the thread, on which it is called, terminates.
- `void interrupt()`: Is used to interrupt the execution of a thread.
- `static boolean interrupted()`: Is used to determine if the current thread has been interrupted by another thread.

You can use the following code to understand the usage of the methods in the `Thread` class:

```
class MainThreadDemo
{
  public static void main(String args[])
  {
    Thread t= Thread.currentThread();
    System.out.println(" The current thread: " + t);
    t.setName("MainThread");
    System.out.println(" The current thread after name change : "
    + t);
    System.out.println(" The current Thread is going to sleep for
    10 seconds");
    try
    {
   t.sleep(10000);
     }
    catch(InterruptedException e)
    {
     System.out.println("Main thread interrupted");
    }
   System.out.println(" After 10 seconds..........the current
      Thread is exiting now.");
    }
  }
```

Once the preceding code is executed, the following output is displayed:

```
The current thread: Thread[main,5,main]
The current thread after name change : Thread[MainThread,5,main]
The current Thread is going to sleep for 10 seconds
After 10 seconds..........the current Thread is exiting now.
```

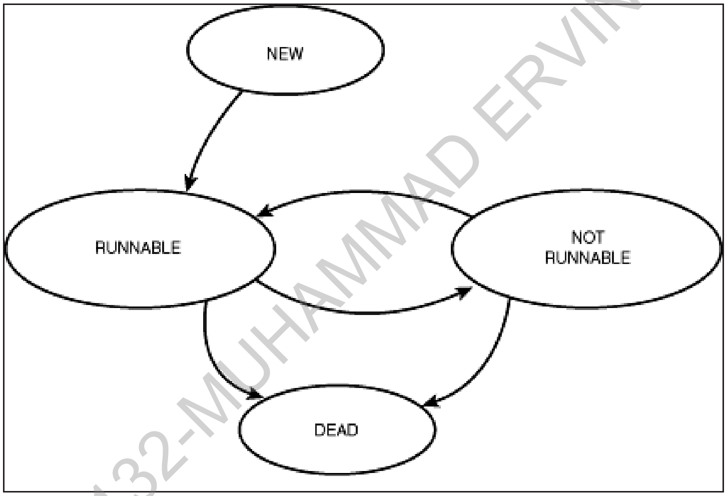
In the preceding code, the first thread to be executed is the main thread. It is created automatically when a Java program is executed. In the main program, a reference to the current thread is obtained by calling the `currentThread()` method and its reference is stored in the reference variable, `t`. The `setName()` method is called to set the name of the thread, and the information about the thread is displayed. The main thread is then made to sleep for 10 seconds by using the `sleep()` method, and after 10 seconds, the thread terminates. The `sleep()` method throws the `InterruptedException` exception whenever another thread interrupts the sleeping thread.

The Life Cycle of a Thread

The various states in the life cycle of a thread are:

- New
- Runnable
- Not Runnable
- Terminated or Dead

The following figure shows the life cycle of a thread.



The Life Cycle of a Thread

The New Thread State

When an instance of the `Thread` class is created, the thread enters the new thread state. The following code snippet shows how to instantiate the `Thread` class:

```
Thread newThread = new Thread(this, "threadName");
```

In the preceding code snippet, a new thread is created. This new thread is an empty object of the `Thread` class, and no system resources, such as memory, are allocated to it. You have to invoke the `start()` method to start the thread. The following code snippet shows how to start a thread:

```
newThread.start();
```

The Runnable Thread State

When the `start()` method of a thread is invoked, the thread enters the runnable state. The `start()` method allocates the system resources to the thread, schedules the thread, and passes the control to its `run()` method.

A processor cannot execute more than one thread at a time. Therefore, the processor maintains a thread queue. When a thread starts, it is queued for the processor time, and it waits for its turn to execute. As a result, the state of the thread is said to be runnable and not running.

The Not Runnable Thread State

A thread is in the not runnable state if it is:

- **Sleeping**: A thread is put into the sleeping mode by calling the `sleep()` method. A sleeping thread continues with its execution after the specified time of sleep has elapsed.
- **Waiting**: A thread can be made to wait for some specified condition that needs to be satisfied by calling the `wait()` method of the `Object` class. The thread can be issued a notification of the status of the condition by invoking the `notify()` or `notifyAll()` method of the `Object` class.
- **Being blocked by another thread**: If blocked by an I/O operation, a thread enters the not runnable state.

The Dead Thread State

A thread enters the dead state once the statements in the `run()` method have executed. Assigning a `null` value to a thread object also changes the state of the thread to dead. The `isAlive()` method of the `Thread` class is used to determine whether a thread is alive or not. You cannot restart a dead thread. In addition, a thread can be killed by stopping its execution with the help of the `stop()` method.

Creating Threads

Consider a scenario of a jumbled up word game where you need to implement a timer in the game. The timer will be used to provide a duration of 60 seconds within which a player has to find the correct word corresponding to a jumbled up word. Further, the timer will be displayed on the game screen when a player is playing the game. When the timer reaches 0, the "Time Up!!!" message will be displayed to the player and the game will stop.

In order to implement the preceding functionalities, you can create a thread. This can be done by instantiating an object of the `Thread` class. You can create a thread in a Java application by using the following approaches:

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

Further, you can also create multiple threads in an application by using the preceding approaches and set their priorities to determine their execution sequence in the JRE.

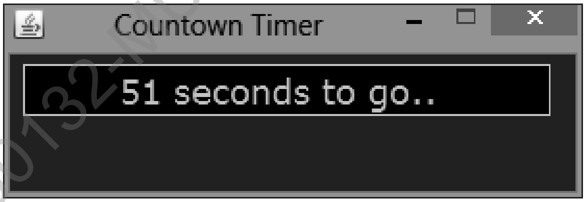
Creating a Thread by Extending the Thread Class

You can create threads by extending the `Thread` class. You need to specify the task to be performed by a thread in the overridden `run()` method. You can use the following code to create a thread to implement the timer by extending the `Thread` class:

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.*;

public class CountdownTimer extends Thread {
    JTextField tf;
    JLabel l;
    JFrame fr;
    public void run()
    {
         buildGUI();
    }

    void display() {

        for (int i = 60; i >= 0; i--)
        {
            try {
                Thread.sleep(1000);
                String s = Integer.toString(i);

                tf.setText("        "+ s + " seconds to go..");
            } catch (Exception e) {
                System.out.println(e);
            }

        }
        JOptionPane.showMessageDialog(fr, "Time Up !!!!");
```

```
            tf.setText("");
            tf.setEnabled(false);
        }

        public void buildGUI() {
            fr = new JFrame("Countown Timer");
            JPanel p = new JPanel();
            l = new JLabel("");
            tf = new JTextField(15);
            tf.setEnabled(false);
            Font f = new Font("Verdana", 0, 18);
            tf.setFont(f);
            tf.setBackground(Color.BLACK);
            p.setBackground(Color.blue);
            fr.add(p);
            p.add(tf);
            p.add(l);
            fr.setVisible(true);
            fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            fr.setSize(300, 100);
            fr.setResizable(false);
            display();
        }

        public static void main(String args[]) {
            CountdownTimer obj = new CountdownTimer();
            obj.start();
        }
    }
```

After executing the preceding code, the following output is displayed:



51 seconds to go..

The Output

Note
The time in the preceding figure may vary.

In the preceding code, the `CountdownTimer` class extends the `Thread` class. A new instance of the `CountdownTimer` class is created in the `main()` method. The `start()` method begins the execution of a thread by calling the `run()` method that provides the implementation of the timer.

Creating a Thread by Implementing the Runnable Interface

Consider a scenario where you need to implement threads in a Java application and also extend another class to use its functionality, such as `JFrame`. Because Java does not support multiple inheritance, it provides the `Runnable` interface to solve this problem. The `Runnable` interface only consists of the `run()` method, which is executed when the thread is started.

When a program needs to inherit from a class other than the `Thread` class, you need to implement the `Runnable` interface to implement threads. The signature of the `run()` method is:

```
public void run()
```

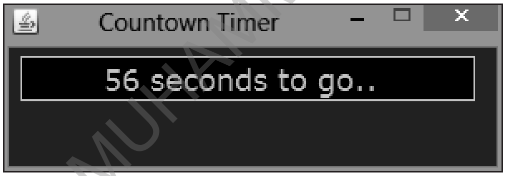
The `run()` method contains the code that the new thread will execute. When the `run()` method is called, another thread starts executing concurrently with the main thread in the program.

The class that implements the `Runnable` interface is used to create an instance of the `Thread` class. The new thread object starts by calling the `start()` method. You can use the following code to create a thread by implementing the `Runnable` interface for implementing the timer functionality:

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.*;

public class CountdownTimer implements Runnable {
    JTextField tf;
    JLabel l;
    JFrame fr;
    public void run()
    {
         buildGUI();
    }

    void display() {

        for (int i = 60; i >= 0; i--)
        {
            try {
                Thread.sleep(1000);
                String s = Integer.toString(i);

                tf.setText("        "+ s + " seconds to go..");
            } catch (Exception e) {
                System.out.println(e);
            }

        }
        JOptionPane.showMessageDialog(fr, "Time Up !!!!");
        tf.setText("");
        tf.setEnabled(false);
    }

    public void buildGUI() {
        fr = new JFrame("Countown Timer");
```

```
            JPanel p = new JPanel();
            l = new JLabel("");
            tf = new JTextField(15);
            tf.setEnabled(false);
            Font f = new Font("Verdana", 0, 18);
            tf.setFont(f);
            tf.setBackground(Color.BLACK);
            p.setBackground(Color.blue);
            fr.add(p);
            p.add(tf);
            p.add(l);
            fr.setVisible(true);
            fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            fr.setSize(300, 100);
            fr.setResizable(false);
            display();
        }

        public static void main(String args[]) {
            CountdownTimer obj = new CountdownTimer();
            Thread CountdownThread = new Thread(obj);       //Initialize thread
            CountdownThread.start();
        }
    }
```

Once the preceding code is executed, the following output is displayed:



56 seconds to go..

The Output

Note
The time in the preceding figure may vary.

In the preceding code, the `CountdownTimer` class implements the `Runnable` interface. In the `main()` method, an object of the `CountdownTimer` class is created and the object is passed as a parameter to the constructor of the `Thread` class. The preceding object represents a runnable task. The `start()` method begins the execution of a thread by calling the `run()` method, which provides the implementation of the timer.

Creating Multiple Threads

Suppose you need to develop a car racing game where multiple objects, such as obstacles, need to move across the frame, simultaneously. For this, you need to create multiple threads where each thread represents an obstacle. You can create multiple threads in a program either by implementing the `Runnable` interface or extending the `Thread` class.

For the preceding scenario, you can use the following code to create multiple threads by using the `Thread` class:

```
import java.awt.Color;
import java.util.Random;
import javax.swing.*;
public class Race extends Thread{

    String ThreadName;
    JLabel l;
    JPanel l1,l2,l3;
    JFrame fr;

    public Race()
    {
    buildGUI();
    }
    public Race(String s)
    {
    super(s);
    }

    public void run()
    {

         if(Thread.currentThread().getName().equals("ObstacleA"))
         {
         runObstacleA();
         }
         if(Thread.currentThread().getName().equals("ObstacleB"))
         {
         runObstacleB();
         }
         if(Thread.currentThread().getName().equals("ObstacleC"))
         {
         runObstacleC();
         }

    }

    public void runObstacleA()
    {
         Random ran = new Random();
         int s = ran.nextInt(1000);
         for(int i=-10;i<400;i++)
         {

           l1.setBounds(i, s, 20, 20);
         try {
         Thread.sleep(5);

         }
```

```
            catch(Exception e)
            {
            System.out.println(e);
            }
         }
       runObstacleC();
       }


       public void runObstacleB()
       {
        Random ran = new Random();
            int r = ran.nextInt(180);

            for(int i=-10;i<400;i++)
            {
              l2.setBounds(i, r, 20, 20);
            try {
            Thread.sleep(11);

            }
            catch(Exception e)
            {
            System.out.println(e);
            }
         }
       runObstacleA();
       }

       public void runObstacleC()
       {
         Random ran = new Random();
            int m = ran.nextInt(10);

            for(int i=-10;i<400;i++)
            {
              l3.setBounds(i, m, 20, 20);
            try {
            Thread.sleep(10);

            }
            catch(Exception e)
            {
            System.out.println(e);
            }
         }
       runObstacleB();
       }
       public void buildGUI()
       {
           fr = new JFrame("Moving objects");
           fr.setVisible(true);
           fr.setSize(400,200);
           fr.setLayout(null);
           l = new JLabel("");
           l.setBounds(10,10,400,20);
           fr.add(l);

           l1 = new JPanel();
           l1.setSize(20, 20);
           l1.setBackground(Color.red);
           l1.setBounds(10,40,20,20);
           fr.add(l1);
           l2 = new JPanel();
           l2.setSize(20, 20);
           l2.setBackground(Color.blue);
           l2.setBounds(10,80,20,20);
           fr.add(l2);

           l3 = new JPanel();
           l3.setSize(20, 20);
           l3.setBackground(Color.black);
           l3.setBounds(10,120,20,20);
           fr.add(l3);

       }

       public static void main(String args[])
   {
                          Race obj = new Race();
                          Thread Obstacle1 = new Thread(obj);
                        Thread Obstacle2 = new Thread(obj);
                  Thread Obstacle3 = new Thread(obj);

                  Obstacle1.setName("ObstacleA");
                  Obstacle2.setName("ObstacleB");
                  Obstacle3.setName("ObstacleC");
                  Obstacle1.start();
                  Obstacle2.start();
                  Obstacle3.start();
       }
   }
```

In the preceding code, three thread references, `Obstacle1`, `Obstacle2`, and `Obstacle3` are created and the object `obj` of the `Race` class is passed as parameter. Then, `Obstacle1`, `Obstacle2`, and `Obstacle3` are given the names, `ObstacleA`, `ObstacleB`, and `ObstacleC`, respectively. These obstacles will appear at random positions.

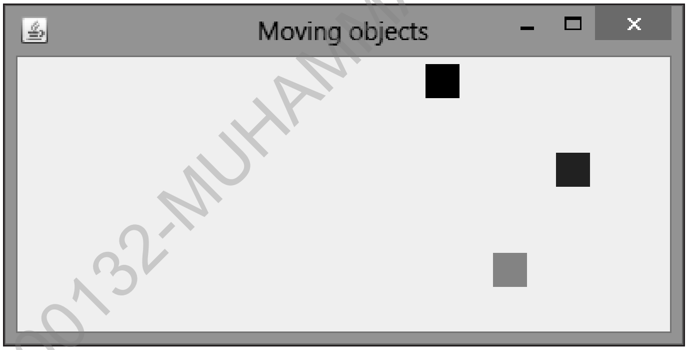
You can also use the following code to create multiple threads by using the `Runnable` interface:

```
import java.awt.Color;
import java.util.Random;
import javax.swing.*;

public class Race implements Runnable{

    String ThreadName;
    JLabel l;
    JPanel l1,l2,l3;
```

```
JFrame fr;

public Race()
{
buildGUI();
}

public void run()
{
     if(Thread.currentThread().getName().equals("ObstacleA"))
     {
     runObstacleA();
     }
     if(Thread.currentThread().getName().equals("ObstacleB"))
     {
     runObstacleB();
     }
     if(Thread.currentThread().getName().equals("ObstacleC"))
     {
     runObstacleC();
     }

}

public void runObstacleA()
{
     Random ran = new Random();
     int s = ran.nextInt(1000);
     for(int i=-10;i<400;i++)
     {

       l1.setBounds(i, s, 20, 20);
     try {
     Thread.sleep(5);

     }
     catch(Exception e)
     {
     System.out.println(e);
     }
  }
runObstacleC();
}

public void runObstacleB()
{
 Random ran = new Random();
     int r = ran.nextInt(180);

     for(int i=-10;i<400;i++)
     {
       l2.setBounds(i, r, 20, 20);
     try {
     Thread.sleep(11);
     }
     catch(Exception e)
     {
     System.out.println(e);
     }
  }
runObstacleA();
}

public void runObstacleC()
{
  Random ran = new Random();
     int m = ran.nextInt(10);

     for(int i=-10;i<400;i++)
     {
       l3.setBounds(i, m, 20, 20);
     try {
     Thread.sleep(10);

     }
     catch(Exception e)
     {
     System.out.println(e);
     }
  }
runObstacleB();
}

public void buildGUI()
{
    fr = new JFrame("Moving objects");
    fr.setVisible(true);
    fr.setSize(400,200);
    fr.setLayout(null);

    l = new JLabel("");
    l.setBounds(10,10,400,20);
    fr.add(l);

    l1 = new JPanel();
    l1.setSize(20, 20);
    l1.setBackground(Color.red);
    l1.setBounds(10,40,20,20);
    fr.add(l1);

    l2 = new JPanel();
    l2.setSize(20, 20);
    l2.setBackground(Color.blue);
    l2.setBounds(10,80,20,20);
    fr.add(l2);

    l3 = new JPanel();
    l3.setSize(20, 20);
    l3.setBackground(Color.black);
```

```
            l3.setBounds(10,120,20,20);
            fr.add(l3);

        }

        public static void main(String args[])
    {
              Race obj = new Race();
              Thread Obstacle1 = new Thread(obj);
              Thread Obstacle2 = new Thread(obj);
              Thread Obstacle3 = new Thread(obj);

    Obstacle1.setName("ObstacleA");
    Obstacle2.setName("ObstacleB");
    Obstacle3.setName("ObstacleC");
    Obstacle1.start();
    Obstacle2.start();
    Obstacle3.start();
        }
    }
```

In the preceding code, three threads, ObstacleA, ObstacleB, and ObstacleC, share the memory of the CPU. These obstacles will appear at random positions on the frame and move across the frame. Once the preceding code is executed, the following output is displayed:



The Output

Sometimes, in a multithreaded program, there may be a requirement to ensure that the execution of the main thread finishes after all the other threads have executed. For this, they will need to know whether a thread has finished its execution or else they can join the execution of a thread with the main() method. This can be determined by using the isAlive() method and the join() method.

Using the isAlive() Method

The isAlive() method is used to check the existence of a thread. You can use the isAlive() method to find the status of a thread. The signature of the isAlive() method is:

```
    boolean isAlive()
```

In the preceding signature, the `isAlive()` method returns `true` if the thread is running, and returns `false` if the thread is new or gets terminated. The thread can be in a runnable or not runnable state if the `isAlive()` method returns `true`. The following code uses the `isAlive()` method:

```
class NewThreadClass implements Runnable
{
   Thread t;
   NewThreadClass()
   {
          t = new Thread(this,"ChildThread" );
          System.out.println("Thread created: " + t);
          t.start();
   }
   public void run()
   {
          try
          {
                 for(int i=1;i<=5;i++)
                 {
                        System.out.println(t + "loop :" + i);
                        Thread.sleep(100);
                 }
          }
          catch( InterruptedException obj)
          {
                 System.out.println("Thread :" + t + "interrupted");
          }
   }
}
class IsAliveDemo
{
    public static void main(String args[])
    {
          NewThreadClass obj = new NewThreadClass();
          System.out.println(obj.t  + "is alive ? : " +
            obj.t.isAlive());
   try
   {
                 for(int i=1;i<=5;i++)
                 {
                 System.out.println("Main Thread loop:" + i);
                 Thread.sleep(200);
                 }
   }
   catch(InterruptedException e)
          {
          System.out.println("Main thread is interrupted");}
          System.out.println(obj.t  + "is alive ? : " +
            obj.t.isAlive());
          System.out.println("Main Thread is exiting");
   }
}
```

Once the preceding code is executed, the following output is displayed:

```
Thread created: Thread[ChildThread,5,main]
Thread[ChildThread,5,main]is alive ? : true
Main Thread loop:1
Thread[ChildThread,5,main]loop :1
Thread[ChildThread,5,main]loop :2
Main Thread loop:2
Thread[ChildThread,5,main]loop :3
Thread[ChildThread,5,main]loop :4
Thread[ChildThread,5,main]loop :5
Main Thread loop:3
Main Thread loop:4
Main Thread loop:5
Thread[ChildThread,5,]is alive ? : false
Main Thread is exiting
```

In the preceding code, the `isAlive()` method is called on the thread to check whether the thread is alive or dead.

Using the join() Method

The `join()` method allows a thread to wait until the thread, on which it is called, terminates. In addition, the `join()` method enables you to specify the maximum amount of time that you need to wait for the specified thread to terminate. The signature of the `join()` method is:

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

In the preceding signature, if another thread interrupts, the `join()` method throws the `InterruptedException` exception. The execution of a thread can be interrupted by another thread by using the `interrupt()` method of the `Thread` class. To determine if the current thread has been interrupted by another thread, the `Thread` class provides the `interrupted()` method.

Further, the `join()` method returns the control to the calling method when the specified thread dies. The following code uses the `join()` method:

```
class ChildThread implements Runnable
{
  Thread t;
  ChildThread()
  {
   t = new Thread(this,"ChildThread" );
   System.out.println("Thread created: " + t);
   t.start();
  }
  public void run()
  {
   try
   {
                for(int i=1;i<=5;i++)
          {
                System.out.println(t + "loop :" + i);
                Thread.sleep(500);
          }
```

```
    }
    catch( InterruptedException obj)
    {
          System.out.println("Thread :" + t + "interrupted");
    }
  }
}
public class JoinDemo
{
   public static void main(String args[])
    {
        ChildThread obj = new ChildThread();
               System.out.println(obj.t  + "is alive ? : " +
          obj.t.isAlive());
               try
         {
          System.out.println("Main thread waiting for child thread to
finish");
                 obj.t.join();
          }
          catch(InterruptedException e)
          {
            System.out.println("Main thread is interrupted");
}
            System.out.println(obj.t  + "is alive ? : " +
            obj.t.isAlive());
            System.out.println("Main Thread is exiting");
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Thread created: Thread[ChildThread,5,main]
Thread[ChildThread,5,main]is alive ? : true
Main thread waiting for child thread to finish
Thread[ChildThread,5,main]loop :1
Thread[ChildThread,5,main]loop :2
Thread[ChildThread,5,main]loop :3
Thread[ChildThread,5,main]loop :4
Thread[ChildThread,5,main]loop :5
Thread[ChildThread,5,]is alive ? : false
```

In the preceding code, the `join()` method is called on the child thread and the main thread waits for the child thread to finish its execution.

Identifying the Thread Priorities

JRE executes threads based on their priority. Since a CPU is able to execute only one thread at a time, the threads that are ready for execution are scheduled in a queue to be executed by the processor. The threads are scheduled by JRE by using fixed-priority scheduling. Each thread has a priority that affects its position in the thread queue of the processor. A thread with a higher priority is scheduled for execution before the threads with a lower priority.

Defining Thread Priority

Thread priorities are integers in the range of 1 to 10 that specify the priority of one thread with respect to the priority of another thread. The execution of multiple threads on a single CPU in a specified order is called scheduling.

If the processor encounters another thread with a higher priority, the current thread is pushed back and the thread with the higher priority is executed. The next thread of a lower priority starts executing if a higher priority thread stops or becomes not runnable. A thread is pushed back in the queue by another thread if it is waiting for an I/O operation. A thread can also be pushed back in the queue when the time for which the `sleep()` method was called on another higher priority thread gets over.

Setting the Thread Priority

Once a thread is created, its priority is set by using the `setPriority()` method of the `Thread` class. The signature of the `setPriority()` method is:

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

In the preceding signature, the `newPriority` parameter specifies the new priority setting for a thread. The priority levels should be within the range of two constants, `MIN_PRIORITY` and `MAX_PRIORITY`. The `MIN_PRIORITY` and `MAX_PRIORITY` constants have the values 1 and 10, respectively. The `setPriority()` method throws the `IllegalArgumentException` exception if the priority level is less than `MIN_PRIORITY` and greater than `MAX_PRIORITY`.

The thread can be set to a default priority by specifying the `NORM_PRIORITY` constant in the `setPriority()` method. `MAX_PRIORITY` is the highest priority that a thread can have. `MIN_PRIORITY` is the lowest priority a thread can have, and `NORM_PRIORITY` is the default priority set for a thread.

You can use the following code to set the priorities of various threads:

```
class ChildThread implements Runnable
{
    Thread t;
    ChildThread(int p)
    {
      t = new Thread(this,"ChildThread" );
      t.setPriority(p);
      System.out.println("Thread created: " + t);
    }

   public void run()
  {
    try
          {
                 for(int i=1;i<=5;i++)
          {
                       System.out.println(t + "loop :" + i);
                       Thread.sleep(500);
                       }
          }
          catch( InterruptedException obj)
          {
          System.out.println("Thread :" + t + "interrupted");}
          }
}
class PriorityDemo
{
  public static void main(String args[])
   {
   ChildThread obj1 = new ChildThread(Thread.NORM_PRIORITY - 2);
   ChildThread obj2 = new ChildThread(Thread.NORM_PRIORITY + 2);
          ChildThread obj3 = new ChildThread(Thread.NORM_PRIORITY + 3);

   //Starting the threads with different priority
   obj1.t.start();
          obj2.t.start();
          obj3.t.start();
          try
   {
     System.out.println("Main thread waiting for child thread to
       finish");
       obj1.t.join();
       obj2.t.join();
     obj3.t.join();
      }

   catch(InterruptedException e)
          {
          System.out.println("Main thread is interrupted");}
          System.out.println(obj1.t  + "is alive ? : " +
            obj1.t.isAlive());
                 System.out.println(obj2.t  + "is alive ? : " +
            obj2.t.isAlive());
          System.out.println(obj3.t  + "is alive ? : " +
            obj3.t.isAlive());
                 System.out.println("Main Thread is exiting");
     }
    }
```

Once the preceding code is executed, the following output is displayed:

```
Thread created: Thread[ChildThread,3,main]
Thread created: Thread[ChildThread,7,main]
Thread created: Thread[ChildThread,8,main]
Main thread waiting for child thread to finish
Thread[ChildThread,3,main]loop :1
Thread[ChildThread,7,main]loop :1
Thread[ChildThread,8,main]loop :1
Thread[ChildThread,7,main]loop :2
Thread[ChildThread,8,main]loop :2
Thread[ChildThread,3,main]loop :2
Thread[ChildThread,7,main]loop :3
Thread[ChildThread,8,main]loop :3
Thread[ChildThread,3,main]loop :3
Thread[ChildThread,7,main]loop :4
Thread[ChildThread,8,main]loop :4
Thread[ChildThread,3,main]loop :4
Thread[ChildThread,7,main]loop :5
```

```
Thread[ChildThread,8,main]loop :5
Thread[ChildThread,3,main]loop :5
Thread[ChildThread,3,]is alive ? : false
Thread[ChildThread,7,]is alive ? : false
Thread[ChildThread,8,]is alive ? : false
Main Thread is exiting
```

Activity 5.1: Implementing Threads in Java

Practice Questions

1. Identify the minimum value to set the thread priority.

   a. 5
   b. 10
   c. 1
   d. 15

2. Identify the method that is used to check the existence of a thread.

   a. `isAlive()`
   b. `sleep()`
   c. `syncrhonize()`
   d. `alive()`

3. Identify the method that returns the name of a thread.

   a. `name()`
   b. `getName()`
   c. `getThreadName()`
   d. `currentThreadName()`

4. Identify the method that is used to pause the execution of a thread for a period of time.

   a. `isAlive()`
   b. `sleep()`
   c. `notify()`
   d. `stop()`

5. Consider the following statements:

   Statement A: You cannot restart a dead thread.

   Statement B: When a thread is blocked by an I/O operation, it enters the not runnable state.

   Which one of the following options is correct based on the preceding statements?

   a. Both the statements are false.
   b. Both the statements are true.
   c. Statement A is true and Statement B is false.
   d. Statement B is true and Statement A is false.

Summary

In this chapter, you learned that:

- A thread is defined as the path of execution of a program. It is a sequence of instructions that is executed to define a unique flow of control.
- A program that creates two or more threads is called a multi-threaded program.
- The various advantages of multithreading are:
  - Improved performance
  - Minimized system resource usage
  - Program structure simplification
- The various disadvantages of multithreading are:
  - Race condition
  - Deadlock
  - Lock starvation
- The `java.lang.Thread` class is used to construct and access the individual threads in a multi-threaded application.
- The various states in the life cycle of a thread are:
  - New
  - Runnable
  - Not Runnable
  - Terminated or Dead
- You can create a thread in the following ways:
  - By extending the `Thread` class
  - By implementing the `Runnable` interface
- Thread priorities are the integers in the range of 1 to 10 that specify the priority of one thread with respect to the priority of another thread.
- You can set the thread priority after it is created by using the `setPriority()` method declared in the `Thread` class.

Implementing Thread Synchronization and Concurrency

You have learned how to create a multi-threaded Java application. In addition, you have learned that in a multi-threaded application, a resource can be accessed by multiple threads at a time. However, there may be situations when you need that at a time, only one thread should access the resource. For this, Java provides thread synchronization.

Moreover, to improve the performance of a multi-threaded application, Java supports concurrency. To implement concurrency, a program is split into tasks, which can be executed in parallel.

This chapter focuses on implementing thread synchronization and concurrency.

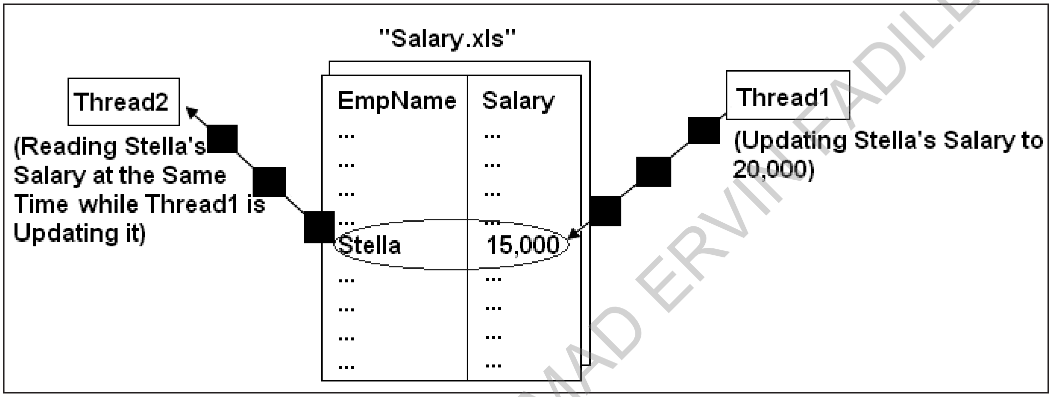
Objectives

In this chapter, you will learn to:

- Implement thread synchronization
- Implement concurrency

Implementing Thread Synchronization

In a multithreaded environment, when two threads need to share data, you must try to coordinate their activities to ensure that one thread does not change the data used by the other thread to avoid errors. For example, if you have two threads, one that reads a salary record from a file and another that tries to update the salary record at the same time, there is a possibility that the thread that is reading the salary record does not get the updated value. The following figure shows two threads accessing the data of a file at the same time.


"Salary.xls"
Thread2
(Reading Stella's Salary at the Same Time while Thread1 is Updating it)
EmpName
Salary
Stella
15,000
Thread1
(Updating Stella's Salary to 20,000)

The Two Threads Accessing Data at the Same Time

In the preceding figure, Thread1 is updating the salary of Stella to $20,000. At the same time, another thread, Thread2, is reading the salary of Stella before it is being updated. Therefore, the information read by Thread2 is faulty.

To avoid such situations, Java enables you to coordinate and manage the actions of multiple threads at a given time by using synchronized threads and implementing communication between threads.

Synchronizing Threads

Consider the scenario of a banking application that allows you to access a bank account and perform operations, such as deposit and withdraw amount. In addition, these operations can be performed by using mediums, such as the Internet and Automated Teller Machine (ATM). In such a scenario, there is a possibility that multiple operations are performed on an account simultaneously. This can lead to loss of data or inconsistency in data. Therefore, to avoid such issues, you need to implement a mechanism using which only a single operation can be performed on an account at a time. For this, you can apply thread synchronization.

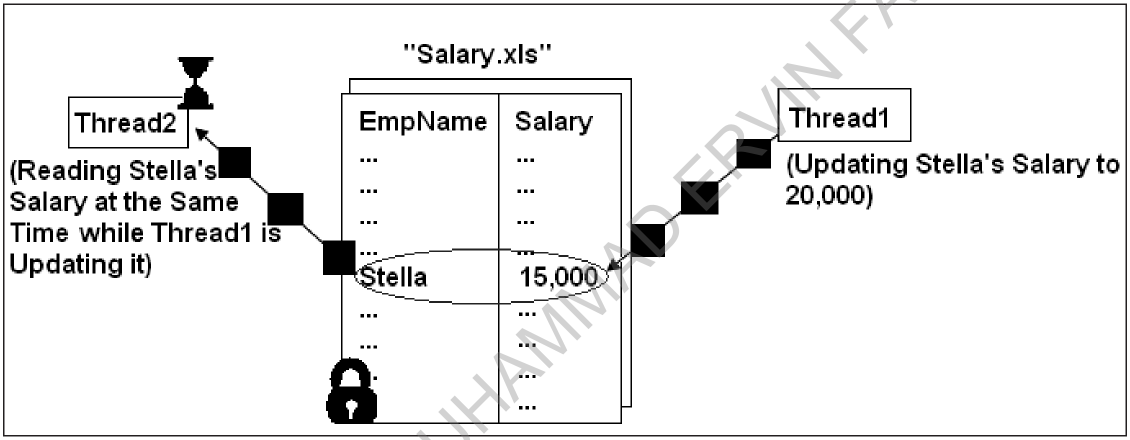
The synchronization of threads ensures that if two or more threads need to access a shared resource, then that resource is used by only one thread at a time. You can implement thread synchronization by using the following approaches:

- By using the synchronized methods
- By using the synchronized statement

The Synchronized Methods

You can synchronize a method by using the `synchronized` keyword. Synchronization is based on the concept of a monitor, which is an object used as a lock. All the Java objects have a monitor and only one thread can hold the monitor of an object at a given time. To enter an object's monitor, you need to call a synchronized method. The monitor controls the way in which synchronized methods access an object or class. When a thread acquires a lock, it is said to have entered the monitor. The monitor ensures that only one thread has access to the resources at any given time.

When a thread is within a synchronized method, all the other threads that try to call the method at the same time have to wait. During the execution of a synchronized method, the object is locked so that no other synchronized method can be invoked. The monitor is automatically released when the method completes its execution. The following figure shows an example of thread synchronization.


"Salary.xls"
Thread2
(Reading Stella's Salary at the Same Time while Thread1 is Updating it)
EmpName
Salary
Stella
15,000
Thread1
(Updating Stella's Salary to 20,000)

An Example of Thread Synchronization

In the preceding figure, Thread1 has a lock on the Salary.xls file, while updating the salary of Stella in the file. Thread2 wants to read the salary of Stella from the Salary.xls file and is waiting for Thread1 to release the lock on the file. While a thread is sleeping or waiting, it temporarily releases the lock it holds.

The following code snippet demonstrates how to implement synchronization for the preceding scenario by using a synchronized method:

```
class EmployeeDetails
{
// Details of the employee
}
class EmployeeSalaryTransaction
{
public synchronized int processSalary()
{
// Code to process the salary
}
}
```

In the preceding code snippet, the `processSalary()` method is a synchronized method. This method cannot be accessed by multiple threads at a time.

The Synchronized Statement

Synchronization among threads can also be achieved by using synchronized statements. The synchronized statement is used where the synchronization methods are not used in a class and you do not have access to the source code. Further, when a method contains only a small section of critical code for which synchronization is required, then you can use the synchronized statement instead of synchronizing the entire method. You can synchronize the access to an object of this class by placing the calls to the methods defined by it inside a `synchronized` block. The following code snippet shows how to use the `synchronized` block:

```
synchronized(obj)
{
/* statements to be synchronized*/
}
```

In the preceding code snippet, `obj` is a reference of the object to be synchronized.

The following code snippet demonstrates how to implement synchronization, in case of the banking application, by using the synchronized statement:

```
class EmployeeDetails
{
    // Details of the employee
}
class EmployeeSalaryTransaction
{
public int processSalary()
    {
            synchronized(obj)
        {
            // Code to process the salary
        }
    }
}
```

In the preceding code snippet, `obj` refers to an object of the `EmployeeDetails` class. The code to process the salary is specified within a `synchronized` block to ensure that it cannot be accessed by multiple threads at a time.

Implementing Inter-threaded Communication

Threads can communicate with each other. Multithreading eliminates the concept of polling by a mechanism known as interprocess communication. For example, there are two threads, thread1 and thread2. The data produced by thread1 is consumed by thread2. thread1 is the producer, and thread2 is the consumer. The consumer has to repeatedly check if the producer has produced data. This is a waste of CPU time as the consumer occupies the CPU time to check whether the producer is ready with data. A thread cannot proceed until the other thread has completed its task.

A thread may notify another thread that the task has been completed. This communication between threads is known as inter-threaded communication. The various methods used in inter-threaded communication are:

- `wait()`: Informs the current thread to leave its control over the monitor and wait until another thread calls the `notify()` method.
- `notify()`: Wakes up a single thread that is waiting for the monitor of the object being executed. If multiple threads are waiting, one of them is chosen randomly.
- `notifyAll()`: Wakes up all the threads that are waiting for the monitor of the object.

The following code shows the producer-consumer problem, but without inter-threaded communication:

```
class SynchronizedMethods
{
  int d;
    synchronized void getData()
    {
        System.out.println("Got data:" + d);

    }
    synchronized void putData(int d)
    {
        this.d = d;
        System.out.println("Put data:" + d);
   }
}
class Producer extends Thread
{
   SynchronizedMethods t;
   public Producer(SynchronizedMethods t)
   {
   this.t = t;
   }
   public void run()
   {
   int data =700;
   while(true)
        {
     System.out.println("Put Called by producer");
     t.putData(data++);
        }
   }
}
class Consumer extends Thread
{
   SynchronizedMethods t;
   public Consumer(SynchronizedMethods t)
   {
   this.t = t;
   }
   public void run()
   {
      while(true)
      {
   System.out.println("Get Called by consumer");
```

```
            t.getData();
         }
      }
   }
   public class ProducerConsumer
   {
      public static void main(String args[])
      {
         SynchronizedMethods obj1 = new SynchronizedMethods();
         Producer p = new Producer(obj1);
         Consumer c = new Consumer(obj1);
         p.start();
         c.start();
      }
   }
```

Once executed, the preceding code runs infinitely, as shown in the following output:

```
   .
   .
   .
   .
   Put Called by producer
   Put data:29670
   Put Called by producer
   Put data:29671
   Put Called by producer
   Put data:29672
   Put Called by producer
   Put data:29673
   Put Called by producer
   Put data:29674
   Put Called by producer
   Put data:29675
   Put Called by producer
   Put Called by producer
   .
   .
   .
   .
```

To avoid such errors, `wait()`, `notify()`, and `notifyAll()` methods are used. All the three methods can be called from within the synchronized methods.

The following code shows the inter-thread communication by using the `wait()` and `notify()` methods:

```
   class SynchronizedMethods
   {
   int d;
   boolean flag = false;
   synchronized int getData()
   {
   if(!flag)
   {
   try
   {
```

```
wait();
}

catch(InterruptedException e)
{
System.out.println(" Exception caught");
}
}
System.out.println("Got data:" + d);
flag=false;
notify();
return d;
}
synchronized void putData(int d)
{

if(flag)
{

try
{
wait();
}
catch(InterruptedException e)
{
System.out.println(" Exception caught");
}
}
this.d = d;
flag=true;
System.out.println("Put data with value:" + d);
notify();

}
}

class Producer implements Runnable
{
SynchronizedMethods t;
public Producer(SynchronizedMethods t)
{
this.t = t;
new Thread(this,"Producer").start();
System.out.println("Put Called by producer");
}
public void run()
{

int data =0;
while(true)
{
data=data+1;
t.putData(data);
}
}
```

```
}
class Consumer implements Runnable
{
SynchronizedMethods t;
public Consumer(SynchronizedMethods t)
{
this.t = t;
new Thread(this,"Consumer").start();

System.out.println("Get Called by consumer");
}
public void run()
{

while(true)
{
t.getData();
}
}
}
public class InterThreadComm
{
public static void main(String args[])
{
SynchronizedMethods obj1 = new SynchronizedMethods();
Producer p = new Producer(obj1);
Consumer c = new Consumer(obj1);

}
}
```

Once the preceding code is executed, the following output is displayed:

```
Put Called by producer
Put data with value:1
Get Called by consumer
Got data:1
Put data with value:2
Got data:2
Put data with value:3
Got data:3
Put data with value:4
Got data:4
```

Note

The preceding output may vary.

Just a minute:

The ______ method is used to wake up all the threads that are waiting for the monitor of the object.

Answer:

notifyAll()

Activity 6.1: Implementing Synchronization

Implementing Concurrency

In a multithreaded environment, it is important to implement synchronization to eliminate the interference of threads and coordinate their activities. However, synchronization does not ensure that a multithreaded application is utilizing the system resources efficiently. There may be a situation in an application where multiple threads are waiting to obtain a lock on a synchronized method or code that is already locked. As a result, the efficiency of the application is hampered. Therefore, it is important to improve the efficiency of multithreaded applications and ensure that they are using the hardware resources of the system, such as the processor, to their potential. For this, the Java programming language supports concurrency.

Concurrency enables Java programmers to improve the efficiency and resource utilization of their applications by creating concurrent programs. Most computers today have more than a single processor to provide a faster performance. Therefore, by applying concurrency in Java applications, the program tasks can be managed to execute simultaneously by the systems that have multiple processors. You can achieve concurrency by dividing a program into multiple threads, where each thread has a separate task to perform. This minimizes the wastage of processing time and improves the execution speed of an application.

To implement concurrency, Java provides the `java.util.concurrent` package. This package contains the necessary classes and interfaces that can be used to implement the different ways to achieve concurrency and high performance in Java applications.

Implementing Atomic Variables and Locks

The `java.util.concurrent` package has the following sub packages to support concurrent programming:

- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`

The java.util.concurrent.atomic Package

Consider a scenario of a multithreaded Java application that deals with updating and reading data, such as integer values stored in variables. You want to ensure that any change made to the data is updated immediately so that all the threads that access the variables get the latest value. You want to ensure that all the updates to variables are successfully accomplished and any change made is not lost due to problems, such as power outages. Moreover, you want that even in case of failures, such as starvation, the data is successfully updated. For this, you can use atomic variables. *Atomic variables* are the single variables on which operations, such as increment and decrement, are done in a single step. Such operations are known as atomic operations.

An *atomic operation* is an operation that is performed as a single unit. In addition, an atomic operation cannot be stopped in between by any other operations. Therefore, when multiple threads are working with an atomic variable to perform an operation, then that operation either happens completely in a single step or does not happen at all. The use of atomic variables improves performance as compared to synchronization and helps in avoiding inconsistency in data. The `java.util.concurrent.atomic` package provides many classes and methods that can be used to create atomic variables for performing atomic operations.

The following table lists the various classes of the `java.util.concurrent.atomic` package.

Class	***Description***
`AtomicBoolean`	*A `boolean` value that may be updated atomically.*
`AtomicInteger`	*An `int` value that may be updated atomically.*
`AtomicIntegerArray`	*An `int` array in which elements may be updated atomically.*
`AtomicLong`	*A `long` value that may be updated atomically.*
`AtomicLongArray`	*A `long` array in which elements may be updated atomically.*

The Classes of the java.util.concurrent.atomic Package

Consider the following code that demonstrates an atomic operation of an atomic integer variable:

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicVariableDemo
{
    public static void main(String a[])
    {
        AtomicInteger value = new AtomicInteger(5);
        System.out.println("Intial value: " + value.get());
        value.getAndIncrement();
        System.out.println("After incrementing, the value: " + value.get());
        value.set(40);
        System.out.println("After setting, the value: " + value.get());
        value.getAndDecrement();
        System.out.println("After decrementing, the value: " + value.get());
        value.compareAndSet(39, 45);
        System.out.println("After updating, the value: " + value.get());
    }
}
```

In the preceding code, `AtomicInteger` named `value` is created with the value, `5`. The `get()` method of the `AtomicInteger` class is used to get the current value of the `value` variable. The `getAndIncrement()` method is then used to atomically increment the current value of the reference variable by `1`. Further, the `getAndDecrement()` method is used to atomically decrement the current value by `1`. The `compareAndSet()` method sets the value of the value variable to `45` if its value is `39`.

The java.util.concurrent.locks Package

You have learned how to implement synchronization in a multithreaded application. However, in an application that uses synchronized threads, if a thread is waiting to acquire a lock on an object and the lock is not available to be acquired, the thread may keep waiting to acquire the lock. This can result in a deadlock situation. Moreover, when there are multiple threads that are waiting to acquire a lock, it is not possible to determine the thread that will acquire the lock once it is available. There is a possibility that the thread that has been waiting for the longest period of time never acquires the lock. In addition, when using synchronization, if an exception is raised while a thread is executing the critical section, then the lock may never be released. This can degrade the performance of an application. In order to overcome the preceding drawbacks, Java provides the `ReentrantLock` class in the `java.util.concurrent.locks` package. This class implements the `Lock` interface. A reentrant lock is used for mutual locking amongst threads. A reentrant lock can be acquired by the same thread any number of times.

In addition, this class provides useful methods that can be used to perform locking in a flexible and more productive manner by using a lock. A lock is a tool that is used for controlling access to a shared resource.

The following table lists the constructors of the `ReentrantLock` class.

Constructor	***Description***
`ReentrantLock()`	*Is used to create an instance of the `ReentrantLock` class.*
`ReentrantLock(boolean fairness)`	*Is used to create an instance of the `ReentrantLock` class and set the fairness. The fairness option when set to true ensures that the lock is given to the longest waiting thread in the queue of waiting threads.*

The Constructors of the ReentrantLock Class

The following table lists some of the methods of the `ReentrantLock` class.

Method	***Description***
`void lock()`	*This method is used to acquire a lock. Further, this method cannot be interrupted.*
`boolean tryLock()`	*This method is used to acquire a lock only if it is available. If the lock is available, it returns with a `true` value, else it returns with a `false` value.*
`boolean tryLock(long time, TimeUnit unit)`	*This method is used to acquire a lock only if it is available in the specified time period. If the lock is available, it returns with a `true` value, else it returns with a `false` value.*
`void unlock()`	*This method is used to release a lock.*
`boolean isFair()`	*This method is used to return `true` if the fairness is set to `true`.*
`boolean isLocked()`	*This method is used to query if the lock is currently held by any thread.*

The Methods of the ReentrantLock Class

Consider the following code that shows how to implement a reentrant lock:

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockImplementation implements Runnable {
    final Lock lock = new ReentrantLock();
    String name;
    Thread t;

    public void createThreads(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("Creating New Thread: " + t.getName());
        t.start();
    }

    public static void main(String args[]) {
        ReentrantLockImplementation obj = new ReentrantLockImplementation();
        obj.createThreads("Thread 1");
        obj.createThreads("Thread 2");
        obj.createThreads("Thread 3");
    }

    public void run() {
        do {
            try {
                if (lock.tryLock(400, TimeUnit.MILLISECONDS)) {
                    try {
                     System.out.println(Thread.currentThread().getName()+"
acquired the lock. "
                                );
                        Thread.sleep(1000);
                    } finally {
                        lock.unlock();
                        System.out.println(Thread.currentThread().getName()+"
released the lock. "
                                );
                    }
                    break;
                }
                else
                {
                    System.out.println( Thread.currentThread().getName() + "
could not acquire the lock. Need to try for the lock again.");
                }
            }
            catch (InterruptedException e)
            {
                System.out.println(e);
            }
        } while (true);
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Creating New Thread: Thread 1
Creating New Thread: Thread 2
Creating New Thread: Thread 3
Thread 1 acquired the lock.
Thread 3 could not acquire the lock. Need to try for the lock again.
Thread 2 could not acquire the lock. Need to try for the lock again.
Thread 3 could not acquire the lock. Need to try for the lock again.
Thread 2 could not acquire the lock. Need to try for the lock again.
Thread 3 acquired the lock.
Thread 1 released the lock.
Thread 2 could not acquire the lock. Need to try for the lock again.
Thread 2 could not acquire the lock. Need to try for the lock again.
Thread 2 acquired the lock.
Thread 3 released the lock.
Thread 2 released the lock.
```

Note

The preceding output may change as the order of thread execution cannot be predicted.

In the preceding code, three threads namely `Thread 1`, `Thread 2`, and `Thread 3` are created. Also, a new `ReentrantLock` named `lock` is created. Then, on executing the `run()` method, a thread acquires a lock in the `try` block. Further, the `lock.tryLock(400, TimeUnit.MILLISECONDS)` statement is used to check if a lock is available in the time period of `400` milliseconds. If it is, then it is acquired, else, the threads need to try again for the lock. Once the task is complete, a thread releases the lock in the `finally` block, which ensures that the lock is finally released when a thread finishes its execution.

In addition, to implement separate locks for the read and writes operations, respectively, Java provides the `ReentrantReadWriteLock` class in the `java.util.concurrent.locks` package. This class allows you to have separate read and write versions of the same lock in such a way that only one thread is able to write and other threads are able to read at a given time. The `ReentrantReadWriteLock` class implements the `ReadWriteLock` interface.

The following table lists the constructors of the `ReentrantReadWriteLock` class.

Constructor	***Description***
`ReentrantReadWriteLock()`	*This method is used to create an instance of the `ReentrantReadWriteLock` class.*
`ReentrantReadWriteLock (boolean fairness)`	*This method is used to create an instance of the `ReentrantReadWriteLock` class and set the fairness. When the fairness option is set to `true`, it ensures that the lock is given to the longest waiting thread in the queue of waiting threads.*

The Constructors of the ReentrantReadWriteLock Class

The following table lists some of the methods of the `ReentrantReadWriteLock` class.

Method	***Description***
`Lock readLock()`	*This method is used to return the lock that is used for reading.*
`Lock writeLock()`	*This method is used to return the lock that is used for writing.*

The Methods of the ReentrantReadWriteLock Class

Consider the following code that shows how to implement `ReentrantReadWriteLock`:

```
import java.util.Random;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class ThreadLock extends Thread {

    private static final ReentrantReadWriteLock rwlock = new
ReentrantReadWriteLock();
    @Override
    public void run()
    {
        try {
            readData();
            sleep(2000);
            writeData();
        } catch (final Exception e)
        {
            System.out.println(e);
        }
    }
    public void readData()
    {
        try {
            rwlock.readLock().lock();
            System.out.println(Thread.currentThread().getName() + " is
reading and the value is 5");
        } finally {
            System.out.println(Thread.currentThread().getName() + " is
exiting after reading.");
            rwlock.readLock().unlock();
        }
    }

    public void writeData()
    {
        try {
            rwlock.writeLock().lock();
            Random rand = new Random();
            int n = rand.nextInt(50);
            System.out.println(Thread.currentThread().getName() + " has the
write lock and is writing.");
            System.out.println("The new value is: " + (5 + n));
        }
        finally
        {
            System.out.println(Thread.currentThread().getName() + " is
releasing the lock and exiting after writing.");
            rwlock.writeLock().unlock();
        }
    }
}

public class ReentrantLockTest {
    public static void main(final String[] args) throws Exception {
        ThreadLock obj = new ThreadLock();
        ThreadLock obj2 = new ThreadLock();
        obj.start();
        obj2.start();
    }
}
```

In the preceding code, a new `ReentrantReadWriteLock` lock named `rwlock` is created in the `ThreadLock` class that extends the `Thread` class. A method named `readData()` is created in the same class. When a thread begins executing this method, it obtains a read lock in the `try` block and releases it after performing its task in the `finally` block. The `readData()` method can be accessed by multiple threads for reading. Further, when a thread enters the `writeData()` method, it obtains a write lock and only the thread that has acquired the write lock can enter the critical section of the `writeData()` method. Finally, after performing the task, the thread releases the write lock so that it can be acquired by another waiting thread.

Once the preceding code is executed, the following output is displayed:

```
Thread-1 is reading and the value is 5
Thread-0 is reading and the value is 5
Thread-1 is exiting after reading.
Thread-0 is exiting after reading.
Thread-1 has the write lock and is writing.
The new value is: 9
Thread-1 is releasing the lock and exiting after writing.
Thread-0 has the write lock and is writing.
The new value is: 54
Thread-0 is releasing the lock and exiting after writing.
```

Note

The preceding output may change as the order of thread execution cannot be predicted.

Identifying Concurrency Synchronizers

Consider a scenario of a multithreaded application where you want to achieve coordination and concurrency amongst the threads of the application to minimize data inconsistency and errors. For example, you want the multiple threads of your application to exchange data or you want one or more threads to execute after a specific operation has been performed by another thread in the application. In such a scenario, you can use the synchronizer classes provided in the `java.util.concurrent` package. The following synchronizer classes are available:

- `Semaphore`
- `CountdownLatch`
- `CyclicBarrier`
- `Phaser`
- `Exchanger`

Semaphore

A *semaphore* is a lock with an internal counter. It is used to restrict access to shared resources by multiple threads at the same time. In addition, a semaphore can also be used to restrict the number of threads that can access a shared resource. Semaphores contain permits that should be acquired by a thread that wants access to a shared resource. Once the thread completes its work with the shared resource, the permit must be released by the thread. The `acquire()` and the `release()` methods are used to acquire and release a lock, respectively. If a permit is not available for a thread, then the requesting thread must wait until a permit is available to be acquired. As compared to a `synchronized` block, a semaphore provides the fairness capability. The fairness capability of the semaphore can be used to control the order in which the permits are given to requesting threads. When the fairness is set to `true`, then the semaphore gives permits in the order they were requested by the threads. For example, the thread that requested the permit first is given the permit. When fairness is set to `false`, a permit can be given to any thread that is waiting to acquire the permit. The use of a semaphore with fairness set to `true` helps in avoiding the starvation problem amongst threads in a concurrent environment.

CountDownLatch

This synchronizer class is useful in a situation where one or more threads need to wait until a set of operations or events is performed. Further, it is also useful when there is a need to synchronize multiple threads with the execution of a set of events or operations. When this synchronizer class is implemented, it is initialized with an integer number. This number is a counter value that represents the number of operations after which the waiting threads will start executing. This counter value can be retrieved by using the `getCount()` method. A thread that needs to wait for the completion of the preceding operations uses a method named `await()`. This method puts that thread to sleep until the operations are completed. When an operation completes, another method named `countDown()` is used to reduce the number of count by one. Finally, when the countdown reaches zero, all the sleeping threads are released.

CyclicBarrier

This synchronizer class is useful in a situation where multiple threads are required to wait at a predefined execution point that can be known as a barrier. For this, the `await()` method is used. This method is used by threads to wait for each other to reach the barrier. Once all the threads reach at the same barrier, then a task is performed. When this class is instantiated, you can specify the number of threads that will be participating to reach an execution point. The number of threads specified cannot vary over time. Additionally, during instantiation, the task to be performed, once the wait state is reached, can be specified. Then, the threads can complete their part and reach the same execution wait point. When the execution wait point is reached by all the threads, they can start working on the next part of their task.

Phaser

This synchronizer class is useful in a situation where independent threads need to work in phases or steps in order to complete a task. This class provides a similar functionality as provided by the `CyclicBarrier` class. However, by using this class, the number of threads that require to be synchronized by using a `Phaser` can be changed dynamically. To implement synchronization by using the `Phaser` class, an object of `Phaser` is created. A `Phaser` object can also be created with or without any threads. Further, the `register()` method is used to register each thread and the `arriveAndDeregister()` method is used to deregister a registered thread. The `bulkRegister()` method can also be used to register multiple threads in bulk. The `arriveAndAwaitAdvance()` method can be used to notify each thread to wait until the other threads reach the execution point. Once all the threads reach the execution point, the task to be executed can be specified by using the `onAdvance()` method.

Exchanger

This synchronizer class is useful in a situation where there is a need to perform synchronization and exchange of data between two concurrent running tasks. This class allows two threads to wait for a common point of exchange. If a thread reaches the point of exchange early, then it waits for the other thread to reach the common point of exchange. When both threads reach at this common point, they exchange the data shared by them. The method used to exchange data between threads is named `exchange()`. When a thread gets ready to exchange data, it invokes the `exchange()` method. If the other thread is not ready, the thread that invoked the exchange method is put to sleep till the time the other thread gets ready for data exchange.

Identifying Concurrency Collections

Consider a scenario where you need to create a concurrent program that implements collections to store and manage data. Some of the classes in the collection framework, such as `ArrayList`, cannot be used to work with multiple threads as the access to its data in a concurrent program cannot be controlled. For example, in a program, concurrent tasks on an unsynchronized collection of objects, such as `ArrayList`, by multiple threads can result in data inconsistency errors that can further affect the required functionality of the application. To avoid this, Java provides the concurrency collection classes in the `java.util.concurrent` package. The use of the collection classes improves the performance of a concurrent application and improves its scalability by allowing multiple threads to share a data collection. Some of the concurrency collections available in the `java.util.concurrent` package that can be used for concurrent programming are:

- `ArrayBlockingQueue`: This is a blocking FIFO based queue in which the elements are maintained by using an array. In a FIFO queue, the new elements are inserted at the end of the queue and an element is removed from the head of the queue. You can specify the capacity of the number of elements that can be stored in the queue, while creating an instance of this class. Moreover, there is an upper bound on the number of elements that can be stored in the queue. This upper bound can be specified only at the time of instantiation and cannot be changed later. If a thread attempts to add an element to this queue when it is full, the thread will be blocked until an existing element is removed from the queue.

- LinkedBlockingQueue: This is a FIFO based queue where the maximum capacity of elements to be stored in the list can be specified. This queue is implemented by using a linked list that is a collection of nodes. If a thread attempts to add an element to this queue when it is full, the thread will be blocked until an existing element is removed from the queue. The throughput of this queue is higher than ArrayBlockingQueue.
- PriorityBlockingQueue: This is a blocking queue that orders elements based on their priority. If a thread attempts to add an element to this queue when it is full, the thread will be blocked until an existing element is removed from the queue. In addition, a thread is blocked when a thread attempts to remove an element and the queue is empty.
- SynchronousQueue: This is a blocking queue that blocks the request to add data to the queue until a request to retrieve the data from the queue is received, and vice versa. This queue does not have the capacity to store any data element.
- DelayQueue: This queue performs the ordering of its elements on the basis of the remaining time before which they can be eliminated from the queue. In such a queue, an object can only be retrieved from the queue when its delay time is zero. Only those objects that implement the Delayed interface can be used with this queue.
- ConcurrentHashMap: This class is an implementation of the java.util.Map interface that provides concurrency. This class is suitable for the multithreaded environment as it segments the internal table. This allows multiple threads to perform concurrent read operations. However, the lock can be applied only on the segment that needs to be updated. As compared to a Hashtable class, this class does not allow a key to have a null value. Also, the duplicate values are not allowed.
- ConcurrentLinkedQueue: This is a FIFO based queue that is based on linked nodes. This queue can be used for concurrent access. In this queue, the head of the queue represents the element that has been in the queue for the longest period. On the other hand, the tail represents the element that has been in the queue for the smallest time. This queue does not allow null elements to be entered in the queue.
- ConcurrentSkipListMap: This is a concurrent implementation of the NavigableMap interface. You can use an object of the ConcurrentSkipListMap map to store key-value pairs that are sorted according to the natural ordering of its keys, or by a comparator provided at map creation time, depending on which a constructor is used. In addition, the insertion, removal, update, and access operations in a ConcurrentSkipListMap collection can be safely executed concurrently by multiple threads.
- ConcurrentSkipListSet: Similar to ConcurrentSkipListMap, you can use the ConcurrentSkipListSet class to store the elements in a Set collection, where the addition insertion, removal, update, and access operations can be safely executed concurrently by multiple threads. The ConcurrentSkipListSet class implements the NavigableSet interface of the java.util package to enable navigation methods reporting closest matches for given search targets. In addition, similar to the ConcurrentSkipListMap class, the elements stored in a ConcurrentSkipListSet are sorted according to the natural ordering or by a comparator.
- CopyOnWriteArrayList: This is a thread-safe implementation of the ArrayList class. In this list, the operations, such as add and set, are performed by making a fresh copy of the array. Such a list is preferred as compared to a synchronized ArrayList, when the number of reads to be performed is more than the number of updates to be performed.
- CopyOnWriteArraySet: This class represents a set that uses CopyOnWriteArrayList for its operations. As it uses CopyOnWriteArrayList for its operations, it is thread-safe. Further, its use is preferred when the number of reads to be performed is more than the number of updates to be performed.

Implementing ExecutorService

Consider a scenario where you need to build a Java application, in which several concurrent tasks need to be performed. In order to create such a concurrent application, you need to create and manage several threads. This is a cumbersome task as it requires writing a lot of program code. Further, with multiple threads, it becomes a challenge to efficiently manage the resources of a computer and the performance of the application. To avoid the preceding problems, you can use executors. An executor is a mechanism to execute tasks.

The `java.util.concurrent` package provides the executors feature. The executors can be used effectively to manage the thread tasks, such as their instantiation and execution, which improve the performance of an application when working with a large number of threads. By using executors, the creation and management of threads can be separated from the rest of the application. Some of the following executor interfaces defined in the `java.util.concurrent` package are:

- `Executor`
- `ExecutorService`

Executor

The `Executor` interface is used to execute tasks by using an executor. A *task* is an object that implements the `Runnable` or `Callable` interface. A task that implements the `Runnable` interface is a runnable task, and a task that implements the `Callable` interface is a callable task. The difference between a runnable task and a callable task is that the latter can return a result and throw a checked exception. While the functionality of a runnable task is defined by the `run()` method, the functionality of a callable task is defined by the `call()` method. As compared to the `run()` method, the `call()` method returns a result and is also allowed to throw exceptions if it cannot compute a result.

The `Executor` interface provides the `execute()` method that executes the `Runnable` tasks instead of using the `start()` method to start the runnable tasks. This method does not execute the `Callable` tasks. The signature of the `execute()` method is:

```
public void execute(Runnable task)
```

In the preceding signature, `task`, is a runnable task that is executed sometime in the future. The tasks are executed by using threads from a thread pool, which is a collection of threads that can be used to perform a work. Such threads are also known as worker threads. The worker threads do not die once they have completed their tasks. Instead, they become idle and can be reused to perform a task. This improves the efficiency by eliminating the need of creating separate threads.

Some of the limitations of using the `Executor` interface are:

- An executor cannot be used to execute a collection of runnable tasks.
- The executor executes runnable tasks and as the `run()` method of `Runnable` does not return a value, a runnable task cannot return a value to its calling method.
- An executor cannot be used for actions, such as cancel the execution of runnable tasks or determine when a runnable task has finished its execution.
- You cannot shutdown an executor.

The preceding limitations can be overcome by using the `ExecutorService` interface.

ExecutorService

An ExecutorService extends the Executor interface. It is also used for the purpose of submitting and executing tasks by using the thread pools. ExecutorService can be used with a large pool of threads to execute runnable or callable tasks. ExecutorService provides the additional methods that add more functionality for managing the tasks of an executor, such as shutting it down. The following table lists some of the methods of the ExecutorService interface.

Method	***Description***
boolean isShutdown()	*This method returns true if the executor has been shut down.*
void shutdown()	*This method is used to initiate a shutdown of the executor service.*
boolean isTerminated()	*This method is used to check if the executor service is in the terminated state. It returns true if all the pending tasks have finished executing after the shutdown.*
List<Runnable> shutdownNow()	*This method is used to stop all the tasks that are executing, and return a list of the tasks that are awaiting execution.*
Future<?> submit(Runnable task)	*This method is used to submit a runnable task for execution. It also returns a Future object which provides information about the status and result of the task.*
Future<?> submit(Callable<T> task)	*This method is used to submit a callable task and return a Future object of that task.*

The Methods of the ExecutorService Interface

You can use a fixed thread pool, ExecutorService, by using the newFixedThreadPool() method. The newFixedThreadPool() method has the following signature:

```
public static ExecutorService newFixedThreadPool(int numberofThreads)
```

In the preceding signature, numberofThreads specifies the number of threads in the pool.

You can create a thread pool comprising of two threads to execute the tasks that are submitted for execution by using the following code snippet:

```
ExecutorService name = Executors.newFixedThreadPool(2);
```

In the preceding code snippet, name is the name of the fixed thread pool executor service to be created. In order to create a thread pool with a fixed amount of threads, the newFixedThreadPool() method of the Executors class is used. The Executors class is a utility class. It provides the functionality to create the Executor and ExceutorService objects.

You can also create a cached thread pool, `ExecutorService`, by using the `newCachedThreadPool()` method. This method has the following signature:

```
public static ExecutorService newCachedThreadPool()
```

A cached thread pool allows:

- The creation of new threads as and when they are required.
- The reuse of threads as the threads used in the pool can be reused after they finish their tasks.
- The termination of threads that have been idle for a period of 60 seconds.

You can use the following code snippet to create a new cached thread pool executor service:

```
ExecutorService threadpool = Executors.newCachedThreadPool();
```

In the preceding code snippet, a new cached thread pool executor service named `threadpool` is created. Consider the following code that shows how to work with the executor service and execute `Runnable` tasks:

```
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task implements Runnable {

    String taskname;
    public Task(String name)
    {
    taskname = name;
    }
      public void run()
    {
            System.out.println("The task name is " +taskname +" executed by "
+Thread.currentThread().getName());
    }
}
class TaskExecutor {
    public static void main(String a[])
    {
      Task task1 = new Task("Task1");
      Task task2 = new Task("Task2");
      Task task3 = new Task("Task3");
    ExecutorService threadexecutor = Executors.newCachedThreadPool();
    System.out.println("Executor started");
    threadexecutor.execute(task1);
    threadexecutor.execute(task2);
    threadexecutor.execute(task3);
    threadexecutor.shutdown();
    if(threadexecutor.isShutdown())
         {
            System.out.println("All tasks completed. The executor is shutting
down.");
         }
    }
}
```

The output of the preceding code is:

```
Executor started
The task name is Task2 executed by pool-1-thread-2
The task name is Task1 executed by pool-1-thread-1
The task name is Task3 executed by pool-1-thread-3
All tasks completed. The executor is shutting down.
```

In the preceding code, three tasks namely, `task1`, `task2`, and `task3` are completed. Further, a new cached thread pool executor service named `threadexecutor` is created and all the tasks are executed by the executor service. Further, it is checked if the executor service is shut down and an appropriate message is displayed.

Note
The preceding output may vary.

Consider the following code that shows how to work with the executor service and execute `Callable` tasks:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
class Task implements Callable {

    String taskname;
    public Task(String name)
    {
    taskname = name;
    }
      public String call()
{
      System.out.println("The task name is " +taskname +" executed by "
+Thread.currentThread().getName());
      return taskname;
    }
}
public class CallableTaskDemo {
    public static void main(String a[])
    {
      Task task1 = new Task("Task1");
      Task task2 = new Task("Task2");
      Task task3 = new Task("Task3");
    ExecutorService threadexecutorPool1 = Executors.newFixedThreadPool(3);
    ExecutorService threadexecutorPool2 = Executors.newFixedThreadPool(3);
    Future<String> f1=threadexecutorPool1.submit(task1);
    Future<String> f2=threadexecutorPool2.submit(task2);
    Future<String> f3=threadexecutorPool2.submit(task3);

    }
}
```

The output of the preceding code is:

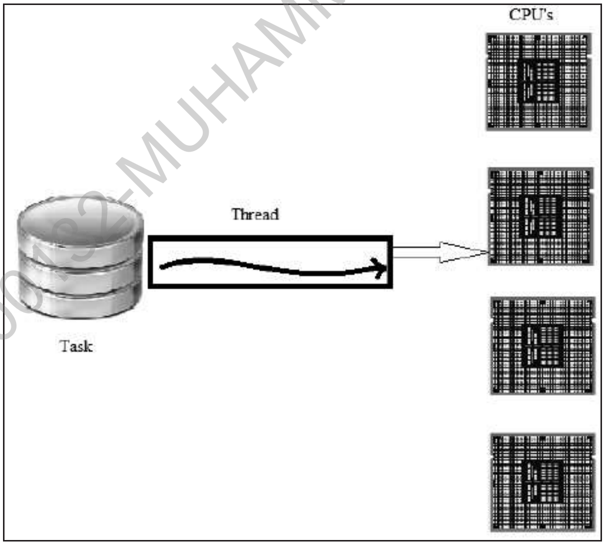
```
The task name is Task1 executed by pool-1-thread-1
The task name is Task3 executed by pool-2-thread-2
The task name is Task2 executed by pool-2-thread-1
```

In the preceding code, a class named `Task` is created. The `Task` class implements the `Callable` interface. This class also overrides `call()`. Further, in the class, `CallableTaskDemo`, two executor services named `threadexecutorPool1` and `threadexecutorPool2` are created. Then, the task objects, `task1`, `task2`, and `task3` of the `Task` class, are created. The `task1` task is submitted to the `threadexecutorPool1` pool for execution, and `task2` and `task3` are submitted to `threadexecutorPool2` for execution.

Implementing Fork/Join Framework

Consider a situation where you need to create a high-scale concurrency application that is supposed to run on computers with multiple processors. In case of such an application, there may be a situation where the application is not utilizing all the available processors for executing its tasks. This will result in the wastage of computing power and loss of efficiency. To overcome this problem and achieve the maximum CPU utilization, you need to ensure that several tasks are executed in parallel on multiple processors. To cater to the preceding requirements, Java provides the fork/join framework.

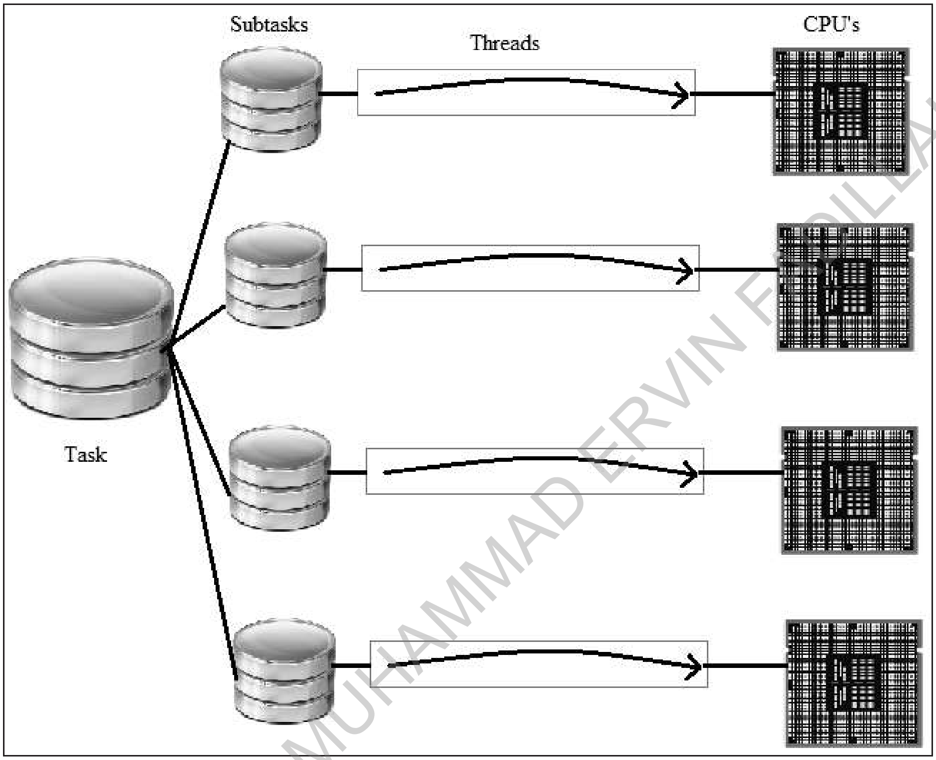
The following figure shows how the computing power can be wasted when an application is not utilizing all the available processors for executing its tasks.



The Wastage of CPU Utilization

The fork/join framework is based on the principle of decomposing a larger task into smaller tasks until it cannot be further broken down. Then, the smaller tasks are executed in parallel and their results are combined to obtain the final result.

The following figure shows how the effective CPU utilization can be achieved by using the fork/join framework.



The Effective CPU Utilization

The fork/join framework utilizes a thread pool to execute the subtasks of a task. In this framework, if a thread remains idle after completing its task, then it can be used to steal the pending work of another thread. This is known as work stealing. Work stealing improves the execution speed and performance of an application. However, if there is a single CPU, then the fork/join framework is not beneficial.

The following table lists the classes of the `java.util.concurrent` package that can be used for implementing the fork/join framework.

Class	***Description***
`ForkJoinPool`	*This method implements the `ExecutorService` interface. It is used to create a pool of threads to execute tasks and implement work stealing.*
`ForkJoinTask`	*This method is the super class of all the tasks that will be executed in `ForkJoinPool`. These tasks are lighter in weight than threads. `ForkJoinTask` starts execution when it is submitted to `ForkJoinPool`.*

Class	***Description***
`RecursiveTask`	*This method is used when a task to be executed by using the fork/join framework needs to return a result.*
`RecursiveAction`	*This method is used when a task to be executed by using the fork/join framework does not need to return a result.*

The Classes of the java.util.concurrent Package

Consider a scenario where you need to find out the maximum value from a large array of integers. In such a scenario, you can implement the fork/join framework to perform the computation in parallel and achieve the maximum processing efficiency. The following code shows how to implement the fork/join programming to find the maximum value from an array of integers:

```
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class MaxFind extends RecursiveTask<Integer> {

  private static final int SEQUENTIAL_THRESHOLD = 100;

  private final int[] data;
  private final int start;
  private final int end;

  public MaxFind(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
  }

  public MaxFind(int[] data) {
    this(data, 0, data.length);
  }

  protected Integer compute() {
    final int length = end - start;
    if (length < SEQUENTIAL_THRESHOLD) {
      return computeDirectly();
    }
    final int split = length / 2;
    final MaxFind left = new MaxFind(data, start, start + split);
    left.fork();
    final MaxFind right = new MaxFind(data, start + split, end);
    return Math.max(right.compute(), left.join());
      }

  private Integer computeDirectly() {
    System.out.println(Thread.currentThread() + " is searching array index: "
+ start    + " to " + end);
    int max = Integer.MIN_VALUE;
    for (int i = start; i < end; i++) {
      if (data[i] > max) {
        max = data[i];
      }
    }
       return max;
  }
  public static void main(String[] args) {
    // create a random data set
    final int[] data = new int[200];
    final Random random = new Random();
    for (int i = 0; i < data.length; i++) {
      data[i] = random.nextInt(350);
    }
// submit the task to the pool
    final ForkJoinPool pool = new ForkJoinPool(4);
    final MaxFind finder = new MaxFind(data);
    System.out.println("The max value is: "+pool.invoke(finder));

  }
}
```

The output of the preceding code is:

```
Thread[ForkJoinPool-1-worker-1,5,main] is searching array index: 150 to 200
Thread[ForkJoinPool-1-worker-4,5,main] is searching array index: 0 to 50
Thread[ForkJoinPool-1-worker-2,5,main] is searching array index: 50 to 100
Thread[ForkJoinPool-1-worker-3,5,main] is searching array index: 100 to 150
The max value is: 349
```

In the preceding code, a class named MaxFind is created. This class extends the RecursiveTask class to implement the fork/join framework. The use of the RecursiveTask class enables a programmer to divide a larger task into smaller tasks, and run them concurrently to achieve efficiency. In the main method, an array of 200 integers is created and random values upto the range of 350 are inserted in it. Then, a ForkJoinPool object is created with 4 worker threads that will be used to perform the computation. Further, a fork/join task named finder is created and submitted to fork/join pool by using the invoke() method. The invoke() method will be used to return the result of the fork/join task.

In the MaxFind class, the method named compute contains the logic to perform the computation and divide the task into subtasks, if required. The SEQUENTIAL_THRESHOLD parameter is used to specify a threshold that is used to determine if the task is large enough to be divided into subtasks. In this case, SEQUENTIAL_THRESHOLD has a value, 100, that indicates that if the array size is equal to or less than 100 elements, then the task is to be carried out sequentially as a single task without dividing it into multiple tasks. However, if the array size is greater than 100, then the task will be divided into multiple tasks. The fork() method is used to split the task into subtasks named left and right and compute them in parallel. Further, the join() method is used to get the result of the computations of the left and right subtasks as they are executed in parallel. The computeDirectly() method is used to search the array for the maximum number and return the maximum number.

Just a minute:

The ______ class is the super class of all the tasks that will be executed in `ForkJoinPool`.

Answer:

`ForkJoinTask`

Practice Questions

1. Identify the keyword that is used to synchronize a method.
   a. `synchronize`
   b. `synchronized`
   c. `lock`
   d. `monitor`

2. Identify the method that is used to release a lock temporarily.
   a. `wait()`
   b. `release()`
   c. `synchronize()`
   d. `notify()`

3. Identify the method that is used to wake up a single thread that is waiting for the monitor of the object being executed.
   a. `wait()`
   b. `release()`
   c. `synchronize()`
   d. `notify()`

4. Identify the package that provides the classes and methods that can be used to create atomic variables for performing atomic operations.
   a. `java.util.concurrent.atomic`
   b. `java.util.atomic`
   c. `java.util.concurrent.atomicInteger`
   d. `java.util.concurrent.Integer`

5. Identify the method that is used when a lock must be acquired by a thread only if it is available.
   a. `unlock()`
   b. `tryLock()`
   c. `lock()`
   d. `checklock()`

Summary

In this chapter, you learned that:

- The synchronization of threads ensures that if two or more threads need to access a shared resource, then that resource is used by only one thread at a time.
- You can implement thread synchronization by using the following approaches:
  - By using the synchronized methods
  - By using the synchronized statement
- Synchronization is based on the concept of a monitor, which is an object used as a lock.
- When a thread is within a synchronized method, all the other threads that try to call the method at the same time have to wait.
- The synchronized statement is used where the synchronization methods are not used in a class and you do not have access to the source code.
- Multithreading eliminates the concept of polling by a mechanism known as interprocess communication.
- The various methods used in inter-threaded communication are:
  - `wait()`
  - `notify()`
  - `notifyAll()`
- Concurrency enables Java programmers to improve the efficiency and resource utilization of their applications by creating concurrent programs.
- To implement concurrency, Java provides the `java.util.concurrent` package.
- The `java.util.concurrent` package has the following sub packages to support concurrent programming:
  - `java.util.concurrent.atomic`
  - `java.util.concurrent.locks`
- Atomic variables are the single variables on which operations, such as increment and decrement, are done in a single step. Such operations are known as atomic operations.
- An atomic operation is an operation that is performed as a single unit.
- A reentrant lock is used for mutual locking amongst threads.
- In addition, to implement separate locks for the read and writes operations, respectively, Java provides the `ReentrantReadWriteLock` class in the `java.util.concurrent.locks` package.
- The following synchronizer classes are available:
  - `Semaphore`
  - `CountdownLatch`
  - `CyclicBarrier`
  - `Phaser`
  - `Exchanger`
- The use of the collection classes improves the performance of a concurrent application and improves its scalability by allowing multiple threads to share a data collection.

- The executors can be used effectively to manage the thread tasks, such as their instantiation and execution, which improve the performance of an application when working with a large number of threads.
- The `Executor` interface is used to execute tasks by using an executor.
- `ExecutorService` provides the additional methods that add more functionality for managing the tasks of an executor, such as shutting it down.
- The fork/join framework is based on the principle of decomposing a larger task into smaller tasks until it cannot be further broken down.

Chapter 7

Working with Streams

Most programs accept input from the user, process the same, and produce the output. Therefore, all programming languages support the reading of input from an input stream, such as a file, and display the output from an output stream, such as a console. Java handles all the input and output operations in the form of streams that act as a sequence of bytes or characters traveling from a source to a destination. When a stream of data is being sent, it is said to be written; and when a stream of data is being received, it is said to be read.

This chapter focusses on reading the data from an input stream and writing the same to an output stream.

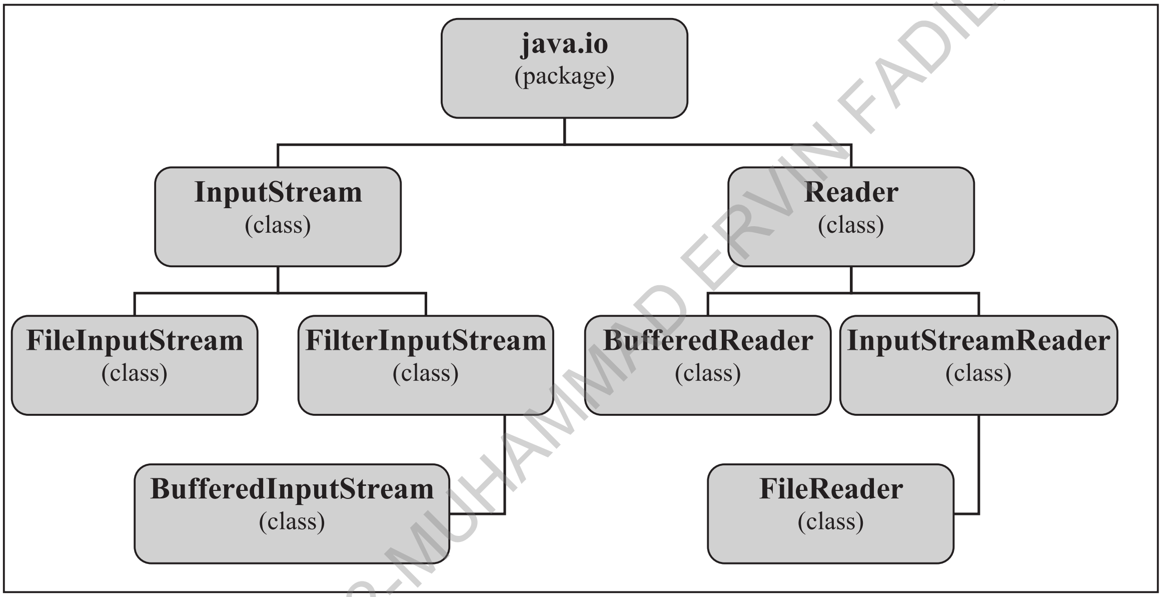
Objectives

In this chapter, you will learn to:

- Work with input stream
- Work with output stream

Working with Input Stream

Consider a scenario where you need to develop a Java application that converts the documents from one format into another, such as .txt into .pdf. To develop such an application, you need to read data from an input stream, such as a file. The data can be read in the form of bytes or characters from these streams. For this, Java provides the `InputStream` class and its subclasses. To read data in the form of characters, Java provides the `Reader` classes inside the `java.io` package. The following figure shows the class hierarchy of the `java.io` package.



The Input Stream Class Hierarchy

The `FileInputStream` and `FilterInputStream` classes are subclasses of the `InputStream` class. The `BufferedInputStream` class is a subclass of the `FilterInputStream` class. The `BufferedReader` and `InputStreamReader` classes are the subclasses of the `Reader` class. The `FileReader` class is the subclass of the `InputStreamReader` class.

Using the FileInputStream Class

The *`FileInputStream` class* is used to read data and the steams of bytes from the file. The `FileInputStream` class provides various constructors that can be used to create an instance.

The following table lists the constructors of the `FileInputStream` class.

Constructor	***Description***
`FileInputStream(File file)`	*It creates an instance of `FileInputStream` by opening a connection to an actual file, `file`, in the file system.*
`FileInputStream(FileDescriptor fdObj)`	*It creates an instance of `FileInputStream` by using the file descriptor, `fdObj`, which represents an existing connection to an actual file in the file system.*
`FileInputStream(String name)`	*It creates `FileInputStream` by opening a connection to an actual file specified in the `name` parameter.*

The Constructors of the FileInputStream Class

The following table lists the methods of the `FileInputStream` class with their description.

Method	***Description***
`int read()`	*Is used to read a byte of data from the input stream.*
`FileDescriptor getFD()`	*Is used to return the `FileDescriptor` object that represents the connection to the actual file in the file system.*

The Methods of the FileInputStream Class

Consider the following code to read the data from a file using the `FileInputStream` class:

```
import java.io.FileInputStream;
import java.io.IOException;
public class FileInputStreamDemo
{

  public static void main(String[] args)
  {
    int i;
    char c;
    try (FileInputStream f = new FileInputStream("D:\\Files\\File.txt"))
    {
      while ((i = f.read()) != -1)
      {
        c = (char) i;
        System.out.print(c);
      }
    }
    catch (IOException ex)
```

```
                {
                    System.out.println(ex);
                 }
             }
         }
```

In the preceding code, the `FileInputStreamDemo` class is created that is reading data from the `File.txt` file by using the `read()` method. The `System.out.print(c);` statement prints the bytes of data present in the file.

Using the BufferedInputStream Class

The *`BufferedInputStream` class* is used to perform the read operations by using a temporary storage, buffer, in the memory. A buffer is used to store the retrieved data when the first system call to read data is made. In addition, for every subsequent request to read data, the buffer is used to retrieve data by a program instead of making system calls. The retrieval of data by using a buffer instead of a system call is much faster; and hence, it improves the efficiency of the program. The `BufferedInputStream` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `BufferedInputStream` class.

Constructor	***Description***
`BufferedInputStream(InputStream in)`	*It creates an instance of `BufferedInputStream` and saves the specified `InputStream` arguments in the `in` parameter for later use.*
`BufferedInputStream(InputStream in, int size)`	*It creates an instance of `BufferedInputStream` with the specified buffer size, and saves the `InputStream` argument in the `in` parameter for later use.*

The Constructors of the BufferedInputStream Class

The following table lists the methods of the `BufferedInputStream` class with their description.

Method	***Description***
`int available()`	*Is used to return an estimate of the number of bytes that can be read from this input stream without blocking the next invocation of a method for this input stream.*
`void mark(int readlimit)`	*Is used to mark the present position in the stream. Subsequent calls to `reset()` will attempt to reposition the stream to this point.*

Method	***Description***
void reset()	*The reset() method returns the stream to the marked point.*
boolean markSupported()	*Is used to test if this input stream supports the mark() and reset() methods.*
void close()	*Is used to close the input stream and release any system resource associated with the stream. Once the stream is closed, further read(), available(), reset(), or skip() invocation will throw IOException.*
int read(byte[] b,int off,int len)	*Is used to read bytes from the input stream into the specified byte array, starting from the given offset.*
long skip(long n)	*Is used to skip over n bytes of data from this input stream.*

The Methods of the BufferedInputStream Class

Consider the following code of BufferedInputStream that prints the output on the console:

```
import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;

public class BufferedInputStreamDemo
{
    public static void main(String[] args)
    {
        String s = "This is a BufferedInputStream Demo Program";
        byte buf[] = s.getBytes();

        try(ByteArrayInputStream in = new ByteArrayInputStream(buf);
                BufferedInputStream f = new BufferedInputStream(in)){

        int c;
        while ((c = f.read()) != -1)
        {
            System.out.print((char) c);
        }
    }
 catch(IOException e)
{
        System.out.println(e);
    }
}
}
```

In the preceding code, the `BufferedInputStreamDemo` class has been created. The `System.out.print((char) c);` statement prints the output on the console.

> ***Note***
>
> *A `ByteArrayInputStream` class allows you to create a buffer in memory that will be used as `InputStream`. The input source is a byte array.*

Using the FileReader Class

The *`FileReader` class* is used for reading characters from a file, but it does not define any method of its own. It derives all the methods from its base classes, such as the `Reader` and `InputStreamReader` classes. The `FileReader` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `FileReader` class.

Constructor	***Description***
`FileReader(File file)`	*It creates an instance of `FileReader` that reads from the file specified in the `file` parameter.*
`FileReader(FileDescriptor fd)`	*It creates an instance of `FileReader` that reads from the specified `FileDescriptor` class.*
`FileReader(String fileName)`	*It creates an instance of `FileReader` that reads from the file with the name specified by the `fileName` parameter.*

The Constructors of the FileReader Class

Consider the following code that reads data from the file:

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

class FileReaderDemo {
    public static void main(String args[])
    {
        try (FileReader f = new FileReader("D:\\Files\\file.txt "))
        {
            char[] a = new char[50];
            f.read(a);
            for (char c : a)
            {
                System.out.print(c);
            }
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

In the preceding code, the `FileReaderDemo` class has been created that reads the data from the file, `file.txt`. The `System.out.print(c);` statement prints the output on the console.

Using the BufferedReader Class

The *`BufferedReader` class* is used to read the text from a character-input stream, such as a file, console, and array, while buffering characters. As the data is buffered, the read operation becomes more efficient using the `BufferedReader` class. There are various constructors provided by the `BufferedReader` class that can be used to create an instance. The following table lists the constructors of the `BufferedReader` class.

Constructor	***Description***
`BufferedReader(Reader in)`	*It creates an instance of `BufferedReader` that uses a default-sized input buffer.*
`BufferedReader(Reader in, int sz)`	*It creates an instance of `BufferedReader` that uses an input buffer of the specified size.*

The Constructors of the BufferedReader Class

The following table lists the commonly used methods of the `BufferedReader` class with their description.

Method	***Description***
`void mark(int readAheadLimit)`	*Is used to mark the present position in the stream. Subsequent calls to `reset()` will attempt to reposition the stream to this point.*
`boolean markSupported()`	*Is used to tell whether this stream supports the `mark()` operation.*
`String readLine()`	*Is used to read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.*
`boolean ready()`	*Is used to tell whether this stream is ready to be read. A buffered character stream is ready if the buffer is not empty, or if the underlying character stream is ready.*

The Methods of the BufferedReader Class

Consider the following code that accepts two numbers from the user by using the `BufferedReader` class and prints the sum of these two numbers:

```
import java.io.*;

public class BufferedReaderDemo
{
    public static void main(String args[]) throws IOException
    {
        try(BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))){

        System.out.println("Enter First number");
        String s = br.readLine();
        System.out.println("Enter Second number");
        String s1 = br.readLine();
        int i = Integer.parseInt(s);
        int i1 = Integer.parseInt(s1);
        int i3 = i + i1;
        System.out.println("Sum=" + i3);
        }
    }
}
```

In the preceding code, an object of the `BufferedReaderDemo` class has been created that wraps an `InputStreamReader` class to read data from the console. The `readLine()` method accepts the input from the user and stores it in the string variables, `s` and `s1`, respectively. Thereafter, the string value is converted into the integer value by using the `parseInt()` method, which is the static method of the `Integer` class. Further, the `System.out.println("Sum=" + i3);` statement prints the output.

Just a minute:

Which one of the following methods is used to ensure that the `close()` method of the `FileInpuStream` class is called when there are no more references to it?

1. *`finalize()`*

2. *`ready()`*

3. *`reset()`*

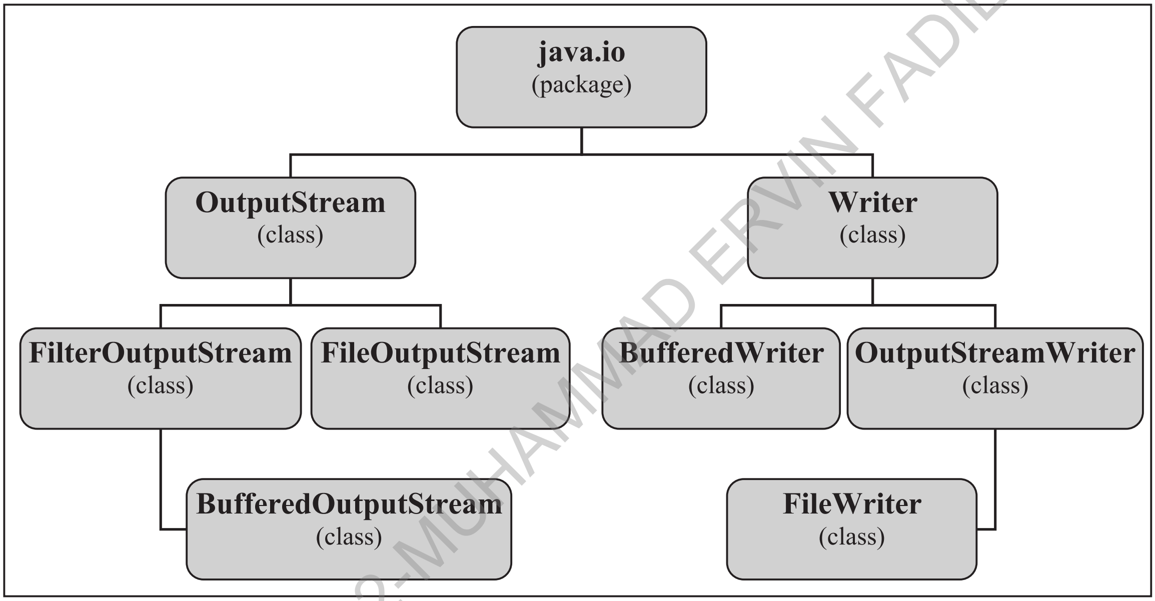
4. *`close()`*

Answer:

1. *`finalize()`*

Working with Output Stream

Consider a scenario where you need to develop a Banking application. In this application, you need to implement a set of functionality to maintain a log for every transaction performed in the application. For this, you need to write the data to an output stream, such as a file. The data can be written in the form of bytes or characters to these streams. To write the data in the form of bytes, Java provides the `OutputStream` classes. To write the data in the form of characters, Java provides the `Writer` classes inside the `java.io` package. The following figure shows the class hierarchy of the `java.io` package.



The Output Stream Class Hierarchy

The `FileOutputStream` and `FilterOutputStream` classes are the subclasses of the `OutputStream` class. The `BufferedOutputStream` class is subclass of the `FilterOutputStream` class. The `BufferedWriter` and `OutputStreamWriter` classes are the subclasses of the `Writer` class. The `FileWriter` class is the subclass of the `OutputStreamWriter` class.

Using the FileOutputStream Class

The *`FileOutputStream` class* is used for writing data, byte by byte, to a file. The instance creation of `FileOutputStream` class is not dependent on a file on which the writing needs to be done. If the file does not exist, then it will create a file before opening it for the output operation. If the file is in the read-only mode, then it will throw an error. The `FileOutputStream` class provides various constructors that can be used to create an instance.

The following table lists the constructors of the `FileOutputStream` class.

Constructor	***Description***
`FileOutputStream(File file, boolean append)`	*It creates an instance of `FileOutputStream` that writes the file represented by the specified `File` object. If the `append` parameter is `true`, then bytes will be written at the end of the file. Otherwise, bytes will be written at the beginning.*
`FileOutputStream(FileDescriptor fdObj)`	*It creates an instance of `FileOutputStream` that writes to the file descriptor, which represents an existing connection with the actual file.*
`FileOutputStream(String name)`	*It creates an instance of `FileOutputStream` that writes to the file with the name specified by the `name` parameter.*
`FileOutputStream(String name, boolean append)`	*It creates an instance of `FileOutputStream` that writes to the file with the name specified in the `name` parameter. If the `append` parameter is `true`, then bytes will be written at the end of the file. Otherwise, bytes will be written at the beginning.*

The Constructors of the FileOutputStream Class

The following table lists the commonly used methods of the `FileOutputStream` class with their description.

Method	***Description***
`FileDescriptor getFD()`	*Is used to return the file descriptor associated with this stream.*
`void write(int b)`	*Is used to write the specified byte to this file output stream.*

The Methods of the FileOutputStream Class

Consider the following code to write the data to a file using the `FileOutputStream` class:

```
import java.io.*;
public class FileOutputStreamDemo
{
    public static void main(String[] args) throws IOException
    {
            boolean bool;
            long pos;
            String s = "This is a FileOutputStream Program";
            byte buf[] = s.getBytes();
```

```
             try (FileOutputStream fos = new
          FileOutputStream("D:\\Files\\File.txt "))
             {
                    for (int i = 0; i < buf.length; i++)
                    {
                           fos.write(buf[i]);
                    }
             }
             catch (Exception e)
             {
                    System.out.println(e);
             }
       }
    }
```

In the preceding code, the `FileOutputStreamDemo` class has been created. The `fos.write(buf[i]);` statement writes the data into the file.

Using the BufferedOutputStream Class

The *`BufferedOutputStream` class* writes bytes to an output stream using a buffer for increased efficiency. The `BufferedOutputStream` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `BufferedOutputStream` class.

Constructor	***Description***
`BufferedOutputStream(OutputStream out)`	*It creates an instance of the buffered output stream that writes data to the specified output stream.*
`BufferedOutputStream(OutputStream out, int size)`	*It creates an instance of the buffered output stream to write data to the specified output stream with the specified buffer size.*

The Constructors of the BufferedOutputStream Class

The following table lists the commonly used methods of the `BufferedOutputStream` class with their description.

Method	***Description***
`void flush()`	*Is used to flush the buffered output stream. This forces any buffered output bytes to be written out to the output stream.*
`void write(byte[] b, int off, int len)`	*Is used to write `len` bytes from the byte array. It starts writing from the offset value, `off`, to the buffered output stream.*

The Methods of the BufferedOutputStream Class

Consider the following code of the BufferedOutputStream class that prints the output on the console:

```
import java.io.BufferedOutputStream;
import java.io.IOException;

public class BufferedOutputStream
{
    public static void main(String[] args)
    {
        try (BufferedOutputStream b = new BufferedOutputStream(System.out))
        {
            String s = "This is a BufferedOutputStream Demo Program";
            byte buf[] = s.getBytes();

            b.write(buf);
            b.flush();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

In the preceding code, the BufferedOutputStream class has been created. The b.write(buf); statement prints the output on the console.

Using the BufferedWriter Class

The *BufferedWriter class* can be used to write text to an output stream. This class writes relatively large chunks of data to an output stream at once. The BufferedWriter class provides various constructors that can be used to create an instance. The following table lists the constructors of the BufferedWriter class.

Constructor	***Description***
BufferedWriter(Writer out)	*It creates an instance of BufferedWriter that uses a default-sized output buffer.*
BufferedWriter(Writer out, int size)	*It creates an instance of BufferedWriter that uses an output buffer of the given size.*

The Constructors of the BufferedWriter Class

The following table lists the commonly used methods of the `BufferedWriter` class with their description.

Method	***Description***
`void newLine()`	*Is used to write a line separator that is defined by the system property, `line.separator`.*
`void write(char[] cbuf, int off, int len)`	*Is used to write a portion of an array of characters.*

The Methods of the BufferedWriter Class

Consider the following code to write the data to the console using the `BufferedWriter` class:

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class BufferedWriterDemo
{
    public static void main(String args[])
    {
        try (BufferedWriter b = new BufferedWriter(new
OutputStreamWriter(System.out)))
        {
            String fruit[] = {"Apple", "Banana", "Grapes"};
            b.write("Different types of fruit are:" + "\n");
            for (int i = 0; i < 3; i++)
            {
                b.write(fruit[i] + "\n");
                b.flush();
            }
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

The output of the preceding code is:

```
Different types of fruit are:
Apple
Banana
Grapes
```

In the preceding code, an array of fruits has been created. Then, an object of `BufferedWriter` has been created that wraps the `OutputStreamWriter` class to write data to the console output stream. The `write()` method writes the string to the console.

Using the FileWriter Class

The `FileWriter` *class* writes character data to a file. This class does not define any methods of its own. It derives all methods from its base classes, such as the `Writer` and the `OutputStreamWriter` classes.

The `FileWriter` class provides various constructors that can be used to create an instance. The following table lists the constructors of the `FileWriter` class.

Constructor	***Description***
`FileWriter(File file)`	*It creates an instance of the `FileWriter` object from a `File` object.*
`FileWriter(File file,boolean append)`	*It creates an instance of the `FileWriter` object from a `File` object. If the second argument is `true`, then characters will be written at the end of the file. Otherwise, characters will be written at the beginning of the file.*
`FileWriter(FileDescriptor fd)`	*It creates an instance of the `FileWriter` object associated with a file descriptor.*
`FileWriter(String fileName,boolean a ppend)`	*It creates an instance of the `FileWriter` object with a given file name. If the second argument is `true`, then characters will be written at the end of the file. Otherwise, characters will be written at the beginning of the file.*

The Constructors of the FileWriter Class

Consider the following code that writes the data to the file:

```
import java.io.FileWriter;
import java.io.IOException;

class FileWriterDemo
{
    public static void main(String args[])
    {
        try (FileWriter f = new FileWriter("D:\\Files\\file.txt "))
        {
            String source = "This is FileWriter Program";
            char buffer[] = new char[source.length()];
            source.getChars(0, source.length(), buffer, 0);
            f.write(buffer);
        }

        catch (IOException e)
        {
            System.out.println(e);
        }
        }
    }
```

In the preceding code, the `FileWriterDemo` class has been created that writes data to the `file.txt` file. The `f.write(buffer);` statement writes the data to the file.

Just a minute:

Which one of the following `FileWriter` constructors will you use if there is a requirement to append data at the end of a new file?

1. *`FileWriter(File file)`*
2. *`FileWriter(File file,boolean append)`*
3. *`FileWriter(FileDescriptor fd)`*
4. *`FileWriter(String fileName,boolean append)`*

Answer:

4. *`FileWriter(String fileName,boolean append)`*

Activity 7.1: Working with Input Stream and Output Stream

Practice Questions

1. Which one of the following constructors of the `FileOutputStream` class is used to write bytes at the end of a given file?

   a. `FileOutputStream(File file,boolean append)`
   b. `FileOutputStream(FileDescriptor fdObj)`
   c. `FileOutputStream(File file)`
   d. `FileOutputStream(String name,boolean append)`

2. State whether the following statement is true or false.

   The `markSupported()` method is used to test whether the input stream supports the `mark()` and `reset()` methods.

3. Which one of the following methods is used to return the estimate number of bytes that can be read from an input stream?

   a. `available()`
   b. `reset()`
   c. `read()`
   d. `ready()`

4. Which one of the following classes does not have its own methods and inherits the methods of the `InputStreamWriter` class?

   a. `FileWriter`
   b. `BufferedWriter`
   c. `BufferedOutputStream`
   d. `FileOutputStream`

5. Which one of the following classes does not have its own methods and inherits the methods of the `InputStreamReader` class?

   a. `FileReader`
   b. `BufferedReader`
   c. `BufferedInputStream`
   d. `FileInputStream`

Summary

In this chapter, you learned that:

- Java handles all the input and output operations in the form of streams that act as a sequence of bytes or characters traveling from a source to a destination.
- When a stream of data is being sent, it is said to be written; and when a stream of data is being received, it is said to be read.
- To read data in the form of characters, Java provides the `Reader` classes inside the `java.io` package.
- The `FileInputStream` class is used to read data and the steams of bytes from the file.
- The `BufferedInputStream` class is used to perform the read operations by using a temporary storage, buffer, in the memory.
- The `FileReader` class is used for reading characters from a file, but it does not define any method of its own.
- The `BufferedReader` class is used to read the text from a character-input stream, such as a file, console, and array, while buffering characters.
- To write the data in the form of bytes, Java provides the `OutputStream` classes.
- To write the data in the form of characters, Java provides the `Writer` classes inside the `java.io` package.
- `FileOutputStream` is used for writing data, byte by byte, to a file.
- The `BufferedOutputStream` class writes bytes to an output stream using a buffer for increased efficiency.
- The `BufferedWriter` class can be used to write text to an output stream.
- The `FileWriter` class writes character data to a file. This class does not define any methods of its own.

Chapter 8

Working with NIO Classes and Interfaces

You have learned how to use traditional Input/Output (I/O) operations using the classes and interfaces in the `java.io` package. However, the traditional I/O operations lack useful features, such as file locking. Therefore, to overcome such issues and further improve the efficiency of Java applications, the New Input Output (NIO) Application Programming Interface (API) has been introduced to provide simpler and high-performance I/O operations.

This chapter focuses on NIO classes and interfaces. In addition, you will learn to read/write from/to a file efficiently using the NIO classes.

Objectives

In this chapter, you will learn to:

- Get familiar with NIO
- Perform the read and write operations on a file

Introducing NIO

In Java applications that perform I/O operations with files, you need to store data and retrieve the same from files and work with directories to access files. I/O operations utilize system resources, and therefore, must be performed efficiently to improve the performance of the applications and achieve the maximum processing productivity. The functionality offered by the `java.io` package for I/O operations had performance related issues, such as a thread performing a write operation was blocked until some data was transferred. Further, there were efficiency-related issues, such as there was no method to copy a file or directory. To overcome this and further simplify I/O operations, Java provides the `java.nio` package. The `java.nio` package provides sub packages and classes that are useful for improving the speed and performance of I/O operations.

In order to perform I/O operations on files and directories using the NIO API, you need to make use of the `Path` interface and the `Paths` class. In addition, to work with files and directories, you need to use the `Files` class. Further, NIO also provides support for monitoring a directory by implementing a watch service.

Using the Path Interface and the Paths Class

The `Path` interface is a component of the `java.nio.file` package, which provides support for identifying and working with files and directories. The `Path` interface is used to represent the path of a file or a directory that is located on a file system. A path comprises the name of the file, directory/directories, and drive. For example, `D:\Java\Sample\explorer.txt` represents the path where `explorer.txt` is the name of the file, `Java\Sample\` are the names of directories, and `D:\` is the name of the drive in the file system.

In order to work with a path, you need to create a reference of the `Path` interface. To obtain a `Path` reference, you need to use the `get()` method of the `Paths` helper class in the `java.nio.file` package. For example, to obtain a `Path` reference, you can use the following code snippet:

```
    Path pathobject = Paths.get(“D:\Java\Sample\explorer.txt”);
```

In the preceding code snippet, a `Path` reference named `pathobject` is created. This reference represents the path, `D:\Java\Sample\explorer.txt`.

Further, the `Path` interface provides various methods, such as `getFileName()` to obtain information on paths of files or directories. The following table lists some of the most commonly used methods of the `Path` interface.

Method Name	***Description***	***Example***

`Path getFileName()`	*Returns the name of the file.*	*`Path pathobj = Paths.get("D:/NIODemo/Hello.java");`*
		`System.out.println(pathobj.getFileName());`
		The output of the preceding code snippet is:
		`Hello.java`

Method Name	***Description***	***Example***

`FileSystem getFileSystem()`	*Returns the name of the file system.*	`Path target=Paths.get("D:/Hello.java");`
		`System.out.println(target.getFileSystem());`
		The output of the preceding code snippet is:
		`sun.nio.fs.WindowsFileSystem@61316264`
		The digits after @ can vary.

`int getNameCount()`	*Returns the count of the number of elements that constitute the path, excluding the root drive letter, such as* `D:/`*.*	`Path target=Paths.get("D:/NIODemo/Hello.java");`
		`System.out.println(target.getNameCount());`
		The output of the preceding code snippet is:
		`2`

`Path getName(int index)`	*Returns the name element of the path as specified by the index value. An index value of* `0` *specifies the name that is closest to the root. An index value of count-1 specifies the name that is farthest from the root.*	`Path target=Paths.get("D:/NIODemo/Hello.java");`
		`System.out.println(target.getName(0));`
		The output of the preceding code snippet is:
		`NIODemo`

`Path getParent()`	*Returns the path where the file or directory is located. If a path is not specified or if the path does not have a parent, then* `null` *is returned.*	`Path target=Paths.get("D:/NIODemo/Hello.java");`
		`System.out.println(target.getParent());`
		The output of the preceding code snippet is:
		`D:\NIODemo`

`Path getRoot()`	*Returns the root drive component of the file, such as* `D:/`*.*	`Path target=Paths.get("D:/NIODemo/Hello.java");`
		`System.out.println(target.getRoot());`
		The output of the preceding code snippet is:
		`D:\`

Method Name	*Description*	*Example*

`boolean isAbsolute()`	*Returns* `true` *if the path is an absolute path. An absolute path represents the entire path hierarchy to locate a path.*	`Path target=Paths.get("D:/NIODemo/Hello.java");`
		`System.out.println(target.isAbsolute());`
		The output of the preceding code snippet is:
		`True`

`int compareTo(Path other)`	*Returns the result of a comparison of two paths. It returns* `0` *if the two paths are equal. It returns a value greater than* `0` *if the path specified is greater than the path specified as an argument. It returns a value that is less than* `0` *if the path specified is lesser than the path specified as an argument.*	`Path target=Paths.get("D:/NIODemo/Hello.java");`
		`Path comparePath=Paths.get("D:/NIODemo/Hello.java");`
		`Path compareNewPath=Paths.get("D:/NIODemo/NIO/Hello.java");`
		`System.out.println(comparePath.compareTo(target));`
		`System.out.println(compareNewPath.compareTo(target));`
		The output of the preceding code snippet is:
		`0`
		`6`

The Methods of the Path Interface

Note

In order to get the same output, as shown in the preceding table, ensure that the files used in the preceding code snippets are created at their respective locations.

Manipulating Files and Directories

Consider a scenario of a Java application where you need to perform various operations with the files and directories, such as searching a file, checking the existence of a file or directory or creating or deleting files or directories. To cater to the preceding requirement, you can use the methods of the `Files` class located in the `java.nio.file` package. In addition, in order to traverse files and directories, you need to use the `FileVisitor` interface.

Some of the file operations that can be performed using the `java.nio.file.Files` class are:

- Create a file or directory.
- Copy a file or directory.
- Move a file or directory.
- Check the existence of a file or directory.
- Delete a file or directory.
- Traverse a file tree.

Creating a File or Directory

To create a file, you need to use the `createFile()` method. The signature of the `createFile()` method is:

```
static Path createFile(Path path, FileAttribute<?>... attrs)
```

This method accepts a `Path` reference and creates a new file at the location specified by the `Path` reference variable. The `FileAttribute` reference is used to specify the attributes that define whether the file is being created for the purpose of reading and/or writing and/or executing. These attributes are file system dependent. Therefore, you need to utilize a file system-specific file permissions class and its helper class. For example, for a POSIX compliant file system, such as Unix, you can use the `PosixFilePermission` enum and the `PosixFilePermissions` class, as shown in the following code snippet:

```
Path target = Paths.get("\Backup\MyStuff.txt");
Set<PosixFilePermission> perms
    = PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>> attr
    = PosixFilePermissions.asFileAttribute(perms);
Files.createFile(target, attr);
```

In the preceding code, `rw-rw-rw-` specifies the permissions, which are applied on the `MyStuff.txt` file when it is created. If the file location does not exist, the `java.nio.file.NoSuchFileException` exception is thrown. In addition, if you try to create a file that already exists at the same location, the `java.nio.file.FileAlreadyExistsException` exception is raised. However, it is not mandatory to specify the file attributes. The following code snippet shows how to create a file without specifying the file attributes:

```
Path pathObject = Paths.get("D:/NIODemo.java");
Files.createFile(pathObject);
```

In the preceding code snippet, `pathObject` is a reference variable of the `Path` interface. `pathObject` stores the path returned by the `get()` method. The `pathObject` reference variable is passed to the `createFile()` method. The `createFile()` method creates a file named `NIODemo.java` in the D:\ drive.

Similarly, to create a directory, the `createDirectory()` method is used. The signature of the `createDirectory()` method is:

```
static path createDirectory(Path dir, FileAttribute<?>... attrs)
```

This method accepts a `Path` reference and creates a new directory at the location specified by the path reference variable. The `FileAttribute` reference is used to specify the attributes that define whether the directory is being created for the purpose of reading and/or writing and/or executing. It is not mandatory to specify the preceding file attributes. If you try to create a directory that already exists at the same location, then `java.nio.file.FileAlreadyExistsException` is raised.

The following code snippet shows how to create a directory without specifying the file attributes:

```
Path pathObject = Paths.get("D:/NIO");
Files.createDirectory(pathObject);
```

In the preceding code snippet, `pathObject` is a reference variable of the `Path` interface. `pathObject` stores the path returned by the `get()` method. The `pathObject` reference variable is passed to the `createDirectory()` method. The `createDirectory()` method creates a directory named `NIO` in the D:\ drive.

If there is a requirement to create a single level or multiple levels of directories, such as D:\User1\Java\Programs, you need to specify the relevant path and use the `createDirectories(Path dir, FileAttribute<?>... attrs)` method. This method does not throw an exception if the directories already exist.

The following code snippet shows how to create multiple directories by using the `createDirectories()` method:

```
Path newdir = Paths.get("D:/User1/Java/Programs");
        try
          {
                   Files.createDirectories(newdir);
          }
       catch (IOException e)
          {
            System.err.println(e);
          }
```

In the preceding code snippet, the `createDirectories()` method creates the directories as specified by the path reference, `newdir`.

Copying a File or Directory

To copy a file, you need to use the `copy()` method of the `Files` class. The signature of the `copy()` method is:

```
public static Path copy(Path source, Path target, CopyOption... options)
```

This method accepts two `Path` references for the source file to be copied and the target file to be created as a copy of the source file, respectively. Further, `CopyOption... options` is used to specify one or more copy options that determine how the copy process should be done. It is not mandatory to specify the copy options. These options are listed under the `StandardCopyOption` and `LinkOption` enums.

The following table lists the commonly used copy options that can be specified with the `copy()` method.

Enum	***Option***	***Description***
StandardCopyOption	*REPLACE_EXISTING*	*This option allows performing the copy process even if the specified target file already exists. Otherwise, if this option is not specified and the target file already exists, an exception is thrown.*
StandardCopyOption	*COPY_ATTRIBUTES*	*This option allows copying the file attributes associated with the source file to the target file. The file attributes define the characteristics of a file, such as its creation time and last modification time.*
LinkOption	*NOFOLLOW_LINKS*	*This option allows you to specify that the symbolic links should not be followed to their actual target file. A symbolic link is a file that contains a reference to another target file or directory.*

The Copy Options Used with the copy() Method

The following code snippet shows how to copy a file:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardCopyOption.*;

public class NIOdemo
{
    public static void main(String[] args)
    {
        Path source = Paths.get("D:/Hello.java");
        Path target=Paths.get("D:/NIODemo/Hello.java");
        try
        {
            Files.copy(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
        }
        catch(IOException e)
        {
        System.out.println(e);
        }
    }
}
```

In the preceding code snippet, the source file to be copied is specified by the `source` path reference and the file to be created with the copy process is specified at the `D:/NIODemo/Hello.java` location. If the source file to be copied does not exist, the java.nio.file.NoSuchFileException exception is thrown. In addition, if a target file already exists and the REPLACE_EXISTING option is not specified, the FileAlreadyExistsException exception is thrown.

Moving a File or Directory

To move an existing file or directory, you can use the move() method specified in the Files class. The signature of the move() method is:

```
public static Path move(Path source, Path target, CopyOption... options)
```

This method accepts two Path references for the source file or directory to be moved and the target file or directory to which the source file or directory should be moved, respectively. However, if the specified target file already exists, the move process fails. Further, CopyOption... options is used to specify one or more copy options that determine how the copy or move process should be done. These options are listed under the StandardCopyOption enum. The following table lists the copy options that can be specified with the move() method.

Option	*Description*
REPLACE_EXISTING	*This option allows performing the move process even if the specified target file already exists.*
ATOMIC_MOVE	*This option is used to perform the move process as an atomic operation. An atomic operation cannot be interrupted in between.*

The Copy Options Used with the move() Method

The following code shows how to move an existing file to another location:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardCopyOption.*;

public class NIOdemo {
    public static void main(String[] args)
{
        Path source = Paths.get("D:/NIODemo/Hello.java");
        Path target=Paths.get("D:/NIODemo/NIO/Hello.java");
        try
        {
                    Files.move(source, target, REPLACE_EXISTING);
        }
        catch(IOException e)
        {
        System.out.println(e);
        }
}
}
```

In the preceding code, the Hello.java file located in the D:/NIODemo directory is moved to the D:\NIODemo\NIO directory. If the source file does not exist at the specified location or if an invalid target location is specified, the NoSuchFileException exception is thrown.

The following code snippet shows how to move an existing directory to another directory:

```
Path source = Paths.get("D:/Pictures");
        Path target=Paths.get("D:/NIODemo/NIO/NewEmptyDirectory");
        Files.move(source, target, REPLACE_EXISTING);
```

When the preceding code snippet is executed, the contents specified at the D:\Pictures path will be moved to the NewEmptyDirectory directory. While moving a directory, it is important to ensure that the target directory is empty. Otherwise, the DirectoryNotEmptyException exception is thrown. If the target directory location does not exist at the specified location, the NoSuchFileException exception is thrown.

Checking the Existence of a File or Directory

To check whether a file or directory exists or it does not exist, you can use the exists() and notExists() methods of the Files class, respectively. The exists() method tests whether a file exists. The signature of the exists() method is:

```
public static boolean exists(Path path, LinkOption... options)
```

This method accepts a path reference, path, which is to be verified for its existence. Further, the link options are used to specify how the symbolic links should be handled if the path represents a symbolic link. This method returns true if the file or directory exists. Otherwise, it returns false. The following code snippet shows how to use the exists() method:

```
Path target=Paths.get("D:/NIODemo/NIO/NewEmptyDirectory");
Boolean pathExists = Files.exists(target, LinkOption.NOFOLLOW_LINKS);
System.out.println(pathExists);
```

In the preceding code snippet, the path, D:/NIODemo/NIO/NewEmptyDirectory, is verified for its existence.

Similarly, you can use the notExists() method to check if a file or directory does not exist. This method returns true if a file or directory does not exist. Otherwise, it returns false. The following code snippet shows how to use the notExists() method:

```
Path target=Paths.get("D:/NIODemo/NIO/NewEmptyDirectory");
Boolean pathExists = Files.notExists(target, LinkOption.NOFOLLOW_LINKS);
System.out.println(pathExists);
```

In the preceding code snippet, the path, D:/NIODemo/NIO/NewEmptyDirectory, is verified for its non existence.

Deleting a File or Directory

To delete a file, directory, or symbolic link, you can use either the delete(Path path) method or the deleteIfExists(Path path) method. The signature of the delete() method is:

```
public static void delete(Path path)
```

The preceding method accepts a Path reference that specifies the path to be deleted. The following exceptions may be thrown by this method:

- NoSuchFileException: Is thrown if the path does not exist.
- DirectoryNotEmptyException: Is thrown if the path specifies a directory that is not empty.
- IOException: Is thrown in case of an I/O error.
- SecurityException: Is thrown if the permission to delete the file is denied.

The following code snippet shows how to delete a file:

```
Path target=Paths.get("D:/NIODemo/NIO/Hello.java");
Files.delete(target);
```

Once the preceding code snippet is executed, the Hello.java file that is located in the D:\NIODemo\NIO directory is deleted.

Similarly, you can use the deleteIfExists() method for deleting a file or a directory. The signature of this method is:

```
public static boolean deleteIfExists(Path path)
```

The preceding method accepts a path reference that specifies the path to be deleted. However, this method does not throw an exception in case the specified path does not exist. In such a case, it returns a boolean value, false. The following code snippet shows how to use the deleteIfExists() method:

```
Path target=Paths.get("D:/NIODemo/NIO/Hello.java");
System.out.println(Files.deleteIfExists(target));
```

Once the preceding code snippet is executed, the Hello.java file located in the D:\NIODemo\NIO directory is deleted if it exists and the appropriate boolean value is printed.

Traversing a File Tree

Consider a scenario where you need to create a Java application that should traverse all the folders and files located in your directory structure to perform an operation such as, create, find, or delete a file. Further, you may want to recursively copy the contents of a directory to another directory. To meet the preceding requirements, you can use the FileVisitor interface provided by NIO. This interface is located in the java.nio.file package.

The FileVisitor interface provides support to traverse a file tree recursively. It also allows you to gain control over the traversal process and perform operations during the traversal process. In order to perform operations during the traversal process, this interface provides some methods. You need to override the following methods to perform operations during the traversal process:

- FileVisitResult preVisitDirectory(): This method is used with a directory and is invoked before visiting its contents.
- FileVisitResult postVisitDirectory(): This method is used with a directory and is invoked after all its contents are visited.
- FileVisitResult visitFile(): This method is used with a file and is invoked when that file is visited during the traversal process.
- FileVisitResult visitFileFailed(): This method is used with a file and is invoked when the file to visit cannot be accessed.

Each of the preceding methods returns a value of the type, the FileVisitResult enum. This value is used to determine how to proceed with the traversal process. Some of the constant values in the FileVisitResult enum are:

- CONTINUE: Is used to indicate that the traversal must continue.
- SKIP_SUBTREE: Is used to indicate that the traversal must proceed without visiting the contents inside the directory.
- TERMINATE: Is used to indicate that the traversal must terminate.

Further, to start the traversal process, you need to use the walkFileTree() method in the Files class. This method accepts the path from where the traversal process needs to be started and the instance of the class that implements the FileVisitor interface. The signature of this method is:

```
public static Path walkFileTree(Path startpoint, FileVisitor<? super Path>
visitor)
```

In the preceding signature, startpoint represents the path reference from which the traversal should begin. FileVisitor<? super Path> visitor represents the instance of the class that implements the FileVisitor interface.

The following code shows how to implement the file tree traversal process by using the FileVisitor interface:

```
import java.io.IOException;
import java.nio.file.FileVisitResult;
import java.nio.file.FileVisitor;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.BasicFileAttributes;

class MyFileVisitor implements FileVisitor<Path> {

    public FileVisitResult postVisitDirectory(Path dir, IOException e) throws
IOException {
        System.out.println("Just Visited " + dir);
        return FileVisitResult.CONTINUE;
    }


    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes
attrs) throws IOException {

        System.out.println("About to visit " + dir);
                return FileVisitResult.CONTINUE;

    }

        public FileVisitResult visitFile(Path file, BasicFileAttributes
attrs) throws IOException {

        System.out.println("Currently visiting "+file);
        System.out.println("Is this file a directory: "+
Files.isDirectory(file));
```

```
    System.out.println("Checking done..!!");
    return FileVisitResult.CONTINUE;

  }

    public FileVisitResult visitFileFailed(Path file, IOException e)
throws IOException {
    System.err.println(e.getMessage());
    return FileVisitResult.CONTINUE;
  }
}
public class NIODemo
{
  public static void main(String a[]) throws IOException
  {
Path listDir = Paths.get("D:/NIO");
MyFileVisitor obj = new MyFileVisitor();
Files.walkFileTree(listDir, obj);
  }
}
```

The output of the preceding code is:

```
About to visit D:\NIO
About to visit D:\NIO\Hello
Just Visited D:\NIO\Hello
Currently visiting D:\NIO\Hello.txt
Is this file a directory: false
Checking done..!!
Just Visited D:\NIO
```

Note

The preceding output may vary based on contents of the `D:/NIO` directory.

In the preceding code, `listDir` represents a path reference from where the traversal should begin. Then, the `walkFileTree()` method is used to start the traversal from the path specified by `listDir` by using the `obj` instance of the `MyFileVisitor` class. In this class, the methods of the `FileVisitor` interface are overridden. The `postVisitDirectory()` method prints the `Just Visited` message and the file or directory name. The `preVisitDirectory()` method prints the `About to visit` message and the file or directory name. The `visitFile()` method prints the name of the currently visited file and also checks if the visited path is a directory by using the `Files.isDirectory()` method. Further, the `visitFileFailed()` method is used to print a message if the file visit fails.

However, there can be a requirement where you do not need to provide an implementation for all the methods of the `FileVisitor` interface, but only for a few methods. In such a case, you can extend the `SimpleFileVisitor` class. For example, when there is a requirement to print only the names of the files in a file tree when they are visited. The `SimpleFileVisitor` class implements the `FileVisitor` interface, and therefore, its methods can be overridden depending upon the requirement of the application.

Consider a scenario, where you need to perform a pattern match to obtain a list and count of filenames that match a pattern, such as *.java. Therefore, to match a filename with the pattern name, you can use the `PathMatcher` class. The following code demonstrates how to search for all the .java files in the D:\ drive of your computer:

```
import java.io.IOException;
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.FileVisitOption;
import java.nio.file.FileVisitResult;
import java.nio.file.FileVisitor;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.EnumSet;

class SearchDemo implements FileVisitor {

    private final PathMatcher matcher;
    int counter = 0;

    public SearchDemo(String pattern) {
        FileSystem fs=FileSystems.getDefault();
        matcher = fs.getPathMatcher("glob:" + pattern);
    }

    void search(Path file) throws IOException {
        Path name = file.getFileName();

        if (name != null && matcher.matches(name)) {

            System.out.println("Searched file located: " + name + " in " +
file.getParent().toAbsolutePath());
            counter++;
        }

    }


    public FileVisitResult postVisitDirectory(Object dir, IOException exc)
            throws IOException {

        return FileVisitResult.CONTINUE;
    }


    public FileVisitResult preVisitDirectory(Object dir, BasicFileAttributes
attrs)
            throws IOException {
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFile(Object file, BasicFileAttributes attrs)
            throws IOException {
        search((Path) file);
        return FileVisitResult.CONTINUE;
    }


    public FileVisitResult visitFileFailed(Object file, IOException exc)
            throws IOException {
        return FileVisitResult.CONTINUE;
    }
}

class MainClass {

    public static void main(String[] args) throws IOException {
        String pattern = "*.java";
        Path fileTree = Paths.get("D:/");
        SearchDemo walk = new SearchDemo(pattern);
        EnumSet opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);
        Files.walkFileTree(fileTree, opts, Integer.MAX_VALUE, walk);
        System.out.println("Total files found: " + walk.counter);
    }
}
```

The output of the preceding code is:

```
Searched file located: Hello.java in D:\NIODemo
Searched file located: NIODemo - Copy.java in D:\
Searched file located: NIODemo.java in D:\
Searched file located: ThreadOutput.java in D:\PPS
Searched file located: Puzzle.java in D:\
Total files found: 5
```

Note
The preceding output may vary.

In the preceding code, in the `main()` method, the pattern is specified as `*.java`. In the constructor, the `getPathMatcher()` method is used. This method is used to create a `PathMatcher` for performing match operations on paths. The `FileSystems.getDefault()` method returns an object of the `FileSystem` class. The `getPathMatcher()` method is used to obtain the object of the `PathMatcher` class. This method is accepting a glob pattern, such as glob:*.java. Further, the `matches()` method is used to compare the name of the browsed files against the glob pattern. By using the `counter` variable, the total files that match the pattern are displayed inside the `main()` method.

Note

A glob pattern is specified as a string and used to specify a pattern to be matched against strings, such as file names.

Just a minute:

The______________ constant of the `FileVisitResult` enum is used to indicate that the traversal must end.

Answer:

`TERMINATE`

Implementing Watch Service

Consider a scenario where you need to develop a Java application to monitor a directory and keep track of the changes that are performed on the directory. For example, you may want to track a directory and get notified if any file is created, deleted, or modified from that directory. To accomplish these tasks, you can use the Watch Service API, which is a part of `java.nio.file`. This API is used to implement the file change notification functionality that enables a Java application to monitor and detect the changes that are made to a directory.

The watch service can be implemented by performing the following steps:

1. Create a watch service.
2. Register the object to be monitored with the watch service.
3. Wait for the events to occur.
4. Retrieve the watch key.
5. Retrieve the pending events for the key.
6. Reset the key.
7. Close the watch service.

Creating a Watch Service

A watch service is used to monitor the objects and detect when an object is modified. You can obtain a reference of `WatchService` by using the following code snippet:

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

Registering the Object to be Monitored with the Watch Service

Once the watch service is created, you need to register the object that you want to watch for modifications with the watch service. Any object that implements the `Watchable` interface can be watched for changes and events. The `Path` class implements the `Watchable` interface, which allows you to register a directory to monitor changes. To register an object, the `Watchable` interface provides the `register()` method. The signature of the `register()` method is:

```
WatchKey register(WatchService watcher, WatchEvent.Kind<?>... events)
```

In the preceding signature, `watcher` refers to the watch service created by using the `newWatchService()` method. `WatchEvent.Kind<?>... events` specifies the events for which the watch service should monitor and provide a notification. These events can be specified using the constant values defined in the `StandardWatchEventKinds` class. The following table lists some of the constants of the `StandardWatchEventKinds` class.

Methods	***Description***
`StandardWatchEventKinds.ENTRY_CREATE`	*A directory entry is created.*
`StandardWatchEventKinds.ENTRY_DELETE`	*A directory entry is deleted.*
`StandardWatchEventKinds.ENTRY_MODIFY`	*A directory entry is modified.*

The StandardWatchEventKinds Class Constants

The `register()` method returns a `WatchKey` object as a registration token for each registered directory. A watch key is created when a `Watchable` object is registered with a watch service. Once registered, a directory's corresponding `WatchKey` is said to be in the `ready` state. In the ready state, `WatchKey` can accept events.

The following code snippet is used to accept a registration token:

```
WatchKey key = path.register(watchService,
StandardWatchEventKinds.ENTRY_CREATE,StandardWatchEventKinds.ENTRY_MODIFY,
StandardWatchEventKinds.ENTRY_DELETE);
```

In the preceding code snippet, the registration token is accepted in `key` when `path` is registered with the watch service.

Waiting for the Events to Occur

Because an event in the directory may occur anytime, it is necessary to keep a watch over it at all times. This requires monitoring the directory in an infinite loop, as shown in the following code snippet:

```
while(true)
{
//retrieve and process the incoming events
…
}
```

Retrieving the Watch Key

When an event occurs in a directory, the state of its corresponding watch key changes to `signaled`. The corresponding watch key is placed in a queue of the watch service from where it can be retrieved.

The following table lists the methods of the `WatchService` class that can be used to retrieve a queued key.

Method	***Description***
`WatchKey take()`	*This method returns a queued key and waits until a key is queued or the infinite loop is stopped.*
`WatchKey poll()`	*This method returns a queued key if it is available, else it returns with a `null` value.*
`WatchKey poll(long timeNum, TimeUnit)`	*This method returns a queued key if it is available, else it waits for the time period indicated by `timeNum` and then tries again. `TimeUnit` specifies the unit of time, such as nanoseconds or milliseconds.*

The Methods to Retrieve a Queued Key

The following code snippet shows how to retrieve a queued key in an infinite loop:

```
while (true)
{
    //retrieve and remove the next watch key
    final WatchKey key = watchService.poll();
    //the thread flow gets here immediately with an available key or a null
    value
}
```

Retrieving the Pending Events for the Key

When a key is in a `signaled` state and another event occurs, then the events are queued up for the corresponding key. The pending events for a corresponding key can be retrieved and removed by using the `pollEvents()` method of the `WatchKey` interface. This method returns a list of the pending events for an object of the type, `WatchEvent`. The list can be further iterated to process each event individually. The type of the list is `WatchEvent<T>`.

The following code snippet shows how to retrieve the pending events for a watch key:

```
while (true)
            {
                    final WatchKey key = watchService.poll();
                    //   process events of the watch key
                    for (WatchEvent<?> watchEvent : key.pollEvents())
                            {
                            // code for processing the events
                            }
            }
```

In the preceding code snippet, the `for` loop is used to retrieve the pending events of the `key` watch key.

Further, if you want to obtain more information about the event, you can use the methods of the `WatchEvent<T>` interface.

The following table lists the methods of the `WatchEvent<T>` interface.

Method	***Description***
`WatchEvent.Kind<T> kind()`	*This method returns the kind of event that occurred.*
`T context()`	*This method returns the context of the event. For example, when a directory entry is created, deleted, or modified, the context is the entry that was created, deleted, or modified.*

The Methods of the WatchEvent<T> Interface

The following code snippet shows how to use the `kind()` method:

```
for (WatchEvent<?> watchEvent : key.pollEvents())
{
   Kind<?> eventKind = watchEvent.kind(); //determine the kind of the event
}
```

In the preceding code snippet, `eventKind` stores the value returned by the `kind()` method.

Resetting the Key

To ensure that a watch key can be used for watching events, you need to reset the watch key if it is in a `signaled` state. The `reset()` method returns the key to the `ready` state. This method returns `true` if the key is valid, else it returns `false`. The following code snippet shows how to reset a key:

```
while(true)
{
    final WatchKey key = watchService.poll();

// code to get event kind
    boolean valid = key.reset();
// code to break
}
```

Closing the Watch Service

Finally, when the watch service is no longer required, you can close it by using the `close()` method of the `WatchService` interface or ensure that the watch service was created in a `try`-with-resources block so that it can be closed automatically, when it is no longer required. The following code snippet shows how to close the watch service by using the `close()` method:

```
watchService.close();
```

The following code shows how to implement a watch service:

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.WatchEvent.*;
import java.nio.file.WatchKey;
import java.nio.file.WatchService;
import static java.nio.file.StandardWatchEventKinds.*;
import java.nio.file.WatchEvent;

public class WatchServiceDemo {
    private Path path = null;
    WatchService watchService;
     private void initializeService() {
        path = Paths.get("D:/NIO");  // the path to the directory to be
monitored
       try {
            watchService = FileSystems.getDefault().newWatchService();
            path.register(watchService, ENTRY_CREATE,
ENTRY_DELETE,ENTRY_MODIFY);   // register the watch service for the path
}
catch (IOException e) {
            System.out.println("IOException"+ e.getMessage());
        }
    }

    private void monitorDirectory() {
        WatchKey key = null;
        while(true) {  // infinite loop to monitor changes
            try {
                key = watchService.take(); // get watch key
                for (WatchEvent event : key.pollEvents()) {
                    Kind kind = event.kind(); // get event kind
                    System.out.println("The event that occurred on " +
event.context().toString() + " is " + kind);
                }
            } catch (InterruptedException e) {
                System.out.println("InterruptedException: "+e.getMessage());
            }
            boolean reset = key.reset();
            if(!reset)
            break;
        }
    }

    public static void main(String[] args) {
        WatchServiceDemo watchservicedemo = new WatchServiceDemo();
        watchservicedemo.initializeService();
        watchservicedemo.monitorDirectory();
    }
}
```

In the preceding code, a new watch service named `watchService` is created. The path to be monitored for changes is `D:/NIO`. Then, this directory path is registered to be monitored with `watchService` for the create, delete, and modify events. Further, an infinite loop is created in the `monitorDirectory()` method to monitor the directory constantly for any modifications. In this method, the events are retrieved and the information related to the kind of the event that has occurred is printed.

Upon execution, when you create a new text document named `WatchDemo`, the creation is detected and the following output is displayed:

```
The event that occurred on WatchDemo.txt is ENTRY_CREATE
```

Next, when you modify the contents of the `WatchDemo.txt` file and save it, this modification is detected and the following output is displayed:

```
The event that occurred on WatchDemo.txt is ENTRY_MODIFY
```

Just a minute:

The _____ state represents the state of the watch key when a directory is registered.

Answer:

ready

Performing the Read and Write Operations on a File

Consider the scenario of an application that reads or writes a large amount of data frequently. As the read and write operations require to utilize the system resources, such as the processor and a storage medium like a hard disk, it is essential that the application performs those read and write operations by using an efficient approach. Further, it is essential to minimize the time that is consumed for the preceding operations. One of the ways to achieve an efficient performance and improve the speed of the read and write operations is the use of buffers. A *buffer* is a storage area in the memory that can be used for writing or reading data. As compared to streams, buffers can be used to provide faster I/O operations.

As compared to the traditional I/O operations, you can improve the speed of the read and write operations performed by an application, such as store text in a file located on the hard disk of your computer by using NIO. In addition, the use of NIO reduces code complexity.

Reading a File

Consider a scenario where you need to create a Java application to efficiently read the contents of a file and display them in the console window. To read the file, you can use the `newBufferedReader()` method defined in the `java.nio.file.Files` class.

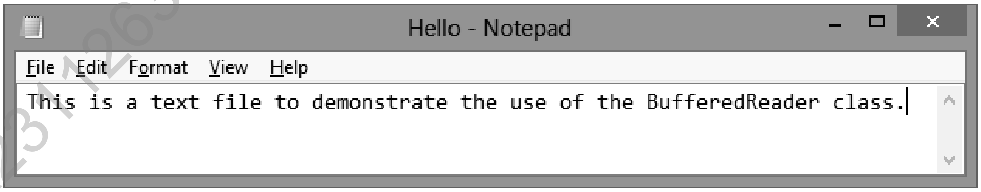
This method returns a `BufferedReader` object that can be used for the purpose of efficiently reading text from the file.

The following code snippet shows how to create a `BufferedReader` reference variable for reading a file:

```
BufferedReader name = Files.newBufferedReader(path, charset);
```

In the preceding code snippet, `name` specifies the reference of the `BufferedReader` class, `path` is the reference variable of `Path` interface, which specifies the path of the file stored, and `charset` is the reference variable of the `Charset` class, which specifies the type of charset, such as `US-ASCII`, to be used for converting the bytes of a file to Unicode characters. The `newBufferedReader()` method throws `IOException` if an I/O error occurs.

Consider a scenario where you need to read the contents of a file named `Hello.txt` located in the `D:\` drive of your computer. The content of the `Hello.txt` file is shown in the following figure.



```
This is a text file to demonstrate the use of the BufferedReader class.
```

The Content of the Hello.txt File

You need to develop a program that reads characters from the preceding file by using the `BufferedReader` class and display them. You can use the following code to implement the preceding functionalities:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class BufferedReaderDemo {
  public static void main(String a[]) {
        Path pathObject = Paths.get("D:/Hello.java");
        Charset charset = Charset.forName("US-ASCII");
        try (BufferedReader reader = Files.newBufferedReader(pathObject,
charset)) {
            String line = null;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException x) {
            System.out.println(x);
        }
    }
}
```

The output of the preceding code is:

```
This is a text file to demonstrate the use of the BufferedReader class.
```

In the preceding code, `pathObject` is created that represents the location of the file to be read. Then, a `BufferedReader` reference variable named `reader` is created. In the `while` loop, the `readLine()` methods reads a line of text and assigns the text to a `String` reference variable named `line` and it is displayed until the end of file is reached.

Writing to a File

Consider a scenario where you need to develop a Java application to write the contents in a file. To perform the preceding task, you can use the `newBufferedWriter()` method defined in the `java.nio.file.Files` class. The `newBufferedWriter()` method creates a `BufferedWriter` object. If the file already exists, then the file is opened for writing. The `write()` method of the object is used to create a file for writing if it does not exist.

The following code snippet shows how to create a `BufferedWriter` reference variable for writing to a file:

```
BufferedWriter name = Files.newBufferedWriter(path, charset, options);
```

In the preceding code snippet, `name` is a `BufferedWriter` reference variable. `path` is the reference variable of the `Path` interface, which specifies the path of the file stored. `charset` is the reference variable of the `Charset` class, which specifies the type of charset, such as `US-ASCII`, to be used for converting the bytes of a file to Unicode characters. The `options` parameter is used to specify how the file is to be used for the write operation. You can specify the options by using the `StandardOpenOption` enum.

The following table lists some of the standard options that can be specified for the write operation.

Option	***Description***
`StandardOpenOption.WRITE`	*This option opens a file for write access.*
`StandardOpenOption.APPEND`	*This option writes to the end of a file, if the file is opened for write access.*
`StandardOpenOption.CREATE`	*This option creates a new file if it does not exist.*
`StandardOpenOption.CREATE_NEW`	*This option creates a new file but throws an exception if the file with the same name already exists.*
`StandardOpenOption.DELETE_ON_CLOSE`	*This option attempts to delete a file after it is closed.*
`StandardOpenOption.TRUNCATE_EXISTING`	*If the file already exists and is open for write access, then its length is truncated to zero.*

The Various Options of the StandardOpenOption Enum

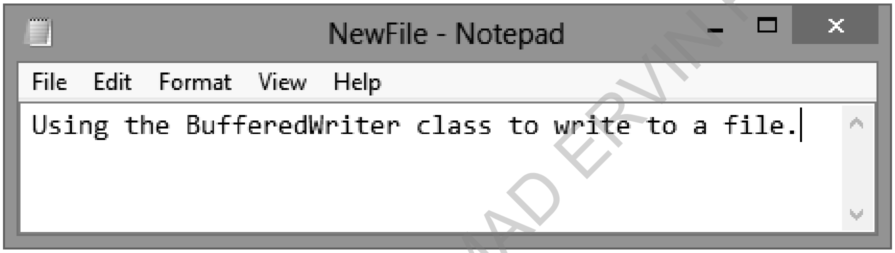
`newBufferedWriter()` throws `IOException` if an I/O error occurs while opening or creating the file. It also throws an `UnsupportedOperationException` exception if an unsupported option is specified while opening the file.

Consider a scenario where you need to develop a Java program to create a file named NewFile.txt in the D:\ drive of your computer with the text using the `BufferedWriter` class to write to a file. You can use the following code to achieve the preceding requirements:

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class BufferedWriterDemo
{
    public static void main(String[] args)
   {
        String content = "Using the BufferedWriter class to write to a 
file.";
        Path file = Paths.get("D:/NewFile.txt");
        Charset charset = Charset.forName("US-ASCII");
        try (BufferedWriter writer = Files.newBufferedWriter(file, charset, 
StandardOpenOption.CREATE))
        {
            writer.write(content);
            System.out.println("Done");
        } catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```

In the preceding code, the `file` reference variable is created that represents the location of the file to be written. Then, a `BufferedWriter` reference variable named `writer` is created. The `write()` method writes the value of the `String` reference variable named `content` to `NewFile.txt`.

Once the preceding code is executed, the `NewFile.txt` file is created. The content of the `NewFile.txt` file is shown in the following figure.



Using the BufferedWriter class to write to a file.

The Content of the NewFile.txt File

In the preceding code, if the `try (BufferedWriter writer = Files.newBufferedWriter(file, charset, StandardOpenOption.CREATE))` statement is replaced with the `try (BufferedWriter writer = Files.newBufferedWriter(file, charset, StandardOpenOption.CREATE_NEW))` statement, then the following exception is generated:

```
java.nio.file.FileAlreadyExistsException: D:\NewFile.txt
```

The preceding exception is raised because a new file cannot be created as a file with the name, `NewFile.txt`, already exists at the specified location.

Activity 8.1: Performing File Operations Using NIO

Practice Questions

1. Which one of the following methods of the `Paths` helper class is used to obtain a file path reference?
    a. `get()`
    b. `getFileName()`
    c. `getName()`
    d. `getRoot()`

2. Identify the method of the `Path` interface that returns the name of the file system.
    a. `getFileName()`
    b. `getFileSystem()`
    c. `FileSystem()`
    d. `getParent()`

3. Identify the method of the `Path` interface that is used to identify if a path represents the entire path hierarchy.
    a. `getFileName()`
    b. `getFileSystem()`
    c. `getRoot()`
    d. `isAbsolute()`

4. Identify the exception that is thrown when you try to create a directory that already exists at the specified location while using the `createDirectory()` method.
    a. `FileAlreadyExistsException`
    b. `DirectoryNotEmptyException`
    c. `NoSuchFileException`
    d. `IOException`

5. Identify the interface that provides support for traversing a file structure recursively.
    a. `FileVisitor`
    b. `FileReader`
    c. `Watchable`
    d. `Path`

Summary

In this chapter, you learned that:

- The `java.nio` package provides sub packages and classes that are useful for improving the speed and performance of I/O operations.
- The `Path` interface is a component of the `java.nio.file` package, which provides support for identifying and working with files and directories.
- Some of the file operations that can be performed using the `java.nio.file.Files` class are:
  - Create a file or directory
  - Copy a file or directory
  - Move a file or directory
  - Check the existence of a file or directory
  - Delete a file or directory
  - Traverse a file tree
- To create a file, you need to use the `createFile()` method.
- To copy a file, you need to use the `copy()` method of the `Files` class.
- To move an existing file or directory, you can use the `move()` method specified in the `Files` class.
- To check whether a file or directory exists or it does not exist, you can use the `exists()` and `notExists()` methods of the `Files` class, respectively.
- To delete a file, directory, or symbolic link, you can use either the `delete(Path path)` method or the `deleteIfExists(Path path)` method.
- The `FileVisitor` interface provides support to traverse a file tree recursively.
- A watch service is used to monitor the objects and detect when an object is modified.
- Any object that implements the `Watchable` interface can be watched for changes and events.
- A buffer is a storage area in the memory that can be used for writing or reading data.
- To read the file, you can use the `newBufferedReader()` method defined in the `java.nio.file.Files` class.

Chapter 9

Introduction to JDBC

Java Database Connectivity (JDBC) is an API that executes SQL statements. It consists of a set of classes and interfaces written in the Java programming language. The interface is easy to connect to any database using JDBC. The combination of Java and JDBC makes the process of disseminating information easy and economical.

This chapter introduces the JDBC architecture. It gives an overview of the different types of drivers supported by JDBC. It elaborates the methods to establish the connection with SQL Server using the Type 4 JDBC driver.

Objectives

In this chapter, you will learn to:

- Identify the layers in the JDBC architecture
- Identify the types of JDBC drivers
- Use JDBC API
- Access result sets

Identifying the Layers in the JDBC Architecture

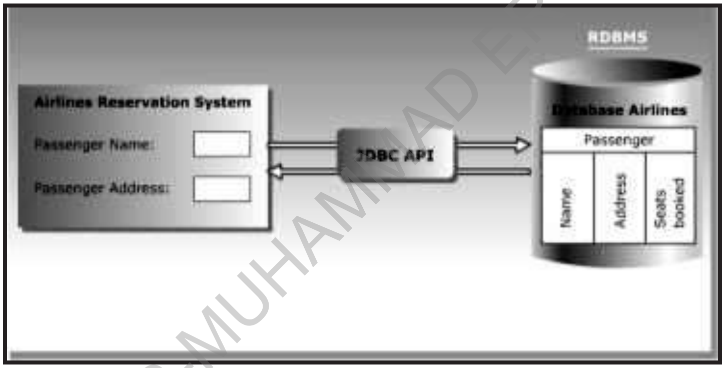
Consider a scenario where you have to develop an application for an airlines company that maintains a record of daily transactions. You install SQL Server as an RDBMS, design the airlines database, and ask the airlines personnel to use it. Will the database alone be of any use to the airlines personnel?

The answer will be negative. The task of updating data in the SQL Server database by using SQL statements alone will be a tedious process. An application will need to be developed that is user friendly and provides the options to retrieve, add, and modify data at the touch of a key to a client.

Therefore, you need to develop an application that communicates with a database to perform the following tasks:

- Store and update the data in the database.
- Retrieve the data stored in the database and present it to users in a proper format.

The following figure shows Airline Reservation System developed in Java that interacts with the Airlines database using the JDBC API.



The Database Connectivity Using the JDBC API

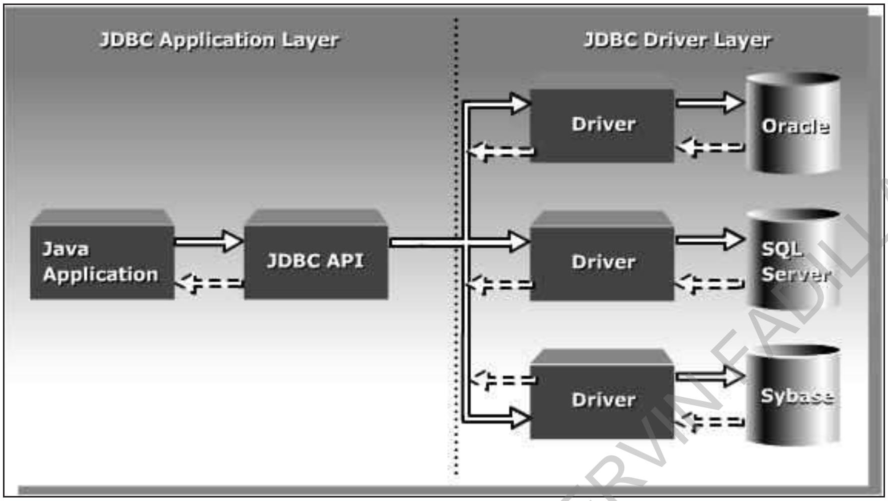
JDBC Architecture

Java applications cannot communicate directly with a database to submit data and retrieve the results of queries. This is because a database can interpret only SQL statements and not Java language statements. Therefore, you need a mechanism to translate Java statements into SQL statements. The JDBC architecture provides the mechanism for this kind of translation.

The JDBC architecture has the following two layers:

- **JDBC application layer**: Signifies a Java application that uses the JDBC API to interact with the JDBC drivers. A JDBC driver is the software that a Java application uses to access a database.
- **JDBC driver layer**: Acts as an interface between a Java application and a database. This layer contains a driver, such as the SQL Server driver or the Oracle driver, which enables connectivity to a database. A driver sends the request of a Java application to the database. Once this request is processed, the database sends the response back to the driver. The response is then translated and sent to the JDBC API by the driver. The JDBC API finally forwards this response to the Java application.

The following figure shows the JDBC architecture.



The JDBC Architecture

Just a minute:

Identify the two layers of the JDBC architecture.

Answer:

The two layers of the JDBC architecture are:

1. *JDBC application layer*
2. *JDBC driver layer*

Identifying the Types of JDBC Drivers

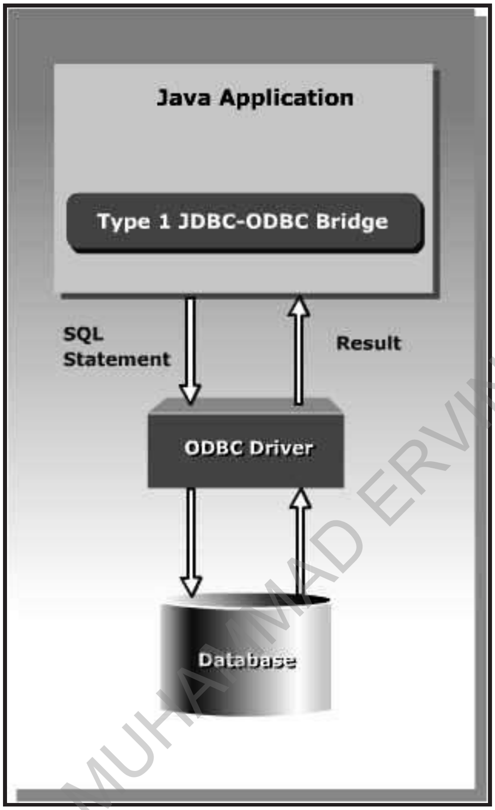
While developing JDBC applications, you need to use JDBC drivers to convert queries into a form that a particular database can interpret. The JDBC driver also retrieves the result of SQL statements and converts the result into equivalent JDBC API class objects that the Java application uses. Because the JDBC driver only takes care of the interactions with the database, any change made to the database does not affect the application. JDBC supports the following types of drivers:

- JDBC-ODBC Bridge driver
- Native-API driver
- Network Protocol driver
- Native Protocol driver

The JDBC-ODBC Bridge Driver

The JDBC-ODBC Bridge driver is called the Type 1 driver. The JDBC-ODBC Bridge driver converts the JDBC method calls into the *Open Database Connectivity (ODBC)* function calls. ODBC is an open standard API to communicate with databases. The JDBC-ODBC Bridge driver enables a Java application to use any database that supports the ODBC driver. A Java application cannot interact directly with the ODBC driver. Therefore, the application uses the JDBC-ODBC Bridge driver that works as an interface between the application and the ODBC driver. To use the JDBC-ODBC Bridge driver, you need to have the ODBC driver installed on the client computer. The JDBC-ODBC Bridge driver is usually used in standalone applications.

The following figure shows how the JDBC-ODBC Bridge driver works.

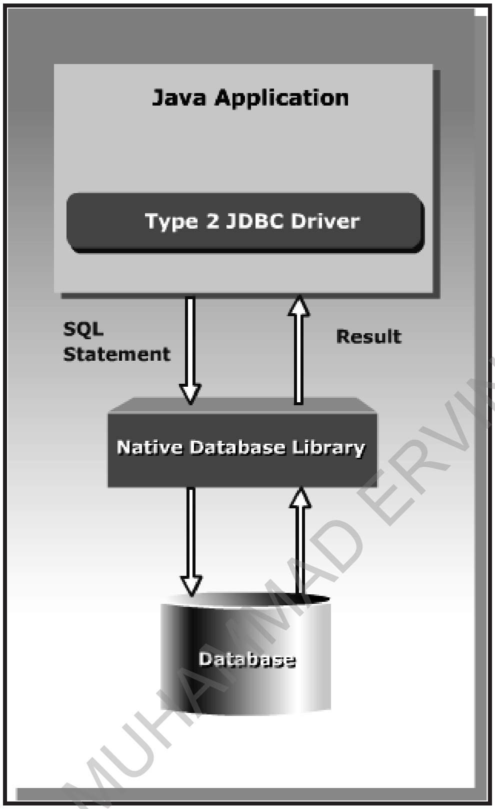


The JDBC-ODBC Bridge Driver

The Native-API Driver

The Native-API driver is called the Type 2 driver. It uses the local native libraries provided by the database vendors to access databases. The JDBC driver maps the JDBC calls to the native method calls, which are passed to the local native *Call Level Interface (CLI)*. This interface consists of functions written in the C language to access databases. To use the Type 2 driver, CLI needs to be loaded on the client computer. Unlike the JDBC-ODBC Bridge driver, the Native-API driver does not have an ODBC intermediate layer. As a result, this driver has a better performance than the JDBC-ODBC Bridge driver and is usually used for network-based applications.

The following figure shows how the Native-API driver works.

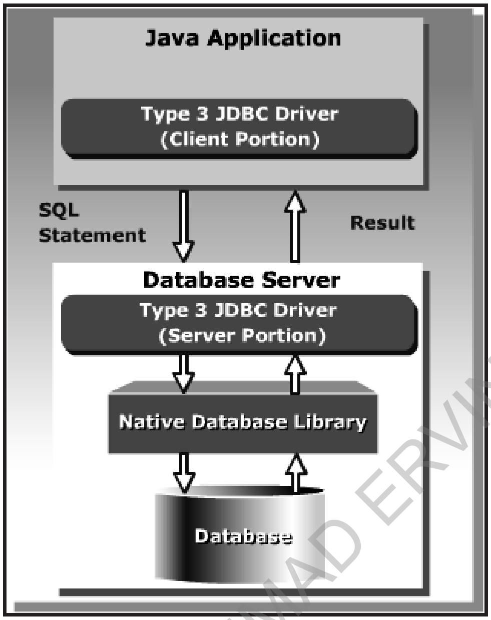


The Native-API Driver

The Network Protocol Driver

The Network Protocol driver is called the Type 3 driver. The Network Protocol driver consists of client and server portions. The client portion contains pure Java functions, and the server portion contains Java and native methods. The Java application sends JDBC calls to the Network Protocol driver client portion, which in turn, translates JDBC calls into database calls. The database calls are sent to the server portion of the Network Protocol driver that forwards the request to the database. When you use the Network Protocol driver, CLI native libraries are loaded on the server.

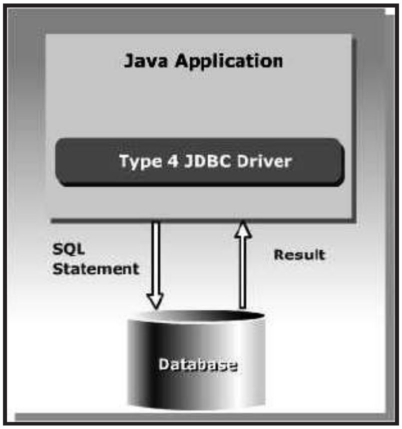
The following figure shows how the Network Protocol driver works.



The Network Protocol Driver

The Native Protocol Driver

The Native Protocol driver is called the Type 4 driver. It is a Java driver that interacts with the database directly using a vendor-specific network protocol. Unlike the other JDBC drivers, you do not require to install any vendor-specific libraries to use the Type 4 driver. Each database has the specific Type 4 drivers. The following figure shows how the Native Protocol driver works.



The Native Protocol Driver

Using JDBC API

You need to use database drivers and the JDBC API while developing a Java application to retrieve or store data in a database. The JDBC API classes and interfaces are available in the `java.sql` and `javax.sql` packages. The classes and interfaces perform a number of tasks, such as establish and close a connection with a database, send a request to a database, retrieve data from a database, and update data in a database. The classes and interfaces that are commonly used in the JDBC API are:

- The `DriverManager` class: Loads the driver for a database.
- The `Driver` interface: Represents a database driver. All JDBC driver classes must implement the `Driver` interface.
- The `Connection` interface: Enables you to establish a connection between a Java application and a database.
- The `Statement` interface: Enables you to execute SQL statements.
- The `ResultSet` interface: Represents the information retrieved from a database.
- The `SQLException` class: Provides information about exceptions that occur while interacting with databases.

To query a database and display the result using Java applications, you need to:

- Load a driver.
- Connect to a database.
- Create and execute JDBC statements.
- Handle SQL exceptions.

Loading a Driver

The first step to develop a JDBC application is to load and register the required driver using the driver manager. You can load and register a driver by:

- Using the `forName()` method of the `java.lang.Class` class.
- Using the `registerDriver()` static method of the `DriverManager` class.

Using the forName() Method

The `forName()` method loads the JDBC driver and registers the driver. The following signature is used to load a JDBC driver to access a database:

```
Class.forName("package.sub-package.sub-package.DriverClassName");
```

You can load the JDBC-Type 4 driver for SQL Server using the following code snippet:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

Using the registerDriver() Method

You can create an instance of the JDBC driver and pass the reference to the registerDriver() method. The following syntax is used for creating an instance of the JDBC driver:

```
Driver d=new <driver name>;
```

You can use the following code snippet to create an instance of the JDBC driver:

```
Driver d = new com.microsoft.sqlserver.jdbc.SQLServerDriver();
```

Once you have created the Driver object, call the registerDriver() method to register the driver, as shown in the following code snippet:

```
DriverManager.registerDriver(d);
```

Connecting to a Database

You need to create an object of the Connection interface to establish a connection of the Java application with a database. You can create multiple Connection objects in a Java application to access and retrieve data from multiple databases. The DriverManager class provides the getConnection() method to create a Connection object. The getConnection() method is an overloaded method that has the following three forms:

- public static Connection getConnection(String url)
- public static Connection getConnection(String url, Properties info)
- public static Connection getConnection(String url, String user, String password)

The getConnection (String url)method accepts the JDBC URL of the database, which you need to access as a parameter. You can use the following code snippet to connect a database using the getConnection() method with a single parameter:

```
String url =
"jdbc:sqlserver://localhost;user=MyUserName;password=password@123";
Connection con = DriverManager.getConnection(url);
```

The following signature is used for a JDBC Uniform Resource Location (URL) that is passed as a parameter to the getConnection() method:

```
<protocol>:<subprotocol>:<subname>
```

A JDBC URL has the following three components:

- protocol: Indicates the name of the protocol that is used to access a database. In JDBC, the name of the access protocol is always jdbc.
- subprotocol: Indicates the vendor of Relational Database Management System (RDBMS). For example, if you use the JDBC-Type 4 driver of SQL Server to access its database, the name of subprotocol will be sqlserver.
- subname: Indicates *Data Source Information* that contains the database information, such as the location of the database server, name of a database, user name, and password, to access a database server.

Creating and Executing JDBC Statements

Once a connection is created, you need to write the JDBC statements that are to be executed. You need to create a `Statement` object to send requests to a database and retrieve results from the same. The `Connection` object provides the `createStatement()` method to create a `Statement` object. You can use the following code snippet to create a `Statement` object:

```
Connection
con=DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Li
brary;user=user1;password=password#1234");
Statement stmt = con.createStatement();
```

You can use static SQL statements to send requests to a database. The SQL statements that do not contain runtime parameters are called static SQL statements. You can send SQL statements to a database using the `Statement` object. The `Statement` interface contains the following methods to send the static SQL statements to a database:

- `ResultSet executeQuery(String str)`: Executes an SQL statement and returns a single object of the type, `ResultSet`. This object provides you the methods to access data from a result set. The `executeQuery()` method should be used when you need to retrieve data from a database table using the `SELECT` statement. The syntax to use the `executeQuery()` method is:

  ```
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery(<SQL statement>);
  ```

  In the preceding syntax, `stmt` is a reference to the object of the `Statement` interface. The `executeQuery()` method executes an SQL statement, and the result retrieved from a database is stored in `rs` that is the `ResultSet` object.

- `int executeUpdate(String str)`: Executes SQL statements and returns the number of rows that are affected after processing the SQL statement. When you need to modify data in a database table using the Data Manipulation Language (DML) statements, `INSERT`, `DELETE`, and `UPDATE`, you can use the `executeUpdate()` method. The syntax to use the `executeUpdate()` method is:

  ```
  Statement stmt = con.createStatement();
  int count = stmt.executeUpdate(<SQL statement>);
  ```

  In the preceding syntax, the `executeUpdate()` method executes an SQL statement and the number of rows affected in a database is stored in `count` that is the `int` type variable.

- `boolean execute(String str)`: Executes an SQL statement and returns a `boolean` value. You can use this method when you are dynamically executing an unknown SQL string. The `execute()` method returns `true` if the result of the SQL statement is an object of `ResultSet` else it returns `false`. The syntax to use the `execute()` method is:

  ```
  Statement stmt = con.createStatement();
  stmt.execute(<SQL statement>);
  ```

  You can use the DML statements, `INSERT`, `UPDATE`, and `DELETE`, in Java applications to modify the data stored in the database tables. You can also use the Data Definition Language (DDL) statements, `CREATE`, `ALTER`, and `DROP`, in Java applications to define or change the structure of database objects.

Querying a Table

Using the SELECT statement, you can retrieve data from a table. The SELECT statement is executed using the executeQuery() method and returns the output in the form of a ResultSet object. You can use the following code snippet to retrieve data from the Authors table:

```
String str = "SELECT * FROM Authors";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(str);
```

In the preceding code snippet, str contains the SELECT statement that retrieves data from the Authors table. The result is stored in the ResultSet object, rs.

When you need to retrieve selected rows from a table, the condition to retrieve the rows is specified in the WHERE clause of the SELECT statement. You can use the following code snippet to retrieve selected rows from the Authors table:

```
String str = "SELECT * FROM Authors WHERE city='London'";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(str);
```

In the preceding code snippet, the executeQuery() method retrieves the details from the Authors table for a particular city.

Inserting Rows in a Table

You can add rows to an existing table using the INSERT statement. The executeUpdate() method enables you to add rows in a table. You can use the following code snippet to insert a row in the Authors table:

```
String str = " INSERT INTO Authors (au_id, au_name, phone, address, city,
state, zip) VALUES ('A004', 'Ringer Albert', '8018260752', ' 67 Seventh Av.
', 'Salt Lake City', 'UT', '100000078')";
Statement stmt = con.createStatement();
int count = stmt.executeUpdate(str);
```

In the preceding code snippet, str contains the INSERT statement that you need to send to a database. The object of the Statement interface, stmt, executes the INSERT statement using the executeUpdate() method and returns the number of rows inserted in the table to the count variable.

Updating Rows in a Table

You can modify the existing information in a table using the UPDATE statement. You can use the following code snippet to modify a row in the Authors table:

```
String str = "UPDATE Authors SET address='10932 Second Av.' where au_id=
'A001'";
Statement stmt = con.createStatement();
int count = stmt.executeUpdate(str);
```

In the preceding code snippet, str contains the UPDATE statement that you need to send to a database. The Statement object executes this statement using the executeUpdate() method and returns the number of rows modified in the table to the count variable.

Deleting Rows from a Table

You can delete the existing information from a table using the DELETE statement. You can use the following code snippet to delete a row from the Authors table:

```
String str = "DELETE FROM Authors WHERE au_id='A005'";
Statement stmt = con.createStatement();
int count = stmt.executeUpdate(str);
```

In the preceding code snippet, str contains the DELETE statement that you need to send to a database. The Statement object executes this statement using the executeUpdate() method and returns the number of rows deleted from the table to the count variable.

Creating a Table

You can use the CREATE TABLE statement to create and define the structure of a table in a database. You can use the following code snippet in a Java application to create a table:

```
String str="CREATE TABLE Publishers"
+"(pub_id VARCHAR(5),"
+"pub_name VARCHAR(50),"
+"phone INTEGER,"
+"address VARCHAR(50), "
+"city VARCHAR(50), "
+"ZIP VARCHAR(20))";
Statement stmt=con.createStatement();
stmt.execute(str);
```

In the preceding code snippet, str contains the CREATE TABLE statement to create the Publishers table. The execute() method is used to process the CREATE TABLE statement.

Altering and Dropping a Table

You can use the ALTER statement to modify the definition of the database object. You use the ALTER TABLE statement to modify the structure of a table. For example, you can use this statement to add a new column in a table, change the data type and width of an existing column, and add a constraint to a column. You can use the following code snippet to use the ALTER statement in a Java application to add a column in the Books table:

```
String str="ALTER TABLE Books ADD price INTEGER";
Statement stmt=con.createStatement();
stmt.execute(str);
```

You can use the DROP statement to drop an object from a database. You use the DROP TABLE statement to drop a table from a database. You can use the following code snippet to drop the MyProduct table using a Java application:

```
String str="DROP TABLE MyProduct";
Statement stmt=con.createStatement();
stmt.execute(str);
```

While creating JDBC applications, you need to handle exceptions. The execute() method can throw SQLException while executing SQL statements.

Handling SQL Exceptions

The java.sql package provides the SQLException class, which is derived from the java.lang.Exception class. SQLException is thrown by various methods in the JDBC API and enables you to determine the reason for the errors that occur while connecting a Java application to a database. You can catch SQLException in a Java application using the try and catch exception handling block. The SQLException class provides the following error information:

- **Error message**: Is a string that describes error.
- **Error code**: Is an integer value that is associated with an error. The error code is vendor specific and depends upon the database in use.
- **SQL state**: Is an *XOPEN* error code that identifies the error. Various vendors provide different error messages to define the same error. As a result, an error may have different error messages. The XOPEN error code is a standard message associated with an error that can identify the error across multiple databases.

The SQLException class contains various methods that provide error information. Some of the methods in the SQLException class are:

- int getErrorCode(): Returns the error code associated with the error occurred.
- String getSQLState(): Returns SQL state for the SQLException object.
- SQLException getNextException(): Returns the next exception in the chain of exceptions.

You can use the following code snippet to catch SQLException:

```
try
{
      String str = "DELETE FROM Authors WHERE au_id='A002'";
      Statement stmt = con.createStatement();
      int count = stmt.executeUpdate(str);
}
catch(SQLException sqlExceptionObject)
{
      System.out.println("Display Error Code");
           System.out.println("SQL Exception "+
           sqlExceptionObject.getErrorCode());
}
```

In the preceding code snippet, if the DELETE statement throws SQLException, then it is handled by the catch block. sqlExceptionObject is an object of the SQLException class and is used to invoke the getErrorCode() method.

Just a minute:

Identify the interface of the `java.sql` *package that must be implemented by all the JDBC driver classes.*

Answer:

`Driver`

Activity 9.1: Creating a JDBC Application to Query a Database

Accessing Result Sets

When you execute a query to retrieve data from a table using a Java application, the output of the query is stored in a `ResultSet` object in a tabular format. A `ResultSet` object maintains a *cursor* that enables you to move through the rows stored in the `ResultSet` object. By default, the `ResultSet` object maintains a cursor that moves in the forward direction only. As a result, it moves from the first row to the last row in `ResultSet`. In addition, the default `ResultSet` object is not updatable, which means that the rows cannot be updated in the default object. The cursor in the `ResultSet` object initially points before the first row.

Types of Result Sets

You can create various types of `ResultSet` objects to store the output returned by a database after executing SQL statements. The various types of `ResultSet` objects are:

- **Read only**: Allows you to only read the rows in a `ResultSet` object.
- **Forward only**: Allows you to move the result set cursor from the first row to the last row in the forward direction only.
- **Scrollable**: Allows you to move the result set cursor forward or backward through the result set.
- **Updatable**: Allows you to update the result set rows retrieved from a database table.

You can specify the type of a `ResultSet` object using the `createStatement()` method of the `Connection` interface. The `createStatement()` method accepts the `ResultSet` fields as parameters to create different types of the `ResultSet` object. The following table lists various fields of the `ResultSet` interface.

ResultSet Field	***Description***
`TYPE_SCROLL_SENSITIVE`	*Specifies that the cursor of the `ResultSet` object is scrollable and reflects the changes in the data made by other users.*
`TYPE_SCROLL_INSENSITIVE`	*Specifies that the cursor of the `ResultSet` object is scrollable and does not reflect changes in the data made by other users.*
`TYPE_FORWARD_ONLY`	*Specifies that the cursor of the `ResultSet` object moves in forward direction only from the first row to the last row.*
`CONCUR_READ_ONLY`	*Specifies the concurrency mode that does not allow you to update the `ResultSet` object.*
`CONCUR_UPDATABLE`	*Specifies the concurrency mode that allows you to update the `ResultSet` object.*

ResultSet Field	***Description***
CLOSE_CURSOR_AT_COMMIT	*Specifies the holdability mode that closes the open* `ResultSet` *object when the current transaction is committed.*
HOLD_CURSOR_OVER_COMMIT	*Specifies the holdability mode that keeps the* `ResultSet` *object open when the current transaction is committed.*

The Fields of the ResultSet Interface

The `createStatement()` method has the following overloaded forms:

- `Statement createStatement()`: Does not accept any parameter. This method creates a `Statement` object for sending SQL statements to the database. It creates the default `ResultSet` object that only allows forward scrolling.
- `Statement createStatement(int resultSetType, int resultSetConcurrency)`: Accepts two parameters. The first parameter indicates the `ResultSet` type that determines whether or not a result set cursor is scrollable or forward only. The second parameter indicates the concurrency mode for the result set that determines whether the data in result set is updateable or read only. This method creates a `ResultSet` object with the given type and concurrency.
- `Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)`: Accepts three parameters. Apart from the `ResultSet` types and the concurrency mode, this method also accepts a third parameter. This parameter indicates the holdability mode for the open result set object. This method creates a `ResultSet` object with the given type, concurrency, and the holdability mode.

Methods of the ResultSet Interface

The `ResultSet` interface contains various methods that enable you to move the cursor through the result set. The following table lists some methods of the `ResultSet` interface that are commonly used.

Method	***Description***
boolean first()	*Shifts the control of a result set cursor to the first row of the result set.*
boolean isFirst()	*Determines whether the result set cursor points to the first row of the result set.*
void beforeFirst()	*Shifts the control of a result set cursor before the first row of the result set.*
boolean isBeforeFirst()	*Determines whether the result set cursor points before the first row of the result set.*

Method	***Description***
`boolean last()`	*Shifts the control of a result set cursor to the last row of the result set.*
`boolean isLast()`	*Determines whether the result set cursor points to the last row of the result set.*
`void afterLast()`	*Shifts the control of a result set cursor after the last row of the result set.*
`boolean isAfterLast()`	*Determines whether the result set cursor points after the last row of the result set.*
`boolean previous()`	*Shifts the control of a result set cursor to the previous row of the result set.*
`boolean absolute(int i)`	*Shifts the control of a result set cursor to the row number that you specify as a parameter.*
`boolean relative(int i)`	*Shifts the control of a result set cursor, forward or backward, relative to the row number that you specify as a parameter. This method accepts either a positive value or a negative value as a parameter.*

The Methods of the ResultSet Interface

You can create a scrollable result set that scrolls backward or forward through the rows in the result set. You can use the following code snippet to create a read-only scrollable result set:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs=stmt.executeQuery ("SELECT * FROM Authors");
```

You can determine the location of the result set cursor using the methods in the `ResultSet` interface. You can use the following code snippet to determine if the result set cursor is before the first row in the result set:

```
if(rs.isBeforeFirst()==true)
System.out.println("Result set cursor is before the first row in the result
set");
```

In the preceding code snippet, `rs` is the `ResultSet` object that calls the `isBeforeFirst()` method.

You can move to a particular row, such as first or last, in the result set using the methods in the `ResultSet` interface. You can use the following code snippet to move the result set cursor to the first row in the result set:

```
if(rs.first()==true)
System.out.println(rs.getString(1) + ", " + rs.getString(2)+ ", " +
rs.getString(3));
```

In the preceding code snippet, `rs` is the `ResultSet` object that calls the `first()` method.

Similarly, you can move the result set cursor to the last row in the result set using the `last()` method.

To move to any particular row of a result set, you can use the `absolute()` method. For example, if the result set cursor is at the first row and you want to scroll to the fourth row, you should enter 4 as a parameter when you call the `absolute()` method, as shown in the following code snippet:

```
System.out.println("Using absolute() method");
rs.absolute(4);
int rowcount = rs.getRow();
System.out.println("row number is " + rowcount);
```

Note

You can pass a negative value to the `absolute()` method to set the cursor to a row with regard to the last row of the result set. For example, to set the cursor to the row preceding the last row, specify `rs.absolute(-2)`.

JDBC allows you to create an updateable result set that enables you to modify the rows in the result set. The following table lists some of the methods used with the updatable result set.

Method	***Description***
`void updateRow()`	*Updates the current row of the `ResultSet` object and updates the same in the underlying database table.*
`void insertRow()`	*Inserts a row in the current `ResultSet` object and the underlying database table.*
`void deleteRow()`	*Deletes the current row from the `ResultSet` object and the underlying database table.*
`void updateString(int columnIndex, String x)`	*Updates the specified column with the given string value. It accepts the column index whose value needs to be changed.*
`void updateString(String columnLabel, String x)`	*Updates the specified column with the given string value. It accepts the column name whose value needs to be changed.*
`void updateInt(int columnIndex, int x)`	*Updates the specified column with the given `int` value. It accepts the column index whose value needs to be changed.*

Method	***Description***
`void updateInt(String columnLabel, int x)`	*Updates the specified column with the given `int` value. It accepts the column name whose value needs to be changed.*

The Methods Used With the Updatable ResultSet

You can use the following code snippet to modify the information of the author using the updatable result set:

```
Statement stmt = con.createStatement();
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT au_id, city, state FROM Authors
WHERE au_id='A004’");
rs.next();
rs.updateString("state", "NY");
rs.updateString("city", "Columbia");
rs.updateRow();
```

In the preceding code snippet, the row is retrieved from the `Authors` table where the author id is `A004`. In the retrieved row, the value in the `state` column is changed to `NY` and the value in the `city` column is changed to `Columbia`.

Just a minute:

Identify the field of the `ResultSet` interface that allows the cursor to be scrollable and reflects the changes in the data.

1. *`TYPE_SCROLL_SENSITIVE`*
2. *`TYPE_SCROLL_INSENSITIVE`*
3. *`CONCUR_READ_ONLY`*
4. *`CONCUR_UPDATAB`*

Answer:

1. *`TYPE_SCROLL_SENSITIVE`*

In addition to `ResultSet`, Java provides `RowSet`. The `RowSet` interface provides the connected and disconnected environments. In a connected environment the `RowSet` object uses a JDBC driver to make a connection to the underlying database and maintains the connection throughout its life time. On the other hand, in a disconnected environment, the `RowSet` object makes a connection to the underlying database only to read data from the `ResultSet` object or write back to the database. After reading or writing data, the `RowSet` object disconnects the connection.

To establish the connected or disconnected environment, the JDBC API provides the following interfaces:

- `JdbcRowSet`: Provides the connected environment that is most similar to the `ResultSet` object. It makes the `ResultSet` object scrollable and updateable.
- `CachedRowSet`: Provides the disconnected environment. It has all the capabilities of the `JdbcRowSet` object. In addition, it has the following additional features:
  - It obtains a connection to a data source and executes the query.
  - It reads the data from the `ResultSet` object and fills the data in the `CachedRowSet` object.
  - It changes the data while it is disconnected.
  - It reconnects to the data source to write the changes back to it.
  - It checks and resolves the conflicts with the data source.
- `WebRowSet`: Provides the disconnected environment. It has all the capabilities of the `CachedRowSet` object. In addition, it can convert its objects into an Extensible Markup Language (XML) document. Further, it can also use the XML document to populate its object.
- `JoinRowSet`: Provides the disconnected environment. It has all the capabilities of the `WebRowSet` object. In addition, it allows you to create `SQL JOIN` between the `RowSet` objects.
- `FilteredRowSet`: Provides the disconnected environment. It has all the capabilities of the `JoinRowSet` object. In addition, it applies filter to make the selected data available.

The preceding interfaces can be used for specific need, such as connected or disconnected. However, there is another way to provide the `RowSet` implementation. For this, the `RowSetProvider` class provides APIs to get a `RowSetFactory` implementation that can be used to instantiate a proper `RowSet` implementation. It provides the following two methods:

- `static RowSetFactory newFactory()`: Is used to create an instance of the `RowSetFactory` implementation.
- `static RowSetFactory newFactory(String factoryClassName, ClassLoader c1)`: Is used to create an instance of the `RowSetFactory` from the specified factory class name.

The `RowSetFactory` interface defines the implementation of a factory that is used to obtain different types of `RowSet` implementations. It provides the following five methods:

- `CachedRowSet createCachedRowSet()`: Is used to create an instance of `CachedRowSet`.
- `FilteredRowSet createFilteredRowSet()`: Is used to create an instance of `FilteredRowSet`.
- `JdbcRowSet createJdbcRowSet()`: Is used to create an instance of `JdbcRowSet`.
- `JoinRowSet createJoinRowSet()`: Is used to create an instance of `JoinRowSet`.
- `WebRowSet createWebRowSet()`: Is used to create an instance of `WebRowSet`.

Practice Questions

1. Which one of the following methods is used to move the cursor to a particular row in a result set?
   a. `absolute()`
   b. `first()`
   c. `isFirst()`
   d. `relative()`

2. Which one of the following components of the JDBC URL indicates the vendor of RDBMS?
   a. Subname
   b. Supername
   c. Subprotocol name
   d. Superprotocol name

3. Identify the field of the `ResultSet` interface that specifies the cursor to be scrollable and does not reflect the changes in the data.
   a. `TYPE_SCROLL_INSENSITIVE`
   b. `TYPE_FORWARD_ONLY`
   c. `TYPE_SCROLL_SENSITIVE`
   d. `TYPE_READ_ONLY`

4. The `Connection` object provides the _______________ method to create a `Statement` object.

5. Which one of the following Java drivers interacts with the database directly using a vendor-specific network protocol?
   a. JDBC-ODBC Bridge driver
   b. Native-API driver
   c. Network Protocol driver
   d. Native Protocol driver

Summary

In this chapter, you learned that:

- The JDBC architecture has the following two layers:
  - JDBC application layer
  - JDBC driver layer
- JDBC supports the following four types of drivers:
  - JDBC-ODBC Bridge driver
  - Native-API driver
  - Network Protocol driver
  - Native Protocol driver
- The classes and interfaces of the JDBC API are defined in the `java.sql` and `javax.sql` packages.
- You can load a driver and register it with the driver manager by using the `forName()` method or the `registerDriver()` method.
- A `Connection` object establishes a connection between a Java application and a database.
- A `Statement` object sends a request and retrieves results to/ from a database.
- You can insert, update, and delete data from a table using the DML statements in Java applications.
- You can create, alter, and drop tables from a database using the DDL statements in Java applications.
- Once the SQL statements are executed, a `ResultSet` object stores the result retrieved from a database.
- You can create various types of the `ResultSet` objects, such as read only, updatable, and forward only.

Creating Applications Using Advanced Features of JDBC

Chapter 10

You have learned how to insert, delete, and update database tables using the `java.sql.Statement` object. However, by using this object, you can use only static values for insertion, updation, or deletion. Therefore, to work with dynamic values, Java provides the `java.sql.PreparedStatement` object.

At times, while executing the transactions in database, all statements within the transaction should either execute successfully or fail. To implement this functionality, Java provides database transaction management. In addition, to improve the efficiency of the application, Java provides batch updates. Further, to work with SQL procedures and functions, Java provides `java.sql.CallableStatement`. Thereafter, to work with metadata of database and tables, Java provides `java.sql.DatabaseMetaData` and `java.sql.ResultSetMetaData`.

This chapter focuses on creating application using `java.sql.PreparedStatement`, managing database transaction, performing batch updates, creating applications using `java.sql.CallableStatement`, and creating application to use the metadata of database and tables.

Objectives

In this chapter, you will learn to:

- Create applications using the PreparedStatement object
- Manage database transactions
- Implement batch updates in JDBC
- Create and call stored procedures in JDBC
- Use metadata in JDBC

Creating Applications Using the PreparedStatement Object

Consider a scenario where New Publishers, a publishing house, maintains details about books and authors in a database. The management of New Publishers wants an application that can help them access the details about authors based on different criteria. For example, the application should be able to retrieve the details of all the authors living in a particular city specified at runtime. In this scenario, you cannot use the `Statement` object to retrieve the details because the value for the city needs to be specified at runtime. You need to use the `PreparedStatement` object as it can accept runtime parameters.

The `PreparedStatement` interface is derived from the `Statement` interface and is available in the `java.sql` package. The `PreparedStatement` object allows you to pass runtime parameters to the SQL statements to query and modify the data in a table.

The `PreparedStatement` objects are compiled and prepared only once by JDBC. The further invocation of the `PreparedStatement` object does not recompile the SQL statements. This helps in reducing the load on the database server and improves the performance of the application.

Methods of the PreparedStatement Interface

The `PreparedStatement` interface inherits the following methods to execute the SQL statements from the `Statement` interface:

- `ResultSet executeQuery()`: Is used to execute the SQL statement and returns the result in a `ResultSet` object.
- `int executeUpdate()`: Executes an SQL statement, such as `INSERT`, `UPDATE`, or `DELETE`, and returns the count of the rows affected.
- `boolean execute()`: Executes an SQL statement and returns the `boolean` value.

Consider an example where you have to retrieve the details of an author by passing the author id at runtime. For this, the following SQL statement with a parameterized query can be used:

```
SELECT * FROM Authors WHERE au_id = ?
```

To submit such a parameterized query to a database from an application, you need to create a `PreparedStatement` object using the `prepareStatement()` method of the `Connection` object. You can use the following code snippet to prepare an SQL statement that accepts values at runtime:

```
stat=con.prepareStatement("SELECT * FROM Authors WHERE au_id = ?");
```

The `prepareStatement()` method of the `Connection` object takes an SQL statement as a parameter. The SQL statement can contain the symbol, `?`, as a placeholder that can be replaced by input parameters at runtime.

Before the SQL statement specified in the `PreparedStatement` object is executed, you must set the value of each `?` parameter. The value can be set by calling an appropriate `setXXX()` method, where `XXX` is the data type of the parameter. For example, consider the following code snippet:

```
stat.setString(1,"A001");
```

```
ResultSet result=stat.executeQuery();
```

In the preceding code snippet, the first parameter of the `setString()` method specifies the index value of the `?` placeholder, and the second parameter is used to set the value of the `?` placeholder. You can use the following code snippet when the value for the `?` placeholder is obtained from the user interface:

```
stat.setString(1,aid.getText());
ResultSet result=stat.executeQuery();
```

In the preceding code snippet, the `setString()` method is used to set the value of the `?` placeholder with the value retrieved at runtime from the `aid` textbox of the user interface.

The `PreparedStatement` interface provides various methods to set the value of placeholders for the specific data types. The following table lists some commonly used methods of the `PreparedStatement` interface.

Method	***Description***
`void setByte(int index, byte val)`	*Sets the Java `byte` type value for the parameter corresponding to the specified index.*
`void setBytes(int index, byte[] val)`	*Sets the Java `byte` type array for the parameter corresponding to the specified index.*
`void setBoolean(int index, boolean val)`	*Sets the Java `boolean` type value for the parameter corresponding to the specified index.*
`void setDouble(int index, double val)`	*Sets the Java `double` type value for the parameter corresponding to the specified index.*
`void setInt(int index, int val)`	*Sets the Java `int` type value for the parameter corresponding to the specified index.*
`void setLong(int index, long val)`	*Sets the Java `long` type value for the parameter corresponding to the specified index.*
`void setFloat(int index, float val)`	*Sets the Java `float` type value for the parameter corresponding to the specified index.*
`void setShort(int index, short val)`	*Sets the Java `short` type value for the parameter corresponding to the specified index.*
`void setString(int index, String val)`	*Sets the Java `String` type value for the parameter corresponding to the specified index.*

The Methods of the PreparedStatement Interface

Retrieving Rows

You can use the following code snippet to retrieve the details of the books written by an author from the Books table by using the PreparedStatement object:

```
String str = "SELECT * FROM Books WHERE au_id = ?";
PreparedStatement ps= con.prepareStatement(str);
ps.setString(1, "A001");
ResultSet rs=ps.executeQuery();
while(rs.next())
{
     System.out.println(rs.getString(1) + " " + rs.getString(2));
}
```

In the preceding code snippet, the str variable stores the SELECT statement that contains one input parameter. The setString() method is used to set the value for the au_id attribute of the Books table. The SELECT statement is executed using the executeQuery() method, which returns a ResultSet object.

Inserting Rows

You can use the following code snippet to create a PreparedStatement object that inserts a row into the Authors table by passing the author's data at runtime:

```
String str = "INSERT INTO Authors (au_id, au_name) VALUES (?,?)";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "1001");
ps.setString(2, "Abraham White");
int rt=ps.executeUpdate();
```

In the preceding code snippet, the str variable stores the INSERT statement that contains two input parameters. The setString() method is used to set the values for the au_id and au_name columns of the Authors table. The INSERT statement is executed using the executeUpdate() method, which returns an integer value that specifies the number of rows inserted into the table.

Updating and Deleting Rows

You can use the following code snippet to modify the state to CA where city is Oakland in the Authors table by using the PreparedStatement object:

```
String str = "UPDATE Authors SET state= ? WHERE city= ? ";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "CA");
ps.setString(2, "Oakland");
int rt=ps.executeUpdate();
```

In the preceding code snippet, two input parameters, state and city, contain the values for the state and city attributes of the Authors table, respectively.

You can use the following code snippet to delete a row from the `Authors` table, where `au_name` is `Abraham White` by using the `PreparedStatement` object:

```
String str = "DELETE FROM Authors WHERE au_name= ? ";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "Abraham White");
int rt=ps.executeUpdate();
```

Just a minute:

Identity the three methods of the `PreparedStatement` interface.

Answer:

The three methods of the `PreparedStatement` interface are:

1. *`ResultSet executeQuery()`*
2. *`int executeUpdate()`*
3. *`boolean execute()`*

Activity 10.1: Creating an Application that Uses the PreparedStatement Object

Managing Database Transactions

A transaction is a set of one or more SQL statements executed as a single unit. A transaction is complete only when all the SQL statements in a transaction execute successfully. If any one of the SQL statements in the transaction fails, the entire transaction is rolled back, thereby, maintaining the consistency of the data in the database.

JDBC API provides the support for transaction management. For example, a JDBC application is used to transfer money from one bank account to another. This transaction gets completed when the money is deducted from the first account and added to the second. If an error occurs while processing the SQL statements, both the accounts remain unchanged. The set of the SQL statements, which transfers money from one account to another, represents a transaction in the JDBC application.

The database transactions can be committed in the following two ways in the JDBC applications:

- **Implicit**: The `Connection` object uses the *auto-commit* mode to execute the SQL statements implicitly. The auto-commit mode specifies that each SQL statement in a transaction is committed automatically as soon as the execution of the SQL statement completes. By default, all the transaction statements in a JDBC application are auto-committed.
- **Explicit**: For explicitly committing a transaction statement in a JDBC application, you need to use the `setAutoCommit()` method. This method accepts either of the two values, `true` or `false`, to set or reset the auto-commit mode for a database. The auto-commit mode is set to `false` to commit a transaction explicitly. You can set the auto-commit mode to `false` using the following code snippet:

  ```
  con.setAutoCommit(false);
  ```

  In the preceding code snippet, `con` represents a `Connection` object.

Committing a Transaction

When you set the auto-commit mode to `false`, the operations performed by the SQL statements are not reflected permanently in a database. You need to explicitly call the `commit()` method of the `Connection` interface to reflect the changes made by the transactions in a database. All the SQL statements that appear between the `setAutoCommit(false)` method and the `commit()` method are treated as a single transaction and executed as a single unit.

The `rollback()` method is used to undo the changes made in the database after the last commit operation. You need to explicitly invoke the `rollback()` method to revert a database in the last committed state. When the `rollback()` method is invoked, all the pending transactions of a database are cancelled and the database gets reverted to the state in which it was committed previously. You can call the `rollback()` method using the following code snippet:

```
con.rollback();
```

In the preceding code snippet, `con` represents a `Connection` object.

You can use the following code to create a transaction that includes two INSERT statements and the transaction is committed explicitly using the commit() method:

```
import java.sql.*;
public class CreateTrans
{
    public static void main(String arg[])
    {
        try
        {
            /*Initialize and load the Type 4 JDBC driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

            /*Establish a connection with a data source*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");)
            {
                /*Set the auto commit mode to false*/
                con.setAutoCommit(false);

                /*Create a transaction*/
                try (PreparedStatement ps = con.prepareStatement("INSERT INTO
Publishers (pub_id, pub_name) VALUES (?, ?)");)
                {
                    /*Specify the value for the placeholders in the
PreparedStatement object*/
                    ps.setString(1, "P006");
                    ps.setString(2, "Darwin Press");
                    int firstctr = ps.executeUpdate();
                    System.out.println("First Row Inserted but not
committed");

                    /*Insert a row in the database table using the
prepareStatement() method*/
                    try (PreparedStatement ps2 = con.prepareStatement("INSERT
INTO Publishers(pub_id, pub_name) VALUES (?, ?)");)
                    {
                        ps2.setString(1, "P007");
                        ps2.setString(2, "MainStream Publishing");
                        int secondctr = ps2.executeUpdate();
                        System.out.println("Second Row Inserted but not
committed");

                        /*Commit a transaction*/
                        con.commit();
                        System.out.println("Transaction Committed, Please
check table for data");

                    }
                }
            }
        }
        catch (Exception e)
        {
            System.out.println("Error : " + e);
        }
    }
}
```

In the preceding code, the auto-commit mode is set to `false` using the `setAutoCommit()` method. All the statements that are executed after setting the auto-commit mode to `false` are treated as a transaction by the database.

Just a minute:

How can you commit a transaction explicitly?

Answer:

You can commit a transaction explicitly by setting the auto-commit mode to `false` *and using the* `commit()` *method.*

Implementing Batch Updates in JDBC

A *batch* is a group of update statements sent to a database to be executed as a single unit. You send the batch to a database in a single request using the same `Connection` object. This reduces network calls between the application and the database. Therefore, processing multiple SQL statements in a batch is a more efficient way as compared to processing a single SQL statement.

Note

A JDBC transaction can consist of multiple batches.

The `Statement` interface provides following methods to create and execute a batch of SQL statements:

- `void addBatch(String sql)`: Adds an SQL statement to a batch.
- `int[] executeBatch()`: Sends a batch to a database for processing and returns the total number of rows updated.
- `void clearBatch()`: Removes the SQL statements from the batch.

You can create a `Statement` object to perform batch updates. When the `Statement` object is created, an empty array gets associated with the object. You can add multiple SQL statements to the empty array for executing them as a batch. You also need to disable the auto-commit mode using the `setAutoCommit(false)` method while working with batch updates in JDBC. This enables you to roll back the entire transaction performed using a batch of updates if any SQL statement in the batch fails. You can use the following code snippet to create a batch of SQL statements:

```
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.addBatch("INSERT INTO Publishers (pub_id, pub_name) VALUES (P001, 'Sage
Publications')");
stmt.addBatch("INSERT INTO Product (pub_id, pub_name) VALUES (P002, 'Prisidio
Press')");
```

In the preceding code snippet, `con` is a `Connection` object. The `setAutoCommit()` method is used to set the auto-commit mode to `false`. The batch contains two `INSERT` statements that are added to the batch using the `addBatch()` method.

The SQL statements in a batch are processed in the order in which the statements appear in a batch. You can use the following code snippet to execute a batch of SQL statements:

```
int[] updcount=stmt.executeBatch();
```

In the preceding code snippet, `updcount` is an integer array that stores the values of the update count returned by the `executeBatch()` method. The update count is the total number of rows affected when an SQL statement is processed. The `executeBatch()` method returns the updated count for each SQL statement in a batch, if it is successfully processed.

Exception Handling in Batch Updates

The batch update operations can throw two types of exceptions, `SQLException` and `BatchUpdateException`. The JDBC API methods, `addBatch()` and `executeBatch()`, throw `SQLException` when any problem occurs while accessing a database. The `SQLException` exception is thrown when you try to execute a `SELECT` statement using the `executeBatch()` method. The `BatchUpdateException` class is derived from the `SQLException` class. The `BatchUpdateException` exception is thrown when the SQL statements in the batch cannot be executed due to:

- Presence of illegal arguments in the SQL statement.
- Absence of the database table.

`BatchUpdateException` uses an array of the update count to identify the SQL statement that throws the exception. The update count for all the SQL statements that are executed successfully is stored in the array in the same order in which the SQL statements appear in the batch. You can traverse the array of the update count to determine the SQL statement that is not executed successfully in a batch. The `null` value in an array of the update count indicates that the SQL statement in the batch failed to execute. You can use the following code to use the methods of the `SQLException` class that can be used to print the update counts for the SQL statements in a batch by using the `BatchUpdateException` object:

```
import java.sql.*;

public class BatchUpdate
{
    public static void main(String args[])
    {
        try
        {
            /*Batch Update Code comes here*/
        }
        catch (BatchUpdateException bexp)
        {
            /*Use the getMessage() method to retrieve the message associated 
with the exception thrown*/
            System.err.println("SQL Exception:" + bexp.getMessage());
            System.err.println("Update Counts:");
            /*Use the getUpdateCount() method to retrieve the update count 
for each SQL statement in a batch*/
            int[] updcount = bexp.getUpdateCounts();
            for (int i = 0; i <= updcount.length; i++)
            {
                /*Print the update count*/
                System.err.println(updcount[i]);
            }
        }
    }
}
```

Creating an Application to Insert Rows in a Table Using Batch Updates

You can execute multiple objects of the `Statement` and `PreparedStatement` interfaces together as batches in a JDBC application using batch updates. You can use the following code to insert data in a table using batch updates:

```
import java.sql.*;

public class BookInfo
{
    public static void main(String args[])
    {
        try
        {
            /*Initialize and load Type 4 JDBC driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

            /*Connect to a data source using Library DSN*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");
                    /*Create a Statement object*/
                    Statement stmt = con.createStatement();)
            {
                con.setAutoCommit(false);

                /*Add the INSERT statements to a batch*/
                stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES
('B004', 'Kane and Able')");
                stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES
('B005', 'The Ghost')");
                stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES
('B006', 'If Tommorrow Comes')");

                /*Execute a batch using executeBatch() method*/
                int[] results = stmt.executeBatch();
                System.out.println("");
                System.out.println("Using Statement object");
                System.out.println("--------------------------");
                for (int i = 0; i < results.length; i++)
                {
                    System.out.println("Rows affected by " + (i + 1) + "
INSERT statement: " + results[i]);
                }

                /*Use the PreparedStatement object to perform batch updates*/
                try (PreparedStatement ps = con.prepareStatement("INSERT INTO
Books (book_id, price) VALUES ( ?, ?)");)
                {
                    System.out.println("");
                    System.out.println("----------------------------------");
                    System.out.println("Using PreparedStatement object");
                    System.out.println("----------------------------------");

                    /*Specify the value for the placeholders*/
                    ps.setString(1, "B007");
                    ps.setInt(2, 575);
                    ps.addBatch();
                    ps.setString(1, "B008");
                    ps.setInt(2, 350);

                    /*Add the SQL statement to the batch*/
                    ps.addBatch();

                    /*Execute the batch of SQL statements*/
                    int[] numUpdates = ps.executeBatch();
                    for (int i = 0; i < numUpdates.length; i++)
                    {
                        System.err.println("Rows affected by " + (i + 1) + "
INSERT statement: " + numUpdates[i]);
                    }

                    /*Commit the INSERT statements in the batch*/
                    con.commit();


                }
            }
        }
        catch (BatchUpdateException bue)
        {
            System.out.println("Error : " + bue);
        }
        catch (SQLException sqle)
        {
            System.out.println("Error : " + sqle);
        }
        catch (Exception e) {
            System.out.println("Error : " + e);
        }
    }
}
```

In the preceding code, two batches are created using the Statement and PreparedStatement objects. The INSERT statements are added to the batch using the addBatch() method and are executed using the executeBatch() method. The executeBatch() method returns an array, which stores the update count for all the SQL statements in the batch. You can display the number of rows affected by each SQL statement in the batch using the for loop.

Creating and Calling Stored Procedures in JDBC

The java.sql package provides the CallableStatement interface that contains various methods to enable you to call database stored procedures. The CallableStatement interface is derived from the PreparedStatement interface.

Creating Stored Procedures

Stored procedures can be created using JDBC applications. You can use the executeUpdate() method to execute the CREATE PROCEDURE SQL statement. Stored procedures can be of two types, parameterized and non parameterized.

You can use the following code snippet to create a non parameterized stored procedure in a JDBC application:

```
String str = "CREATE PROCEDURE Authors_info "
+"AS "
+ "SELECT au_id,au_name "
+ "FROM Authors "
+ "WHERE city = 'Oakland' "
+ "ORDER BY au_name";
Statement stmt=con.createStatement();
int rt=stmt.executeUpdate(str);
```

In the preceding code snippet, the SELECT statement specifies that the data is retrieved from the Authors table in the order of the au_name column. The Connection object con is used to send the CREATE PROCEDURE SQL statement to a database. When you execute the code, the Authors_info stored procedure is created and stored in the database.

You can use the following code snippet to create a parameterized stored procedure:

```
str = " CREATE PROCEDURE Authors_info_prmtz @auth_id varchar(15) ,@auth_name
varchar(20) output, @auth_city varchar(20) output,@auth_state varchar(20)
output "
+ " AS "
+ " SELECT @auth_name=au_name, @auth_city=city, @auth_state=state "
+ " FROM Authors "
+ " WHERE au_id=@auth_id ";
Statement stmt=con.createStatement();
int rt=stmt.executeUpdate(str);
```

In the preceding code snippet, the Authors_info_prmtz stored procedure is created that accepts the author id as a parameter and retrieves the corresponding author information from the database. The retrieved information is stored in the OUT parameters. The output keyword is used to represent OUT parameters.

A stored procedure can accept one or multiple parameters. A parameter of a stored procedure can take any of the following forms:

- IN: Refers to the argument that you pass to a stored procedure.
- OUT: Refers to the return value of a stored procedure.

- INOUT: Combines the functionality of the IN and OUT parameters. The INOUT parameter enables you to pass an argument to a stored procedure. The same parameter can also be used to store a return value of a stored procedure.

> ***Note***
> *When you use the stored procedures to perform database operations, it reduces network traffic because instead of sending multiple SQL statements to a database, a single stored procedure is executed.*

Calling a Stored Procedure Without Parameters

The Connection interface provides the prepareCall() method that is used to create the CallableStatement object. This object is used to call a stored procedure of a database. The prepareCall() method is an overloaded method that has various forms. Some of the commonly used forms are:

- CallableStatement prepareCall(String str): Creates a CallableStatement object to call a stored procedure. The prepareCall() method accepts a string as a parameter that contains an SQL statement to call a stored procedure. You can also specify the placeholders to accept parameters in the SQL statement.
- CallableStatement prepareCall(String str, int resSetType, int resSetConcurrency): Creates a CallableStatement object that returns the ResultSet object, which has the specified result set type and concurrency mode. The method accepts the following three parameters:
  - String object: Contains an SQL statement to call a stored procedure. The SQL statement can contain one or more parameters.
  - ResultSet types: Specifies any of the three Resultset types, TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE.
  - ResultSet concurrency modes: Specifies either of these concurrency modes, CONCUR_READ_ONLY or CONCUR_UPDATABLE, for a result set.
- CallableStatement prepareCall(String str, int resSetType, int resSetConcurrency, int resSetHoldability): Creates a CallableStatement object that returns a ResultSet object that has the specified result type, concurrency mode, and constant to set the result set state.

The following signature is used to call a stored procedure without parameters:

```
exec <procedure_name>
```

You can use the following code snippet to call a stored procedure that does not accept parameters:

```
String str = "exec Authors_info";
            CallableStatement cstmt = con.prepareCall(str);
            ResultSet rs = cstmt.executeQuery();
            while (rs.next())
             {
              System.out.println(" Author Id : " + rs.getString(1) + "\t");
```

```
                System.out.println(" Author Name : " + rs.getString(2) + "\t");
            }
```

In the preceding code snippet, con is the Connection object that invokes the prepareCall() method. The str variable contains the call to the Authors_info stored procedure, and this call is passed as a parameter to the prepareCall() method.

Calling a Stored Procedure with Parameters

The SQL escape syntax is used to call a stored procedure with parameters. The SQL escape syntax is a standard way to call a stored procedure from RDBMS and is independent of RDBMS. The driver searches the SQL escape syntax in the code and converts the SQL escape syntax into the database compatible form. There are two forms of the SQL escape syntax, one that contains result parameter and the other that does not contain result parameters. Both the forms can take multiple parameters. If the SQL escape syntax contains a result parameter, the result parameter is used to return a value from a stored procedure. The result parameter is an OUT parameter. Other parameters of the SQL escape syntax can contain the IN, OUT, or INOUT parameter. The signature of the SQL escape syntax is:

```
{[? =] call <procedure_name> [<parameter1>,<parameter2>, ., <parameterN>]}
```

The placeholders are used to represent the IN, OUT, and INOUT parameters of a stored procedure in the procedure call. The signature to call a stored procedure with parameters is:

```
{ call <procedure_name>(?) };
```

You need to set the value of the IN parameters before the CallableStatement object is executed. Otherwise, SQLException is thrown while processing the stored procedure. The set methods are used to specify the values for the IN parameters. The CallableStatement interface inherits the set methods from the PreparedStatement interface. The signature to set the value of the IN parameter is:

```
<CallableStatement_object>.setInt(<value>);
```

In the preceding signature, the setInt() method is used to set the value for an integer type, IN parameter.

If the stored procedure contains the OUT and INOUT parameters, these parameters should be registered with the corresponding JDBC types before a call to a stored procedure is processed. The JDBC types determine the Java data types that are used in the get methods while retrieving the values of the OUT and INOUT parameters. The registerOut() method is used to register the parameters. SQLException is thrown if the placeholders, representing the OUT and INOUT parameters, are not registered. The prototypes of the registerOut() method are:

- registerOut(int index, int stype): Accepts the position of the placeholder and a constant in the java.sql.Types class as parameters. The java.sql.Types class contains constants for various JDBC types. For example, if you want to register the VARCHAR SQL data type, you should use the STRING constant of the java.sql.Types class. You can use the following method call to the registerOut() method to register a parameter:

  ```
  cstmt.registerOutParameter(1, java.sql.Types.STRING);
  ```

- `registerOut(int index, int stype, int scale)`: Accepts the position of a placeholder, a constant in the `java.sql.Types` class, and a scale of the value that is returned as parameters. You need to define the scale of a parameter while registering numeric data types, such as `NUMBER`, `DOUBLE`, and `DECIMAL`. For example, if you want to register the `DECIMAL` SQL data type that has three digits after decimal, the value for the scale parameter should be three. You can use the following code snippet to specify the scale parameter while invoking the `registerOut()` method:

```
cstmt.registerOutParameter(1, java.sql.Types.DECIMAL, 3);
```

You can use the `prepareCall()` method to call a stored procedure that accepts parameters. The `prepareCall()` method returns a result after processing the SQL and control statements defined in the procedure body. You can use the following code to call a stored procedure with parameters:

```
import java.sql.*;

public class CallProc
{

    public static void main(String args[])
    {
        String id, name, address, city;
        try
        {
            String str = "{call Authors_info_prmtz(?, ?, ?, ?)}";

            /*Initialize and load Type 4 JDBC driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            /*Establish a connection with the database*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");
                    /*Call a stored procedure*/
                    CallableStatement cstmt = con.prepareCall(str);)
            {
                /*Pass IN parameter*/
                cstmt.setString(1, "A001");
                /*Register OUT parameters*/
                cstmt.registerOutParameter(2, Types.VARCHAR);
                cstmt.registerOutParameter(3, Types.VARCHAR);
                cstmt.registerOutParameter(4, Types.VARCHAR);
                /*Process the stored procedure*/
                cstmt.execute();
                /*Retrieve Authors information*/
                name = cstmt.getString(2);
                address = cstmt.getString(3);
                city = cstmt.getString(4);
                /*Display author information*/
                System.out.println("");
                System.out.println("Displaying Author Information");
                System.out.println("-------------------------------");
                System.out.println("First name: " + name);
                System.out.println("Address: " + address);
                System.out.println("City: " + city);
            }
        }
        catch (Exception e)
        {
            System.out.println("Error " + e);
        }
    }
}
```

In the preceding code, the `Authors_info_prmtz` stored procedure is invoked. This stored procedure accepts one `IN` type parameter and four `OUT` type parameters. The `IN` parameter is used to specify the id of an author whose information you want to retrieve. The `OUT` parameters are used to retrieve the first name, last name, address, and city of the authors. The `setString()` method is used to specify the author id, and the `registerOut()` method is used to register the `OUT` parameters.

Using Metadata in JDBC

Metadata is the information about data, such as structure and properties of table. For example, a database contains the `employee` table that has the `name`, `address`, `salary`, `designation`, and `department` columns. The metadata of the `employee` table includes certain information, such as names of the columns, data type of each column, and constraints to enter data values in the columns. JDBC API provides the following two metadata interfaces to retrieve the information about the database and the result set:

- `DatabaseMetaData`
- `ResultSetMetaData`

Using the DatabaseMetaData Interface

The `DatabaseMetaData` interface provides the methods that enable you to determine the properties of a database. These properties include names of database tables, database version, SQL keywords, and isolation levels of the data stored in the database.

You can create an object of `DatabaseMetaData` using the `getMetaData()` method of the `Connection` interface. You can use the following code snippet to create an object of the `DatabaseMetaData` interface:

```
DatabaseMetaData dm=con.getMetaData();
```

In the preceding code snippet, `con` refers to an object of the `Connection` interface.

The methods declared in the `DatabaseMetaData` interface retrieve the database-specific information. The following table lists some commonly used methods of the `DatabaseMetaData` interface.

Method	***Description***
`ResultSet getColumns(String catalog, String schema, String table_name, String column_name)`	*Retrieves the information about a column of a database table that is available in the specified database catalog.*
`Connection getConnection()`	*Retrieves the database connection that creates the `DatabaseMetaData` object.*
`String getDriverName()`	*Retrieves the name of the JDBC driver for the `DatabaseMetaData` object.*
`String getDriverVersion()`	*Retrieves the version of the JDBC driver.*
`ResultSet getPrimaryKeys(String catalog, String schema, String table)`	*Retrieves the information about the primary keys of the database tables.*

Method	***Description***
`String getURL()`	*Retrieves the URL of the database.*
`boolean isReadOnly()`	*Returns a `boolean` value that indicates whether the database is read only.*
`boolean supportsSavepoints()`	*Returns a `boolean` value that indicates whether the database supports savepoints.*

The Methods of the DatabaseMetaData Interface

The methods in the `DatabaseMetaData` interface retrieve information about the database to which a Java application is connected. You can use the following code to retrieve and display the names of various database tables by using the methods of the `DatabaseMetaData` interface:

```
import java.sql.*;

public class TableNames
{
    public static void main(String args[])
    {
        try {
            /*Initialize and load the Type 4 driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            /*Establish a connection with the database*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");)
            {
                /*Create a DatabaseMetaData object*/
                DatabaseMetaData dbmd = con.getMetaData();
                String[] tabTypes = {"TABLE"};
                /*Retrieve the names of database tables*/
                System.out.println("");
                System.out.println("Tables Names");
                System.out.println("-------------------");
                ResultSet tablesRS = dbmd.getTables(null, null, null,
tabTypes);
                while (tablesRS.next()) /*Display the names of database
tables*/
                {
                    System.out.println(tablesRS.getString("TABLE_NAME"));
                }
                            }
        }
        catch (Exception e)
        {
            System.out.println("Error : " + e);
        }
    }
}
```

In the preceding code, a connection is established with the Library data source. The object of the DatabaseMetaData interface is declared using the getMetaData() method. The DatabaseMetaData object is used to retrieve the names of database tables using the getTables() method.

Using the ResultSetMetaData Interface

The ResultSetMetaData interface contains various methods that enable you to retrieve information about the data in a result set, such as numbers, names, and data types of the columns. The ResultSet interface provides the getMetaData() method to create an object of the ResultSetMetaData interface. You can use the following code snippet to create an object of the ResultSetMetaData interface:

```
ResultSetMetaData rm=rs.getMetaData();
```

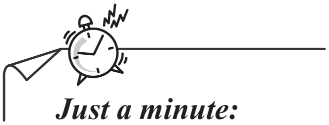
In the preceding code snippet, rs refers to an object of the ResultSet interface. rs calls the getMetaData() method to create an object of the ResultSetMetaData interface.

The following table lists some commonly used methods of the ResultSetMetaData interface.

Method	***Description***
int getColumnCount()	*Returns an integer indicating the total number of columns in a ResultSet object.*
String getColumnLabel(int column_index)	*Retrieves the title of the table column corresponding to the specified index.*
String getColumnName(int column_index)	*Retrieves the name of the table column corresponding to the specified index.*
int getColumnType(int column_index)	*Retrieves the SQL data type of the table column corresponding to the specified index.*
String getTableName(int column_index)	*Retrieves the name of the database table that contains the column corresponding to the specified index.*
boolean isAutoIncrement(int column_index)	*Returns a boolean value that indicates whether the table column corresponding to the specified index increments automatically.*
boolean isCaseSensitive(int column_index)	*Returns a boolean value that indicates whether the table column corresponding to the specified index is case-sensitive.*

Method	*Description*
boolean isReadOnly(int column_index)	*Returns a* `boolean` *value that indicates whether the column in a* `ResultSet` *column corresponding to the specified index is read only.*
boolean isWritable(int column_index)	*Returns a* `boolean` *value that indicates whether the* `ResultSet` *column corresponding to the specified index is writeable.*

The Methods of the ResultSetMetaData Interface



Just a minute:

What are the metadata interfaces used to retrieve information about the database and the result set?

Answer:

The metadata interfaces used to retrieve information about the database and the result set are:

1. `DatabaseMetaData`
2. `ResultSetMetaData`

Activity 10.2: Creating an Application to Determine the Structure of a Table

Practice Questions

1. The ________ interface provides the methods that enable you to determine the properties of a database or RDBMS.

2. Identify the method of the `PreparedStatement` interface that sets the Java `byte` type value for the parameter corresponding to the specified index.

   a. `setByte(int index, byte val)`
   b. `setBytes(int index, byte[] val)`
   c. `setString(int index, String val)`
   d. `setShort(int index, short val)`

3. Why do you need to disable the auto-commit mode while working with batch updates?

4. Which is a standard way to call a stored procedure from RDBMS?

5. Which method of the `Connection` interface is used to create the `callableStatement` object?

   a. `prepareCall()`
   b. `getMetaData()`
   c. `prepareStatement()`
   d. `createStatement()`

Summary

In this chapter, you learned that:

- The `PreparedStatement` object allows you to pass runtime parameters to the SQL statements using the placeholders.
- There can be multiple placeholders in a single SQL statement. An index value is associated with each placeholder depending upon the position of the placeholder in the SQL statement.
- The placeholder stores the value assigned to it until the value is explicitly changed.
- A transaction is a set of one or more SQL statements executed as a single unit. A transaction is complete only when all the SQL statements in a transaction are successfully executed.
- If the `setAutoCommit()` method is set to `true`, the database operations performed by the SQL statements are automatically committed in the database.
- The `commit()` method reflects the changes made by the SQL statements permanently in the database.
- The `rollback()` method is used to undo the effect of all the SQL operations performed after the last commit operation.
- A batch is a group of update statements sent to a database to be executed as a single unit. You send the batch to a database as a single request using the same `Connection` object.
- The `executeBatch()` method returns an integer array that stores the update count for all the SQL statements that are executed successfully in a batch. The update count is the number of database rows affected by the database operation performed by each SQL statement.
- Batch update operations can throw two types of exceptions, `SQLException` and `BatchUpdateException`.
- `SQLException` is thrown when the database access problem occurs. `SQLException` is also thrown when a `SELECT` statement that returns a `ResultSet` object is executed in a batch.
- `BatchUpdateException` is thrown when the SQL statement in the batch cannot be executed due to the problem in accessing the specified table or presence of illegal arguments in the SQL statement.
- The `CallableStatement` interface contains various methods that enable you to call the stored procedures from a database.
- The parameters of a stored procedure can take any of the following three forms:
  - `IN`
  - `OUT`
  - `INOUT`
- Metadata is the information about data, such as the structure and properties of table.
- JDBC API provides two metadata interfaces to retrieve the information about the database and result set, `DatabaseMetaData` and `ResultSetMetaData`.
- The `DatabaseMetaData` interface declares the methods that enable you to determine the properties of a database.
- The `ResultSetMetaData` interface declares the methods that enable you to determine the information of a result set.

- The getMetaData() method of the Connection interface enables you to obtain the objects of the DatabaseMetaData interface. The methods in the DatabaseMetaData interface retrieve the information only about the database to which a Java application is connected.
- The getMetaData() method of the ResultSet interface enables you to create the instance of the ResultSetMetaData interface.

Solutions to Practice Questions

Chapter 1

1. True
2. c. Method-local inner class
3. d. It does not have a name and is either a subclass of a class or an implementer of an interface.
4. True
5. True

Chapter 2

1. True
2. a. `String pattern()`
3. b. Reluctant
4. b. `[b-z&&[^bcd]]`
5. a. Greedy

Chapter 3

1. d. <>
2. True
3. c. `public class Myclass <T> { }`
4. a. `public <T> T myMethod()`
5. a. `?`

Chapter 4

1. d. `Vector`
2. c. `Hashtable`
3. a. `Iterator`
4. a. `put()`
5. a. `Set`

Chapter 5

1. c.1
2. a. `isAlive()`
3. b. `getName()`

4. b. `sleep()`
5. b. Both the statements are true.

Chapter 6

1. b. `synchronized`
2. a. `wait()`
3. d. `notify()`
4. a. `java.util.concurrent.atomic`
5. b. `tryLock()`

Chapter 7

1. a. `FileOutputStream(File file,boolean append)`
2. True
3. a. `available()`
4. a. `FileWriter`
5. a. `FileReader`

Chapter 8

1. a. `get()`
2. b. `getFileSystem()`
3. d. `isAbsolute()`
4. a.`FileAlreadyExistsException`
5. a. `FileVisitor`

Chapter 9

1. a. `absolute()`
2. c. Subprotocol name
3. a. `Type_SCROLL_INSENSITIVE`
4. `createStatement()`
5. d. Native Protocol driver

Chapter 10

1. `DatabaseMetaData`
2. a. `setByte(int index, byte val)`

3. Because it enables you to rollback the entire transaction performed using a batch update if any SQL statement in the batch fails.
4. The SQL escape syntax
5. a. `prepareCall()`

Reference Reading

Programming in Java

The following table lists the references for the Programming in Java course.

S. No.	*Chapter*	*Topic*	*Reference Reading: Books*	*Reference Reading: URLs*
1.	*Implementing Inner Classes and Type Casting*	*Creating Inner Classes*	*The Complete Reference – Java, Seventh Edition by Herbert Schildt*	*http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html*
		Implementing Type Casting	*Introduction to Programming Using Java by David J. Eck*	*http://way2java.com/casting-operations/data-type-casting-type-conversion/*
2.	*Working with Regular Expressions and Localization*	*Processing Strings Using Regex*	*Regular Expressions by Jan Goyvaerts*	*http://books.google.co.in/books?id=587w6j5vPIQC&pg=PA620&dq=pattern+class+in+regular+expressions&hl=en&sa=X&ei=bp6AUePiFc_NrQfG_YHoCw&ved=0CC8Q6AEwAA#v=onepage&q=pattern%20class%20in%20regular%20expressions&f=false* *http://docs.oracle.com/javase/tutorial/essential/regex/*
		Implementing Localization	*Pro Java Programming, Second Edition by BRETT SPELL*	*http://docs.oracle.com/javase/tutorial/i18n/*
3.	*Working with Generics*	*Creating User-defined Generic Classes and Methods*	*The Complete Reference – Java, Seventh Edition by Herbert Schildt*	*http://docs.oracle.com/javase/tutorial/java/generics/index.html*
		Implementing Type-safety	*Java Generics and Collections by Maurice Naftalin, Philip Wadler*	*http://www.tutorialspoint.com/java/java_generics.htm*

S. No.	***Chapter***	***Topic***	***Reference Reading: Books***	***Reference Reading: URLs***
4.	*Working with Collections*	*Using the Set Interface*	*The Complete Reference – Java, Seventh Edition by Herbert Schildt*	*http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html* *http://www.tutorialspoint.com/java/java_set_interface.htm*
		Using the List Interface		*http://docs.oracle.com/javase/tutorial/collections/interfaces/list.html*
		Using the Map Interface		*http://tutorials.jenkov.com/java-collections/list.html* *http://docs.oracle.com/javase/tutorial/collections/interfaces/map.html*
		Using the Deque Interface		*http://tutorials.jenkov.com/java-collections/deque.html* *http://docs.oracle.com/javase/6/docs/api/java/util/Deque.html*
		Implementing Sorting		*http://tutorials.jenkov.com/java-collections/sorting.html* *http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html*
5.	*Working with Threads*	*Using Threads in Java*	*Java Threads by Scott Oaks, Henry Wong*	*http://docs.oracle.com/javase/tutorial/essential/concurrency/* *http://www.roseindia.net/java/thread/*
		Creating Threads	*The Complete Reference – Java, Seventh Edition by Herbert Schildt*	*http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html*

S. No.	**Chapter**	**Topic**	**Reference Reading: Books**	**Reference Reading: URLs**
			Harnessing Java 7: A Comprehensive Approach to Learning Java 7 by Kishori Sharan	*http://books.google.co.in/books?id=13ECdYRKDXQC&pg=PA268&dq=Creating+threads+in+java&hl=en&sa=X&ei=J9nLUbGVC8yVrgeRooDICw&ved=0CDoQ6AEwAg#v=onepage&q=Creating%20threads%20in%20java&f=false*
6.	*Implementing Thread Synchronization and Concurrency*	*Implementing Thread Synchronization*	*The Complete Reference – Java, Seventh Edition by Herbert Schildt*	*http://www.tutorialspoint.com/java/java_thread_synchronization.htm* *http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html*
		Implementing Concurrency	*Java Concurrency In Practice by Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea*	*http://tutorials.jenkov.com/java-concurrency/index.html* *http://books.google.co.in/books?id=EK43StEVfJIC&printsec=frontcover&dq=concurrency+in+java+pdf&hl=en&sa=X&ei=IuHLUfVPhIesB8HXgZAI&ved=0CC0Q6AEwAA*
7.	*Working with Streams*	*Working with Input Stream*	*The Complete Reference – Java, Seventh Edition by Herbert Schildt* *Core Java: a Comprehensive Study by Mahesh P. Matha*	*http://docs.oracle.com/javase/6/docs/api/java/io/package-tree.html* *http://tutorials.jenkov.com/java-io/index.html*
		Working with Output Stream	*Sun Certified Programmer for Java 6 study guide by Kathy Sierra and Bert Bates*	*http://books.google.co.in/books?id=mvzgNSmHEUAC&pg=PT413&dq=io+package+in+java&hl=en&sa=X&ei=b-nLUZXRNsinrgfzh4DICg&ved=0CC0Q6AEwAA*

S. No.	Chapter	Topic	Reference Reading: Books	Reference Reading: URLs
8.	*Working with NIO Classes and Interfaces*	*Introducing NIO*	*Pro Java 7 NIO.2 by Anghel Leonard*	*http://docs.oracle.com/javase/6/docs/api/java/nio/package-summary.html*
		Performing the Read and Write Operations on a File		*http://javavsnet.wordpress.com/2011/08/05/reading-and-writing-in-nio-java-7-new-io-api/* *http://books.google.co.in/books?id=jViEhsJxdIMC&pg=PA828&dq=nio+read+write+operations&hl=en&sa=X&ei=PhPNUb7CMJG0rAfioYCYDg&ved=0CFIQ6AEwBg#v=onepage&q=nio%20read%20write%20operations&f=false*
9.	*Introduction to JDBC*	*Identifying the Layers in the JDBC Architecture*	*Advanced Java Programming with Database Application(http://www.msuniv.ac.in/AdvancedJavaProgrammingwithDatabaseApplication.pdf)*	*http://docs.oracle.com/javase/tutorial/jdbc/overview/architecture.html*
		Identifying the Types of JDBC Drivers		*http://www.jdbc-tutorial.com/jdbc-driver-types.htm*
		Using JDBC API		*http://docs.oracle.com/cd/E19644-01/817-5449/djjdbc.html*
		Accessing Result Sets		*http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html*
10.	*Creating Applications Using Advanced Features of JDBC*	*Creating Applications Using the PreparedStatement Object*	*JDBC Metadata, MySQL, and Oracle Recipes: A Problem-Solution Approach by Mahmoud Parsian*	*http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/getstart/preparedstatement.html*
		Managing Database Transactions		*http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html*

S. No.	***Chapter***	***Topic***	***Reference Reading: Books***	***Reference Reading: URLs***
		Implementing Batch Updates in JDBC		*http://www.tutorialspoint.com/jdbc/jdbc-batch-processing.htm*
		Creating and Calling Stored Procedures in JDBC		*http://www.tutorialspoint.com/jdbc/jdbc-stored-procedure.htm*
		Using Metadata in JDBC		*http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html* *http://www.roseindia.net/jdbc/Jdbc-meta-data-get-tables.shtml*

List of References

Disclaimer: All URLs listed in the Reference Reading section have been examined for accuracy and appropriateness at the time of addition. However, the same cannot be guaranteed over time as websites and their contents change constantly.

Glossary

A

Anonymous Inner Class

An anonymous inner class has no name, and it is either a subclass of a class or an implementer of an interface.

Atomic Operation

An atomic operation is an operation that is performed as a single unit. In addition, an atomic operation cannot be stopped in between by any other operations.

Atomic Variables

Atomic variables are the single variables on which operations, such as increment and decrement, are done in a single step.

Auto-commit

The auto-commit mode specifies that each SQL statement in a transaction is committed automatically as soon as the execution of the SQL statement completes.

B

Batch

A batch is a group of update statements sent to a database to be executed as a single unit.

Buffer

A buffer is a storage area in the memory that can be used for writing or reading data. As compared to streams, buffers can be used to provide faster I/O operations.

BufferedInputStream Class

The `BufferedInputStream` class is used to perform the read operations by using a temporary storage, buffer, in the memory.

BufferedOutputStream Class

The `BufferedOutputStream` class writes bytes to an output stream using a buffer for increased efficiency.

BufferedReader Class

The `BufferedReader` class is used to read the text from a character-input stream, such as a file, console, and array, while buffering characters.

BufferedWriter Class

The `BufferedWriter` class can be used to write text to an output stream.

C

Call Level Interface (CLI)

CLI is an interface consists of functions written in the C language to access databases.

Concurrency

Concurrency enables Java programmers to improve the efficiency and resource utilization of their applications by creating concurrent programs.

Cursor

Cursor enables you to move through the rows stored in the `ResultSet` object.

D

Data Source Information

Data Source Information contains the database information, such as the location of the database server, name of a database, user name, and password, to access a database server.

Deadlock

The deadlock condition arises in a computer system when two threads wait for each other to complete the operations before performing the individual action.

Doubly-linked List

The doubly-linked list is a set of sequentially linked records called nodes where each node contains two links that are the references of the previous node and the next node in the sequence, respectively.

Downcasting

Downcasting is usually done along the class hierarchy in a direction from the base class towards the derived classes.

E

Explicit Casting

Explicit casting occurs when one data type cannot be implicitly converted into another data type.

F

FileInputStream Class

The `FileInputStream` class is used to read data and the steams of bytes from the file.

FileOutputStream Class

The `FileOutputStream` class is used for writing data, byte by byte, to a file.

FileReader Class

The `FileReader` class is used for reading characters from a file, but it does not define any method of its own.

FileWriter Class

The `FileWriter` class writes character data to a file.

G

Generics

Generics mean parameterized types that enable you to write the code that can work with many types of object.

I

Implicit Casting

Implicit casting refers to an automatic conversion of one data type into another.

Inner Classes

Inner classes are the classes defined inside another class.

L

Lock Starvation

Lock starvation occurs when the execution of a thread is postponed because of its low priority.

M

Metadata

Metadata is the information about data, such as structure and properties of table.

Method-local Inner

A method-local inner is a class defined inside the method of the enclosing class.

O

Open Database Connectivity (ODBC)

ODBC is an open standard API to communicate with databases.

R

Regular Expressions

Regular expressions allow you to match a character sequence with another string.

Regular Inner Class

A regular inner class is a class whose definition appears inside the definition of another class.

S

Semaphore

A semaphore is a lock with an internal counter. It is used to restrict access to shared resources by multiple threads at the same time.

Static Inner Classes

Static inner classes are inner classes marked with the static modifier.

Symbolic Link

A symbolic link is a file that contains a reference to another target file or directory.

T

Task

A task is an object that implements the Runnable or Callable interface.

Thread

A thread can be defined as a single sequential flow of control within a program.

Thread priorities

Thread priorities are integers in the range of 1 to 10 that specify the priority of one thread with respect to the priority of another thread.

Type Inference

Type inference enables you to invoke the constructor of a generic class with an empty set of type parameters, <>.

Type Parameter

The type parameter section is represented by angular brackets, < >, that can have one or more types of parameters separated by commas.

U

Upcasting

Upcasting is usually done along the class hierarchy in a direction from the derived class to the base class.

X

XOPEN

The XOPEN error code is a standard message associated with an error that can identify the error across multiple databases.

Appendix

Identifying an Object's Life Cycle

An object goes through the following phases in its lifetime:

1. **Creation**: An object of a class is created using the `new` operator. This operator allocates the memory to an object and returns the memory reference. This memory reference is assigned to a reference variable, which can be used to access the object. In addition, the `new` operator is followed by a call to the constructor, which initializes the object.
2. **Alive**: The object lives its life by providing access to the methods and fields of the class to which the object belongs.
3. **Garbage collection**: An object becomes eligible for garbage collection when it is not used anymore. Garbage collection is a process in which a special daemon thread named garbage collector removes any reference variables pointing to that specific object. This process is called dereferencing the object. Then, garbage collector removes the object from the memory. At this point, the object ceases to exist.

Formatting Strings Using the Formatting Parameters

A formatted string is a string that can display contents with various formatting styles. A formatted string at the minimum consists of two items. The first item is a string with one or more format specifiers. A format specifier begins with a percent sign (%) and is followed by a conversion specifier. For example, the format specifier for the floating-point type of data is %f. This string is followed by the argument variables whose values must be used by the format specifiers. In general, the number of argument variables and format specifiers must be same. In addition, the format specifiers and arguments are matched in order from left to right. Java supports various format specifiers.

The following table lists some of the commonly used format specifiers.

Format Specifiers	***Type***
%b or %B	*Boolean*
%c or %C	*Character*
%d	*Decimal*
%f	*Floating-point*
%s or %S	*String*
%t or %T	*Time and date*

The Format Specifiers

Consider the following code snippet that demonstrates the use of a formatted string:

```
int outof = 100;
double scored = 78.6;
String name = "John";
System.out.printf("%s has got %f score out of %d", name, scored, outof);
```

Once the preceding code snippet is executed, the following output is displayed:

```
John has got 78.600000 score out of 100
```

In the preceding code snippet, the %s, %f, and %d format specifiers are replaced with the arguments that follow the format strings. %s is replaced with the value of the name variable. %f is replaced with the value of the scored variable and %d is replaced with the value of the outof variable.

Certification Mapping

The following table lists the relevant exam objectives of Java Exam 1Z0-803 (Java SE 7 Programmer I) and the corresponding chapter numbers and section names.

Exam Objectives	***Chapter Name***	***Section Name***
Java Basics		
Define the scope of variables	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Define the structure of a Java class	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Create executable Java applications with a main method	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Import other Java packages to make them accessible in your code	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Working With Java Data Types		
Declare and initialize variables	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Differentiate between object reference variables and primitive variables	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Read or write to object fields	*Programming in Java: Chapter 3*	*Implementing Type-safety*
Explain an Object's Lifecycle (creation, "dereference" and garbage collection)	*Appendix*	*NA*
Call methods on objects	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Manipulate data using the StringBuilder class and its methods	*Introduction to Java: Chapter 4*	*Manipulating Strings*
Creating and manipulating Strings	*Introduction to Java: Chapter 4*	*Manipulating Strings*

Exam Objectives	***Chapter Name***	***Section Name***
Using Operators and Decision Constructs		
Use Java operators	*Introduction to Java: Chapter 2*	*Working with Operators*
Use parenthesis to override operator precedence	*Introduction to Java: Chapter 2*	*Using Operator Precedence*
Test equality between Strings and other objects using == and equals ()	*Introduction to Java: Chapter 2 and 4*	*Working with Operators, Manipulating Strings*
Create if and if/else constructs	*Introduction to Java: Chapter 3*	*Working with Conditional Constructs*
Use a switch statement	*Introduction to Java: Chapter 3*	*Working with Conditional Constructs*
Creating and Using Arrays		
Declare, instantiate, initialize and use a one-dimensional array	*Introduction to Java: Chapter 4*	*Manipulating Arrays*
Declare, instantiate, initialize and use multi-dimensional array	*Introduction to Java: Chapter 4*	*Manipulating Arrays*
Declare and use an ArrayList	*Programming in Java: Chapter 4*	*Using the List Interface*
Using Loop Constructs		
Create and use while loops	*Introduction to Java: Chapter 3*	*Working with Loop Constructs*
Create and use for loops including the enhanced for loop	*Introduction to Java: Chapter 3 and 4*	*Working with Loop Constructs, Manipulating Arrays*
Create and use do/while loops	*Introduction to Java: Chapter 3*	*Working with Loop Constructs*
Compare loop constructs	*Introduction to Java: Chapter 3*	*Working with Loop Constructs*
Use break and continue	*Introduction to Java: Chapter 3*	*Working with Conditional Constructs*

Exam Objectives	***Chapter Name***	***Section Name***
Working with Methods and Encapsulation		
Create methods with arguments and return values	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Apply the static keyword to methods and fields	*Introduction to Java: Chapter 1*	*Accessing Class Members*
Create an overloaded method	*Introduction to Java: Chapter 5*	*Implementing Polymorphism*
Differentiate between default and user defined constructors	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Create and overload constructors	*Introduction to Java: Chapter 5*	*Implementing Polymorphism*
Apply access modifiers	*Introduction to Java: Chapter 1*	*Accessing Class Members*
Apply encapsulation principles to a class	*This is covered in the OOPS course, which is a prerequisite for the Introduction to Java course.*	*NA*
Determine the effect upon object references and primitive values when they are passed into methods that change the values	*Introduction to Java: Chapter 7*	*Exploring UI Components*
Working with Inheritance		
Implement inheritance	*Introduction to Java: Chapter 5*	*Implementing Inheritance*
Develop code that demonstrates the use of polymorphism	*Introduction to Java: Chapter 5*	*Implementing Polymorphism*
Differentiate between the type of a reference and the type of an object	*Programming in Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Determine when casting is necessary	*Programming in Java: Chapter 1*	*Implementing Type Casting*

Exam Objectives	***Chapter Name***	***Section Name***
Use super and this to access objects and constructors	*Introduction to Java: Chapter 5*	*Implementing Inheritance*
Use abstract classes and interfaces	*Introduction to Java: Chapter 5*	*Implementing Inheritance*
Handling Exceptions		
Differentiate among checked exceptions, RuntimeExceptions and Errors	*Introduction to Java: Chapter 6*	*Handling Exceptions*
Create a try-catch block and determine how exceptions alter normal program flow	*Introduction to Java: Chapter 6*	*Handling Exceptions*
Describe what Exceptions are used for in Java	*Introduction to Java: Chapter 6*	*Handling Exceptions*
Invoke a method that throws an exception	*Introduction to Java: Chapter 6*	*Handling Exceptions*
Recognize common exception classes and categories	*Introduction to Java: Chapter 6*	*Handling Exceptions*

Mapping of Exam Objectives with the Corresponding Chapters of the Course

Note

To prepare for the exam, you need to refer to the Student Guide as well as the Activity Book.

The following table lists the relevant exam objectives of Java Exam 1Z0-804 (Java SE 7 Programmer) and the corresponding chapter numbers and section names.

Exam Objectives	***Chapter Name***	***Section Name***
Java Class Design		
Use access modifiers: private, protected, and public	*Introduction to Java: Chapter 1*	*Accessing Class Members*

Exam Objectives	***Chapter Name***	***Section Name***
Override methods	*Introduction to Java: Chapter 5*	*Implementing Polymorphism*
Overload constructors and other methods appropriately	*Introduction to Java: Chapter 5*	*Implementing Polymorphism*
Use the instanceof operator and casting	*Introduction to Java: Chapter 2*	*Working with Operators*
Use virtual method invocation	*Programming in Java: Chapter 1*	*Implementing Type Casting*
Override methods from the Object class to improve the functionality of your class	*Programming in Java: Chapter 5 and 6*	*Using Threads in Java, Implementing Thread Synchronization*
Use package and import statements	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Advanced Class Design		
Identify when and how to apply abstract classes	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Construct abstract Java classes and subclasses	*Introduction to Java: Chapter 5*	*Implementing Inheritance*
Use the static and final keywords	*Introduction to Java: Chapter 1*	*Accessing Class Members*
Create top-level and nested classes	*Introduction to Java: Chapter 1*	*Identifying the Building Blocks of a Java Program*
Use enumerated types	*Introduction to Java: Chapter 4*	*Manipulating Enums*
Object-Oriented Design Principles		
Write code that declares, implements and/or extends interfaces	*Introduction to Java: Chapter 5*	*Implementing Inheritance*
Choose between interface inheritance and class inheritance	*Introduction to Java: Chapter 5*	*Implementing Inheritance*

Exam Objectives	***Chapter Name***	***Section Name***
Develop code that implements "is-a" and/or "has-a" relationships	*This is covered in the OOPS course, which is a prerequisite for the Introduction to Java course.*	*NA*
Apply object composition principles	*This is covered in the OOPS course, which is a prerequisite for the Introduction to Java course.*	*NA*
Design a class using the Singleton design pattern	*http://www.oodesign.com/singleton-pattern.html*	*NA*
Write code to implement the DAO pattern	*http://www.oracle.com/technetwork/java/dataaccessobject-138824.html*	*NA*
Design and create objects using a factory, and use factories from the API	*http://www.oracle.com/technetwork/java/dataaccessobject-138824.html*	*NA*
Generics and Collections		
Create a generic class	*Programming in Java: Chapter 3*	*Creating User-defined Generic Classes and Methods*
Use the diamond syntax to create a collection	*Programming in Java: Chapter 3 and 4*	*Implementing Type-safety, Implementing Sorting*
Analyze the interoperability of collections that use raw type and generic types	*Programming in Java: Chapter 4*	*Using the Set Interface*
Use wrapper classes and autoboxing	*Introduction to Java: Chapter 1, and Programming in Java: Chapter 1*	*Identifying the Building Blocks of a Java Program, Implementing Type Casting*
Create and use a List, a Set and a Deque	*Programming in Java: Chapter 4*	*Using the Set Interface, Using the List Interface, Using the Deque Interface*
Create and use a Map	*Programming in Java: Chapter 4*	*Using the Map Interface*

Exam Objectives	***Chapter Name***	***Section Name***
Use java.util.Comparator and java.lang.Comparable	*Programming in Java: Chapter 4*	*Implementing Sorting*
Sort and search arrays and lists	*Programming in Java: Chapter 4*	*Implementing Sorting*
String Processing		
Search, parse and build strings	*Programming in Java: Chapter 2*	*Processing Strings Using Regex*
*Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b. \B, [], ().*	*Programming in Java: Chapter 2*	*Processing Strings Using Regex*
Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings.	*Appendix*	*NA*
Exceptions and Assertions		
Use throw and throws statements	*Introduction to Java: Chapter 5*	*Handling Exceptions*
Use the try statement with multi-catch, and finally clauses	*Introduction to Java: Chapter 5*	*Handling Exceptions*
Autoclose resources with a try-with-resources statement	*Introduction to Java: Chapter 5*	*Handling Exceptions*
Create custom exceptions	*Introduction to Java: Chapter 5*	*Handling Exceptions*
Test invariants by using assertions	*Introduction to Java: Chapter 6*	*Using the assert Keyword*
Java I/O Fundamentals		
Read and write data from the console	*Programming in Java: Chapter 7*	*Working with Input Stream, Working with Output Stream*
Use streams to read and write files	*Programming in Java: Chapter 7*	*Working with Input Stream, Working with Output Stream*

Exam Objectives	***Chapter Name***	***Section Name***
Java File I/O (NIO.2)		
Use the Path class to operate on file and directory paths	*Programming in Java: Chapter 8*	*Introducing NIO*
Use the Files class to check, delete, copy, or move a file or directory	*Programming in Java: Chapter 8*	*Introducing NIO*
Read and change file and directory attributes	*Programming in Java: Chapter 8*	*Introducing NIO*
Recursively access a directory tree	*Programming in Java: Chapter 8*	*Introducing NIO*
Find a file by using the PathMatcher class	*Programming in Java: Chapter 8*	*Introducing NIO*
Watch a directory for changes by using WatchService	*Programming in Java: Chapter 8*	*Introducing NIO*
Building Database Applications with JDBC		
Define the layout of the JDBC API	*Programming in Java: Chapter 9*	*Using JBDC API*
Connect to a database by using a JDBC driver	*Programming in Java: Chapter 9*	*Using JDBC API*
Update and query a database	*Programming in Java: Chapter 9*	*Using JDBC API*
Customize the transaction behavior of JDBC and commit transactions	*Programming in Java: Chapter 10*	*Managing Database Transactions*
Use the JDBC 4.1 RowSetProvider, RowSetFactory, and RowSet interfaces	*Programming in Java: Chapter 9*	*Accessing Result Sets*
Threads		
Create and use the Thread class and the Runnable interface	*Programming in Java: Chapter 5*	*Creating Threads*

Exam Objectives	***Chapter Name***	***Section Name***
Manage and control thread lifecycle	*Programming in Java: Chapter 5*	*Using Threads in Java*
Synchronize thread access to shared data	*Programming in Java: Chapter 6*	*Implementing Thread Synchronization*
Identify potential threading problems	*Programming in Java: Chapter 5*	*Using Threads in Java*
Concurrency		
Use java.util.concurrent collections	*Programming in Java: Chapter 6*	*Implementing Concurrency*
Apply atomic variables and locks	*Programming in Java: Chapter 6*	*Implementing Concurrency*
Use Executors and ThreadPools	*Programming in Java: Chapter 6*	*Implementing Concurrency*
Use the parallel Fork/Join Framework	*Programming in Java: Chapter 6*	*Implementing Concurrency*
Localization		
Read and set the locale by using the Locale object	*Programming in Java: Chapter 2*	*Implementing Localization*
Build a resource bundle for each local	*Programming in Java: Chapter 2*	*Implementing Localization*
Load a resource bundle in an application	*Programming in Java: Chapter 2*	*Implementing Localization*
Format text for localization by using NumberFormat and DateFormat	*Programming in Java: Chapter 2*	*Implementing Localization*

Mapping of Exam Objectives with the Corresponding Chapters of the Course

Note

The reference links are provided for the following topics, as they are out of scope and could not be covered in the course:

- *Design a class using the Singleton design pattern.*
- *Write code to implement the DAO pattern.*
- *Design and create objects using a factory, and use factories from the API.*

Objectives Attainment Feedback

Programming in Java

Name: ____________________ Batch: ____________________ Date: ____________________

The objectives of this course are listed below. Please tick whether the objectives were achieved by you or not. Calculate the percentage at the end, fill in your name and batch details, and return the form to your coordinator.

S. No.	*Objectives*	*Yes*	*No*
1.	Create inner classes	□	□
2.	Implement type casting	□	□
3.	Process strings using regex	□	□
4.	Implement localization	□	□
5.	Create user-defined generic classes and methods	□	□
6.	Implement type-safety	□	□
7.	Use the Set interface	□	□
8.	Use the List interface	□	□
9.	Use the Map interface	□	□
10.	Use the Deque interface	□	□
11.	Implement sorting	□	□
12.	Use threads in Java	□	□
13.	Create threads	□	□

S. No.	*Objectives*	*Yes*	*No*
14.	Implement thread synchronization	□	□
15.	Implement concurrency	□	□
16.	Work with input stream	□	□
17.	Work with output stream	□	□
18.	Get familiar with NIO	□	□
19.	Perform the read and write operations on a file	□	□
20.	Identify the layers in the JDBC architecture	□	□
21.	Identify the types of JDBC drivers	□	□
22.	Use JDBC API	□	□
23.	Access Result Sets	□	□
24.	Create applications using the PreparedStatement object	□	□
25.	Manage database transactions	□	□
26.	Implement batch updates in JDBC	□	□
27.	Create and call stored procedures in JDBC	□	□
28.	Use metadata in JDBC	□	□
	Percentage: (# of Yes/28) * 100		