

Object-oriented Programming in Java



Object-oriented Programming in Java Learner's Guide

© 2013 Aptech Limited

All rights reserved.

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

All trademarks acknowledged.

APTECH LIMITED

E-mail: ov-support@onlinevarsity.com

First Edition - 2013



Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc.* These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

➤ Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

➤ Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

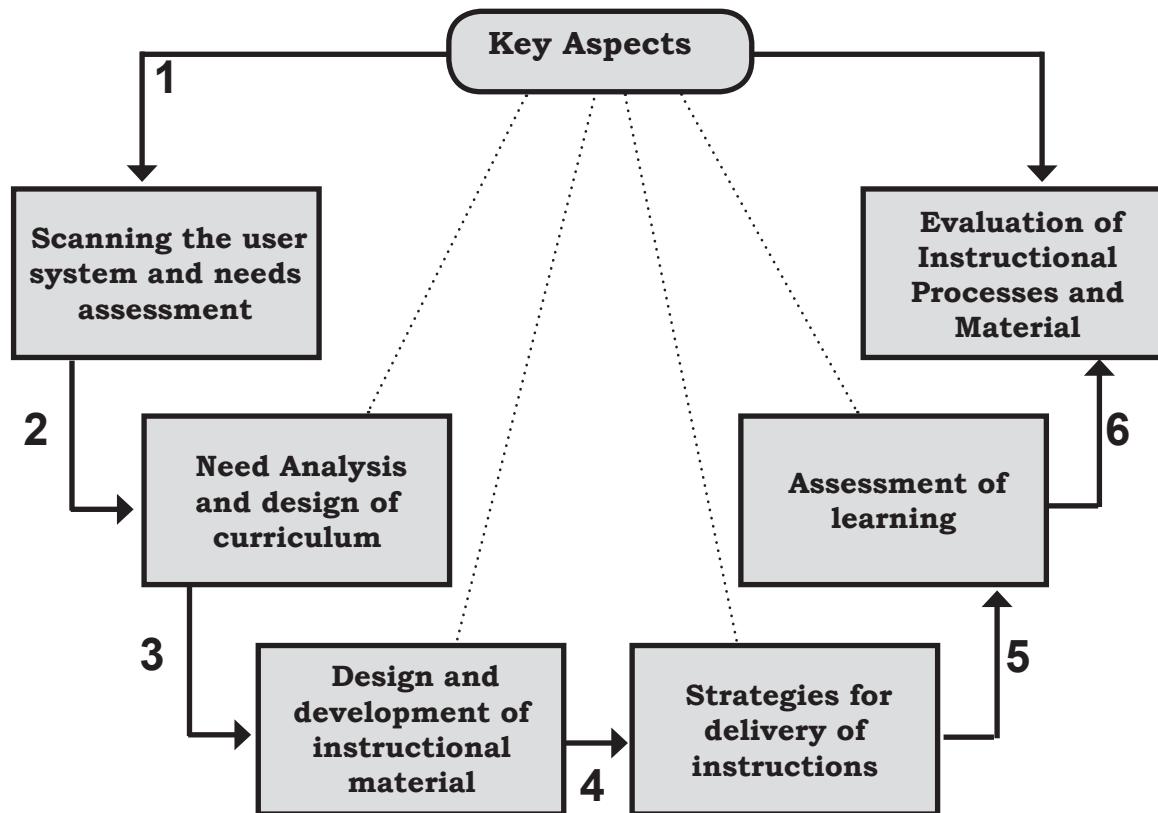
➤ Evaluation of instructional process and instructional materials

The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model



Are you looking for online **HELP?**

We are just a *click* away

To chat with a



Login to **www.onlinevarsity.com**

Preface

The book, Object-oriented Programming in Java, aims to enable the students to use the advanced object-oriented features of Java. This book intends to familiarize the reader with the latest version of Java, that is, Java SE 7. Java SE 7 introduces several new features as well as provides enhancements on the earlier versions. The book begins with the creation of user-defined exceptions and assertions in Java. It explains the `java.lang` package classes and explains storing of objects in different type of collections. The book proceeds with the explanation of the basic concept of multithread programming and its relevance in Java in terms of threads.

The book also explains the concepts of collections, regular expressions and covers various Application Programming Interfaces (APIs) in Java programming language. These include File Input/Output, Concurrency, Generics, and so on.

The book provides the explanation of Java Database Connectivity (JDBC) API 4.0/4.1 and its interfaces to develop Java applications that interact with database objects to persist data in them. The book also discusses on the various design patterns and internationalization features that are supported in Java.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

TechnoWise

Nothing is a waste of time if you

use the experience wisely



Are you a
TECHIE GEEK
looking for updates?

Logon to

www.onlinevarsity.com

Table of Contents

Sessions

1. Exceptions and Assertions
2. java.lang Package
3. Collections API
4. Generics
5. File Handling in Java
6. New Features in File Handling
7. Introduction to Threads
8. Multithreading and Concurrency
9. JDBC API
10. Advanced JDBC Features
11. Design Patterns
12. Internationalization and Localization



Balanced Learner-Oriented Guide

for enriched learning available



Session 1

Exceptions and Assertions

Welcome to the Session, **Exceptions and Assertions**.

This session explains the creation and use of exceptions in Java. It deals with creation of user-defined exceptions and throwing exceptions from methods. Further, this session explains the concept and use of Assertions. Lastly, the session explains the different types of assertions and its limitations.

In this Session, you will learn to:

- Describe exception
- Explain the use of try-catch-finally blocks
- Explain throwing and handling exceptions
- Explain handling multiple exceptions in a single catch block
- Explain the use of try-with-resources
- Describe creation and use of custom exceptions
- Explain assertions and its types



1.1 Introduction

Java programming language provides the ability to handle unexpected events in the program. This is because, even though a code is well written, it is prone to behave erroneously. With the help of exception handling, the developer can ensure that the user gets a proper message in case some error occurs in the program. It also helps to ensure that in case of an error, the data and processes that the program is using do not get affected adversely.

1.2 Overview of Exceptions

An exception is an event occurring during program execution that leads to disruption of the normal flow of the program's instructions. Exception handling in Java is a way for ensuring smooth execution of a program. To handle the exceptions, Java provides the `try-catch-finally` blocks. Using these blocks, the developer can check the program statements for errors and handle them in case they occur.

1.2.1 `try-catch-finally` Block

In order to handle exceptions, the developer needs to identify the statements that may lead to exceptions and enclose them within a `try` block. Next, exception handlers need to be associated with a `try` block by providing one or more `catch` blocks directly after the `try` block.

The syntax of a `try-catch` block is as follows:

Syntax:

```
try {
    . . .
} catch (ExceptionType name) {
} catch (ExceptionType name) {
}
. . .
```

Each `catch` block acts as an exception handler that handles a particular type of exception. The parameter, `ExceptionType`, indicates the type of exception that the handler can handle. This parameter must be the name of a class inheriting from the `Throwable` class.

The code within the `catch` block is executed if and when the exception handler is invoked. The first `catch` block in the call stack whose exception type matches the type of the exception thrown is invoked by the runtime system to handle the exception.

Code Snippet 1 shows an example of `try-catch` block.

Code Snippet 1:

```
public class Calculator{
    int result;
    public void divide(int num1, int num2) {
        try{
            result = num1/num2; //line 1 -- statement that might raise exception
        } catch(ArithmaticException ex) {
            // printing the error message
            System.out.println("Denominator cannot be set to zero!!!"+ex.getMessage());
        }
    }
}
```

In the code, line 1 might raise an exception when the value of `num2` is set to 0. Therefore, a `catch` block is provided to handle the exception. The class that will handle the exception is `ArithmaticException`. When the exception occurs, it will be raised in the `try` block and handled by the `catch` block. The statement within the `catch` block will be executed to inform the user of the error. The statement provides a user-friendly message along with a built-in error message using the `getMessage()` method.

The `finally` block is executed even if an exception occurs in the `try` block. It helps the programmer to ensure that the cleanup code does not get accidentally bypassed by a `break`, `continue`, or `return` statement in the `try` block. It is advisable to write the cleanup code in a `finally` block, even when no exceptions are expected.

Some conditions in which the `finally` block may not execute are as follows:

- When the Java Virtual Machine (JVM) exits while the `try` or `catch` code is being executed.
- If the thread executing the `try` or `catch` code is interrupted or killed.

In these cases, the `finally` block may not execute even though the application as a whole continue execution.

Code Snippet 2 explains the use of the `finally` block.

Code Snippet 2:

```
...
PrintWriter objPwOut=null; // PrintWriter object
public void writeToFile{
    try {
        // initializing the PrintWriter with a file name
        objPwOut = new PrintWriter("C:\\\\MyFile.txt");
    } catch (FileNotFoundException ex) {
        // printing the error message
        System.out.println("File Does not Exist " + ex.getMessage());
    } finally {
        // verifying if the PrintWriter object is still open
        if (objPwOut != null) {
            // closing the PrintWriter object
            objPwOut.close();
            System.out.println("PrintWriter closed");
        }
    }
}
```

In the code, an object of the `PrintWriter` class is created with the file name `MyFile.txt` as an argument. Now, during execution, when the `PrintWriter` tries to access the file `MyFile.txt`, it might throw an exception if the file does not denote an existing, writable, regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file. To inform the user of the situation, a corresponding `catch` block is provided that handles the `FileNotFoundException`. Also, to ensure that after the exception is caught, the `PrintWriter` object is not left open, the `finally` block is provided in which the `PrintWriter` object is closed. Thus, the cleanup code is executed despite of an exception as it was written in the `finally` block.

1.2.2 *throw* and *throws* Keywords

At times, it might be required to let a method transfer the control further up the call stack to handle the exception. For example, while making connection to servers, one might not be able to anticipate the type of format in which the server id will be provided by the user. In this case, it is advisable not to catch the exception and to allow a method further up the call stack to handle it.

If a method does not catch the checked exceptions that can occur within it, it must specify that it can throw these exceptions. For example, the `writeToFile()` method in Code Snippet 2 can be modified to specify the exceptions it can throw instead of catching them. This can be done by adding the `throws` clause to the method declaration followed by a comma-separated list of all the exceptions that the method is liable to throw.

The `throws` clause is written after the method name and argument list and before the opening brace of the method.

Code Snippet 3 shows the modified `writeToFile()` method with the `throws` clause.

Code Snippet 3:

```
...
PrintWriter objPwOut=null;
public void writeToFile throws FileNotFoundException{
    try {
        objPwOut=new PrintWriter("C:\\\\MyFile.txt");
    } finally {
        if (objPwOut != null) {
            objPwOut.close();
            System.out.println("PrintWriter closed");
        }
    }
}
```

Now, to catch the exception further up in the hierarchy, at some point, the code must throw the exception. The exception can be thrown from anywhere such as from the current file, or a file in another package, or packages that come with the Java platform, or the Java runtime environment. Regardless of which code throws the exception, it is always thrown using the `throw` statement.

The Java platform provides several exception classes which are descendants of the `Throwable` class. These classes allow programs to differentiate among the different types of exceptions that can occur during the execution of a program.

All methods use the `throw` statement to throw an exception. The `throw` statement takes a single argument which is a `Throwable` object. `Throwable` objects are instances of any subclass of the `Throwable` class.

The syntax of `throw` statement is as follows:

Syntax:

```
throw <ThrowableObject>;
```

For example, the `writeToFile()` method in Code Snippet 3 can be made to throw the `FileNotFoundException` as shown in Code Snippet 4.

Code Snippet 4:

```
import java.io.*;
public class FileWriting {
    PrintWriter objPwOut=null;
    // Declares the exception in the throws clause
    public void writeToFile() throws FileNotFoundException
    {
        try {
            objPwOut = new PrintWriter("C:\\\\MyFile.txt");
        } catch (FileNotFoundException ex) {
            // Re-throwing the exception
            throw new FileNotFoundException();
        } finally {
            if (objPwOut != null) {
                objPwOut.close();
                System.out.println("PrintWriter closed");
            }
        }
    }
    public static void main(String[] args)
    {
        try{
            FileWriting fw=new FileWriting();
            fw.writeToFile();
        } catch(FileNotFoundException ex)
        {
            // Catching the exception
            System.out.println("File does not Exist "+ex.getMessage());
        }
    }
}
```

In the code, the `FileNotFoundException` is caught by the `catch` block of the `writeToFile()` method. However, the exception is not handled within this block, but is thrown back using the `throw new FileNotFoundException()` statement. Thus, the exception is transferred further up in the call stack. Now, the caller of the `writeToFile()` method will have to handle this exception. Since, `main()` is the caller method, the exception will have to be handled by `main()`. Hence, a `catch` block for `FileNotFoundException` has been provided in the `main()` method where the exception will be handled.

1.2.3 Throwing Exceptions from Methods

One can directly throw exceptions from a method and transfer it to a method higher up in the hierarchy. Consider the `Calculator` class shown in Code Snippet 5.

Code Snippet 5:

```
public class Calculator {
    public void divide(int a, int b) throws ArithmeticException {
        if (b==0) {
            throw new ArithmeticException(); // throwing exception
        }
        int result = a/b;
        System.out.println("Result is "+result);
    }
}

public class TestCalculator {
    public static void main(String[] args) {
        try{
            Calculator objCalc = new Calculator();
            // Invoking the divide() method
            objCalc.divide(Integer.parseInt(args[0]), Integer.parseInt(args[1]));
        } catch(ArithmeticException ex) {
            System.out.println("Denominator cannot be set to zero");
        }
    }
}
```

In the code, the `divide()` method declares the `ArithmaticException` in the `throws` clause. Now, if the value of denominator `b` is set to 0 at runtime, the method throws the `ArithmaticException` using the `throw` statement. Thus, the exception is thrown from the method to be handled by another method further up in the call stack. Here, the `main()` method consists of the `catch` block to handle the exception and print the appropriate message to the user.

1.2.4 Handling Multiple Exceptions in a Single catch Block

Java SE 7 and later versions provide the feature of handling more than one exception in a single `catch` block. This feature helps to reduce code duplication and prevent the use of a much generalized exception.

To create a multiple exception catch block, specify the types of exceptions that catch block can handle separated by a vertical bar (|) as follows:

Syntax:

```
catch (ExceptionType1|ExceptionType2 ex) {
    // statements
}
```

Since, the `catch` block handles more than one type of exception, then the `catch` parameter is implicitly final. Therefore, one cannot assign any values to it within the `catch` block.

Code Snippet 6 shows an example of handling multiple exceptions in a single `catch` block.

Code Snippet 6:

```
public class Calculator {
    public static void main(String[] args)
    {
        int result, sum=0;
        int marks[] = {20, 30, 50};
        try{
            result = Integer.parseInt(args[0]) / Integer.parseInt(args[1]) // line 1
            System.out.println("Result is " + result);
            for(int i=0; i<4; i++){
                sum += marks[i]; // line 2
            }
            System.out.println("Sum is " + sum);
        }
    }
}
```

```
} catch(ArrayIndexOutOfBoundsException|ArithmetiException ex) {  
    // Catching multiple exceptions  
    throw ex;  
}
```

In the code, line 1 is liable to raise the `ArithmeticException` if `args[1]` is specified as 0 at runtime. Similarly, line 2 will raise the `ArrayIndexOutOfBoundsException` exception since the termination condition of the loop is `i<4` whereas the array `marks[]` has only three elements. To handle both the exception types, a single `catch` block has been defined. It will display the error message depending on the type of exception caught, that is, `ArrayIndexOutOfBoundsException` or `ArithmeticException`. The `throw` keyword is used to throw the exception that has been raised to the output stream.

Figure 1.1 shows the output when `args[0]=20` and `args[1]=0` and when `args[1]=10`.

```
args[0]=20 and args[1]=0
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at testproject.Calculator.main(Calculator.java:18)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)

args[0]=20 and args[1]=10
run:
Result is 2
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at testproject.Calculator.main(Calculator.java:23)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 1.1: Multiple Exceptions – Single catch Block

Figure 1.1 shows two output windows. The first output is when `args[1]` is set to 0. In this case, line 1 will raise an exception and the control is transferred to the catch block. Here, the type of exception is identified as `ArithmaticException` and the appropriate message is displayed to the user using the `throw` keyword. The rest of the `try` block is not executed. Therefore, even though line 2 is erroneous, that exception is not raised.

However, in the second output, when `args[1]` is set to 10 or any non-zero number, line 1 executes successfully, but line 2 raises the `ArrayIndexOutOfBoundsException` when the loop count reaches three. The exception is caught in the `catch` block and the appropriate message is displayed to the user along with the result of line 1. Thus, more than one type of exception gets handled in a single `catch` block.

1.3 Using try-with-resources and AutoCloseable Interface

The try-with-resources statement is a `try` statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement is written to ensure that each resource is closed at the end of the statement. Any object that implements `java.lang.AutoCloseable`, which includes all the objects which implement `java.io.Closeable`, can be used as a resource. The `AutoCloseable` interface is used to close a resource when it is no longer needed.

One can find the list of classes that implement either of the interfaces, `AutoCloseable` and `Closeable`, in the Javadoc for these interfaces. The `Closeable` interface extends the `AutoCloseable` interface. A `Closeable` is a source or destination of data that can be closed. The `close()` method is invoked to release resources that the object is holding, such as open files.

The `close()` method of the `Closeable` interface throws exceptions of type `IOException` while the `close()` method of the `AutoCloseable` interface throws exceptions of type `Exception`. Consequently, subclasses of the `AutoCloseable` interface can override this behavior of the `close()` method to throw specialized exceptions, such as `IOException`, or no exception at all.

Code Snippet 7 explains the use of try-with-resources.

Code Snippet 7:

```
public void writeToFile(String path) {
    // Creating a try-with-resources statement
    try (Writer output = new BufferedWriter(new FileWriter(path))) {
        output.write("This is a sample statement.");
    } catch (IOException ex) {
        System.out.println(ex.getMessage());
    }
}
```

In the code, an object of the `Writer` class has been created and initialized with the reference of `BufferedWriter` that writes a line to the file specified by the variable `path`. Here, `BufferedWriter` is a resource that must be closed after the program is finished using it. The declaration statement appears within the parenthesis immediately after the `try` keyword. In Java SE 7 and later, the class `BufferedWriter` implements the `AutoCloseable` interface. Since the `BufferedWriter` instance is declared in a try-with-resources statement, it will be closed regardless of whether the `try` statement completes normally or abruptly due to an exception thrown by the `write()` method.

Note - A try-with-resources statement can have `catch` and `finally` blocks just like an ordinary `try` statement. In a try-with-resources statement, any `catch` or `finally` block is run after the resources declared have been closed.

Prior to Java SE 7, the `finally` block was used to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly.

Code Snippet 8 shows an example that uses a `finally` block instead of a `try-with-resources` statement to close the resources.

Code Snippet 8:

```
static void writeToFile(String path) throws IOException {
    Writer output = new BufferedWriter(new FileWriter(path));
    try {
        output.write("This is a sample statement.");
    } finally {
        if (output != null)
            output.close();
    }
}
```

However, in this example, if the methods `write()` and `close()` both throw exceptions, then the method `writeToFile()` throws the exception that is thrown from the `finally` block; the exception thrown from the `try` block is suppressed.

In contrast, if exceptions are thrown from both; `try` block and the `try-with-resources` statement, then the method `writeToFile()` throws the exception that is thrown from the `try` block; the exception thrown from the `try-with-resources` block is suppressed. However, in Java SE 7 and later, one can retrieve suppressed exceptions also.

Java SE 7 allows declaring one or more resources in a `try-with-resources` statement.

Code Snippet 9 shows an example that declares more than one resource in a single `try-with-resources` statement.

Code Snippet 9:

```
public static void writeToFileContents(String sourceFile, String targetFile)
throws java.io.IOException {
    // Declaring more than one resource in the try-with-resources statement
    try (
        BufferedReader objBr = new BufferedReader(new FileReader(sourceFile));
        BufferedWriter output = new BufferedWriter(new FileWriter(targetFile))
    )
```

```
{
    // code to read from source and write to target file.
}
}
```

In this example, the try-with-resources statement contains two declarations that are separated by a semicolon: `BufferedReader` and `BufferedWriter`. When the block of code that directly follows the declaration terminates, either normally or due to an exception, the `close()` methods of the `BufferedWriter` and `BufferedReader` objects are automatically called in this order. That is, the `close()` methods of resources are called in the opposite order of their creation.

1.4 Enhancements In Exceptions in Java SE 7

The following are the enhancements that have already been covered:

- Multi-catch statements has helped the programmers to program more efficiently and concisely.
- Multi-catch statements also allow the programmer to handle a part of the exception and let it bubble up using the re-throw.
- Try-with-resources statement facilitates less error-prone exception cleanup.

1.5 User-defined Exceptions

While deciding on the type of exception to throw, one can either use the built-in exception classes of the Java platform or create a custom exception class. One can create a custom exception class when:

- The built-in exception type does not fulfill the requirement.
- It is required to differentiate your exceptions from those thrown by classes written by other vendors.
- The code throws more than one related exception.

1.5.1 Creating a User-defined Exception

To create a user-defined exception class, the class must inherit from the `Exception` class. The syntax is as follows:

Syntax:

```
public class <ExceptionName> extends Exception {}
```

where,

`<ExceptionName>`: is the name of the user-defined exception class.

Code Snippet 10 explains creation of a user-defined exception class.

Code Snippet 10:

```
// Creating a user-defined exception class
public class ServerException extends Exception{
    public ServerException()
    {}

    // Overriding the getMessage() method
    @Override
    public String getMessage() // line 1
    {
        return "Connection Failed";
    }
}
```

In the code, **ServerException** is a user-defined exception class that inherits from the built-in `Exception` class. The `getMessage()` method of the `Exception` class has been overridden in the **ServerException** class to print a user-defined message “**Connection Failed**”.

1.5.2 Throwing User-defined Exceptions

To raise a user-defined exception, a method must throw the exception at runtime. The exception is transferred further up in the call stack and handled by the caller of the method.

Code Snippet 11 explains how to throw a user-defined exception.

Code Snippet 11:

```
// creating a class to use the user-defined exception
class MyConnection {

    String ip;
    String port;

    public MyConnection()
    {}

    public MyConnection(String ip, String port) {
```

```

this.ip=ip;
this.port=port;
}
// creating a method that throws the user-defined exception
public void connectToServer() throws ServerException{ // line 1

if(ip.equals("127.10.10.1") && port.equals("1234"))
    System.out.println("Connecting to Server...");
else
    throw new ServerException(); // line 2 – throwing the exception
}
}

```

In the code, the **MyConnection** class consists of a constructor that accepts the ip address and port number of the server and initializes the respective instance variables. The **connectToServer()** method declares that it throws the **ServerException** by using the **throws** clause on line 1. Inside the method, the value of **ip** and **port** variables is verified. If the values do not match the pre-defined values, the method throws the **ServerException** on line 2. Now, in the class **TestConnection**, an object of **MyConnection** is created with incorrect values for **ip** and **port** within the **try** block. Next, on line 3, the **connectToServer()** method has been invoked. Also, the statement has been included within a catch block to handle the user-defined exception, **ServerException**. The statement, **ex.getMessage()** is used to invoke the overridden **getMessage()** method of the **ServerException** class to print the user-defined error message. Figure 1.2 shows the output of the code.

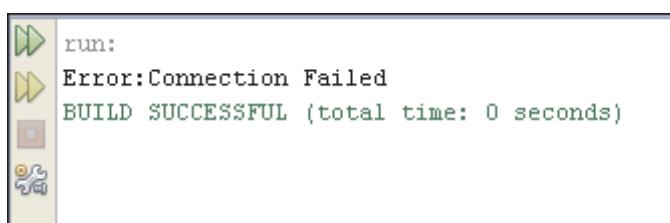


Figure 1.2: User-Defined Exception

The output displays the user-defined error message from the **getMessage()** method of the **ServerException** class. This is because, when the **connectToServer()** method is executed in **main()** method, the **ip** and **port** values do not match the pre-defined values. Hence, the method throws the **ServerException** exception. This is caught in the catch block of **main()** method which invokes the **getMessage()** method of the **ServerException** object. Thus, a custom exception class can be created and exceptions can be handled when the built-in exception classes do not serve the purpose.

1.6 Wrapper Exceptions

To hide the type of exception that is being generated without exactly swallowing the exception, one can use wrapper exceptions. This will lead to implementation substitution. That is, a wrapper exception is used to preserve the abstraction and avoid swallowing of the exceptions. Thus, exception wrapping is catching an exception, wrapping it in a different exception, and throwing the wrapper exception.

Exception wrapping is a standard feature in Java since JDK 1.4. Most of Java's built-in exceptions have constructors that can take a 'cause' parameter. They also provide a `getCause()` method that will return the wrapped exception.

The main reason for using exception wrapping is to prevent the code further up the call stack from knowing about every possible exception in the system. The reason for this is that declared exceptions tend to aggregate towards the top of the call stack. If exceptions are not wrapped, but instead passed on by declaring the methods to throw them, one may end up with top level methods that declare many different exceptions. Declaring all these exceptions in each method to back up the call stack becomes tedious.

Also, one may not want the top level components to know anything about the bottom level components, nor the exceptions they throw. For example, the purpose of **Data Access Object (DAO) interfaces and implementations is to abstract the details of data access from the rest of the application**. Now, if the DAO methods throw `SQLException` then the code using the DAO's will have to catch them. Now, if the implementation is changed so that it reads the data from a Web service instead of from a database, then the DAO methods will have to throw both `RemoteException` and `SQLException`. Further, if the DAO is modified to also read data from a file, it will need to throw `IOException` as well. That is three different exceptions, each bound to their own DAO implementation.

Note - A DAO pattern uses abstraction (an interface) to allow implementation substitution.

To avoid this, the DAO interface methods can throw a single `DAOException`. In each implementation of the DAO interface, that is, database, file, or Web service, the user can catch the specific exceptions (`SQLException`, `IOException`, or `RemoteException`), wrap it in a `DAOException`, and throw the `DAOException`. Then, the code using the DAO interface will only have to deal with the `DAOException`. It does not need to know anything about what data access technology was used in the various implementations.

Code Snippet 12 explains the use of wrapper exceptions.

Code Snippet 12:

```
// creating a user-defined exception class
class CalculatorException extends Exception{ // line 1

    public CalculatorException()
    {
    }
}
```

```
// constructor with Throwables object as parameter  
public CalculatorException(Throwable cause) {  
    super(cause);  
}  
  
// constructor with a message string and Throwables object as parameter  
public CalculatorException(String message, Throwable cause) {  
    super(message, cause);  
}  
  
// creating the Calculator class  
class Calculator { // line 2  
  
    // method to divide two numbers  
    public void divide(int a, int b) throws CalculatorException // line 3  
    {  
        // try-catch block  
        try{  
            int result=a/b; // performing division  
            System.out.println("Result is "+result);  
        }  
        catch(ArithmaticException ex)  
        {  
            // throwing the wrapper exception - line 4  
            throw new CalculatorException("Denominator cannot be zero", ex);  
        }  
    }  
}  
  
// creating the TestCalculator class  
public class TestCalculator {  
    public static void main(String[] args) {
```

```

try{
    // creating object of Calculator class
    Calculator objCalc = new Calculator();

    // invoking the divide method
    objCalc.divide(10, 0);

} catch (CalculatorException ex) {
    // getting the cause from the wrapper
    Throwable t = ex.getCause(); // line 5

    // printing the message and the cause
    System.out.println("Error: " + ex.getMessage()); // line 6
    System.out.println("Cause: " + t); // line 7
}
}
}
}

```

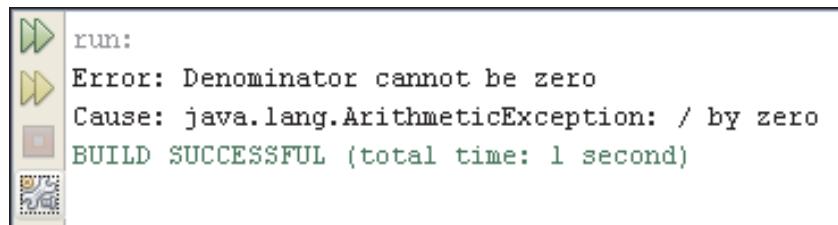
In the code, a user-defined exception class named **CalculatorException** is created. The class has two overloaded constructors that take the `Throwable` object and a message string as parameters. Next, on line 2, another class named **Calculator** is created. It consists of the `divide()` method that throws the **CalculatorException**.

Within the method, a try-catch block is provided that catches the actual exception, that is, `ArithmaticException` when `b` is set to 0. However, within the catch block, the `ArithmaticException` object is wrapped with a custom message string into the **CalculatorException** object on line 4 and then the **CalculatorException** object is thrown.

Next, the **TestCalculator** class is created with the `main()` method. Within `main()`, the `divide()` method is invoked inside the `try` block. In the `catch` block, the **CalculatorException** is handled. However, the actual cause of the **CalculatorException** was `ArithmaticException`. So, to retrieve the cause, the `getCause()` method is used on line 5 and the cause is retrieved in the `Throwable` object.

Lastly, both the message and the cause are printed on lines 6 and 7.

Figure 1.3 shows the output of the code.



```
run:
Error: Denominator cannot be zero
Cause: java.lang.ArithmaticException: / by zero
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 1.3: Wrapper Exception

The output shows the custom message string as well as the actual cause of the exception, that is, `ArithmaticException`. Similarly, if the `divide()` method is liable to throw any other exception, one can catch it in the appropriate `catch` block within the method and then re-throw it after wrapping it in the `CalculatorException` object. Thus, multiple exceptions can be thrown from a method by wrapping it in a single wrapper exception and handled in a single exception handler block further up in the call stack.

1.7 Assertions

An assertion is a statement in Java that allows the programmer to test his/her assumptions about the program. For example, if a method is written that calculates the speed of a particle, one can assert that the calculated speed is less than the speed of light.

Each assertion is composed of a boolean expression that is believed to be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms the assumptions about the behaviour of the program. This helps to increase the programmer's confidence that the code is free of errors.

Writing assertions while programming is one of the most effective and quickest ways to detect and correct errors in the code. Additionally, assertions also help to document the inner workings of a program, thereby, enhancing maintainability. The syntax of assertion statement has the following two forms:

Syntax:

`assert <boolean_expression>;`

where,

`<boolean_expression>`: is a Boolean expression.

When the system runs the assertion, it evaluates the Boolean expression. If it is false, system throws an `AssertionError` without any detailed message.

The second form of the assertion statement is:

`assert <boolean_expression> : <detail_expression> ;`

where,

`<boolean_expression>`: is a Boolean expression.

`<detail_expression>`: is an expression that has a value. It cannot be an invocation of a method that is declared void.

This version of the assert statement is used to provide a detailed message for the `AssertionError`. The system will pass the value of `detail_expression` to the appropriate `AssertionError` constructor. The constructor uses the string representation of the value as the error's detail message.

The purpose of using the detail message is to retrieve and communicate the details of the assertion failure. The message should be able to help the programmer to diagnose and fix the error that led the assertion to fail. Note that the detail message is not intended for the end user error display. Hence, it is generally not required to make these messages understandable in isolation or to internationalize them. The detail message is meant to be interpreted with respect to a full stack trace along with the source code containing the failed assertion.

Similar to all uncaught exceptions, assertion failures are usually labelled in the stack trace with the file and line number from which they were thrown. The second form of the assertion statement should be used instead of the first only if the program contains some additional information that might help diagnose the failure. For example, if the `boolean_expression` involves a relationship between two variables, `a` and `b`, the second form should be used. In such a case, the `detail_expression` can be set to would be "`a: " + a + ", b: " + b`".

In some cases the `boolean_expression` may be expensive to evaluate. For example, a method is written to find the minimum number in an unsorted list of numbers. An assertion is added to verify that the selected element is indeed the minimum. In this case, the work done by the assertion will be equally as expensive as the work done by the method itself.

To ensure that assertions do not become a performance liability in deployed applications, assertions can be enabled or disabled when the program is started. Assertions are disabled by default. Disabling assertions removes their performance related issues entirely. Once disabled, they become empty statements in the code semantics.

Thus, assertions can be added to code to ensure that the application executes as expected. Using assertions, one can test for failing of various conditions. In case the condition fails, one can terminate the application and display the debugging information. However, assertions should not be used if the check to be performed must always be executed at runtime because assertions may be disabled. In such cases, it is advisable to use logic instead of assertions.

Assertion checking is disabled by default. Assertions can be enabled at command line by using the following command:

```
java -ea <class-name>
```

or

```
java -enableassertions <class-name>
```

To enable assertions in NetBeans IDE, perform the following steps:

1. Right-click the project in the **Projects** tab. A pop-up menu appears.
2. Select **Properties**. The **Project Properties** dialog box is displayed.

3. Select **Run** from the **Categories** pane. The runtime settings pane is displayed on the right as shown in figure 1.4.

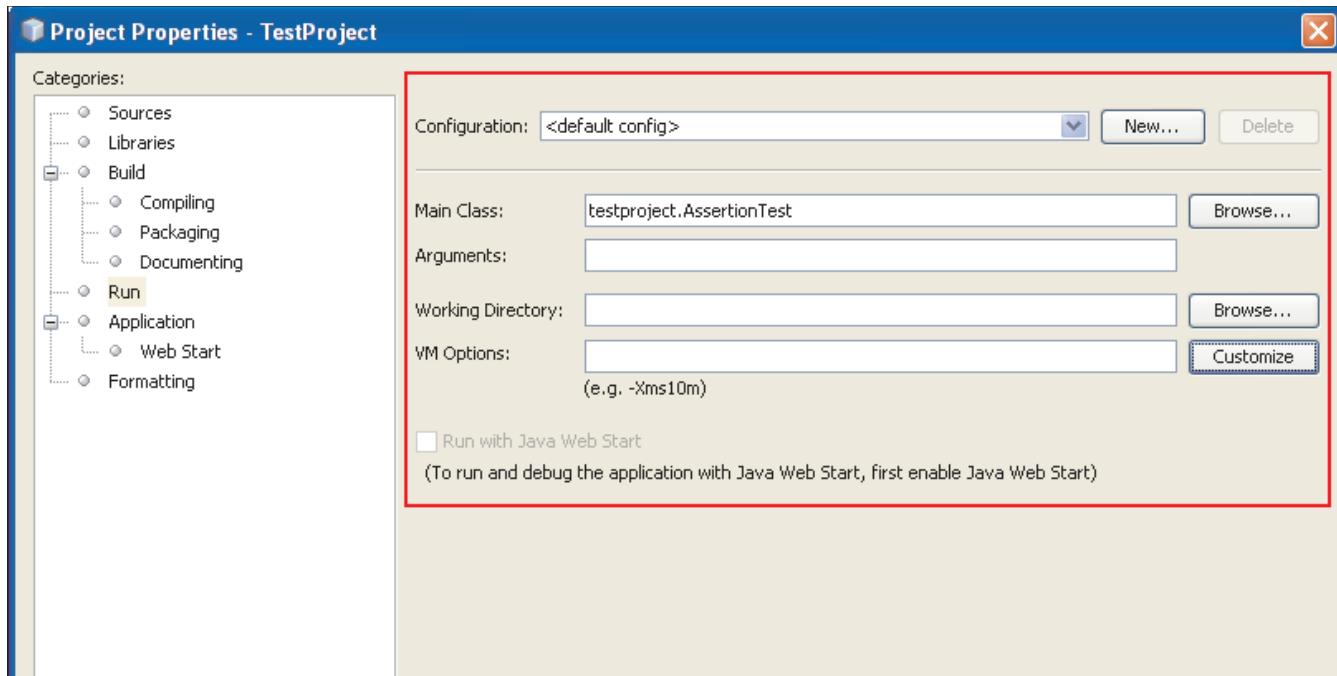


Figure 1.4: Project Properties Dialog Box

4. Click the **Customize** button. The **Project Properties** dialog box is displayed.
 5. Scroll down and select the **ea** checkbox as shown in figure 1.5.

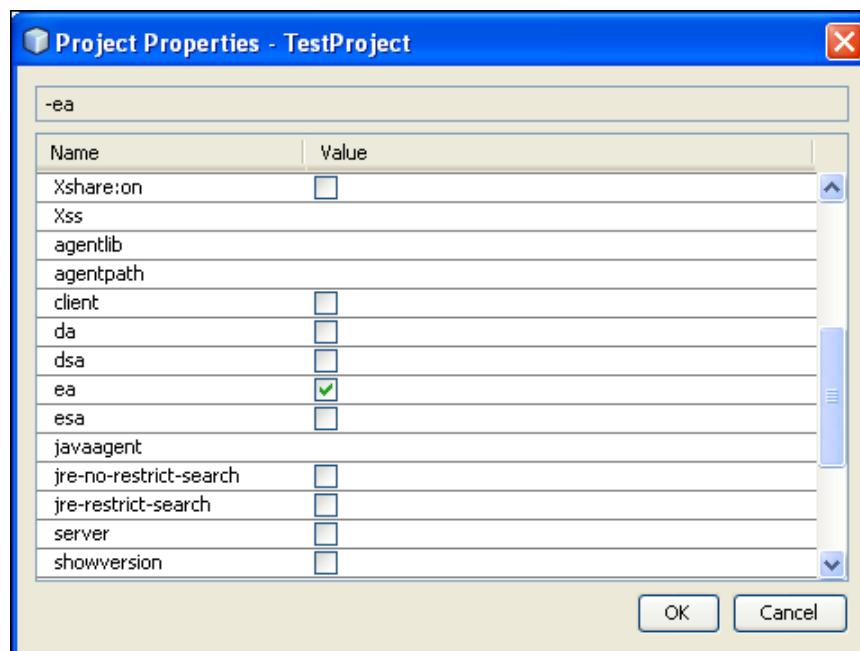


Figure 1.5: -ea Runtime Option Enabled

6. Click **OK**. The **-ea** option is set in the **VM Options** text box as shown in figure 1.6.

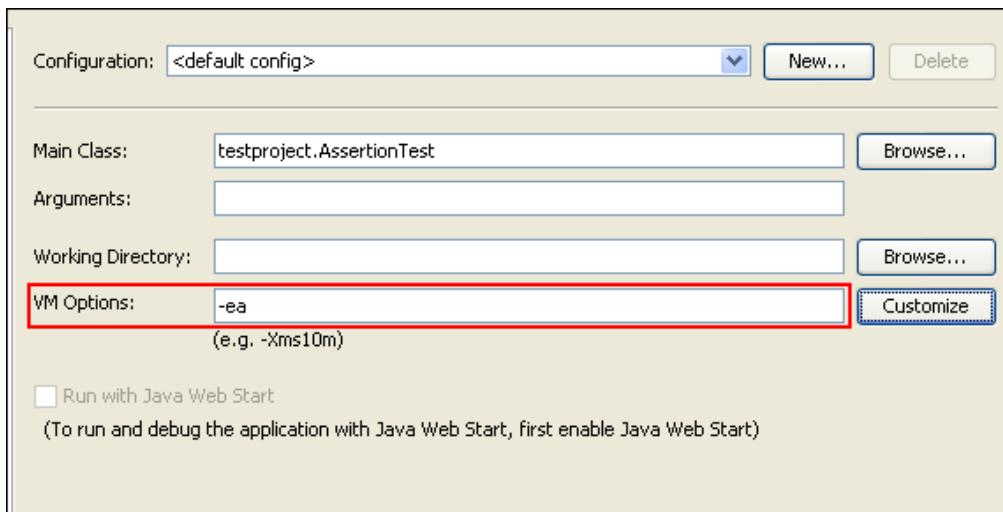


Figure 1.6: -ea Option Applied at Runtime

7. Click **OK**. This will close the **Project Properties** dialog box. Now, assertions will be enabled at runtime in NetBeans IDE.

One can use assertions to document and verify the assumptions and internal logic of a single method in the following ways:

- Internal Invariants
- Control flow Invariants
- Precondition and Postcondition
- Class Invariants

1.7.1 Internal Invariants

Earlier, where assertions were not available, many programmers used comments to indicate their assumptions concerning a program's behavior. For example, one might have written a comment as shown in Code Snippet 13 to explain the assumption about an `else` clause in an `if...else` statement.

Code Snippet 13:

```
public class AssertionTest {

    public static void main(String[] args) {
        int a = 0;
```

```

if(a>0)
    // do this if a is greater than zero
else{
    // do that, unless a is negative
}
}
}

```

The code states that the `else` statement should be executed only if `a` is equal to zero but not if `a>0`. However, at runtime, this can be missed out since no error will be raised even if a negative number is specified at runtime. For such invariants, one can use assertion as shown in Code Snippet 14.

Code Snippet 14:

```

public class AssertionTest{

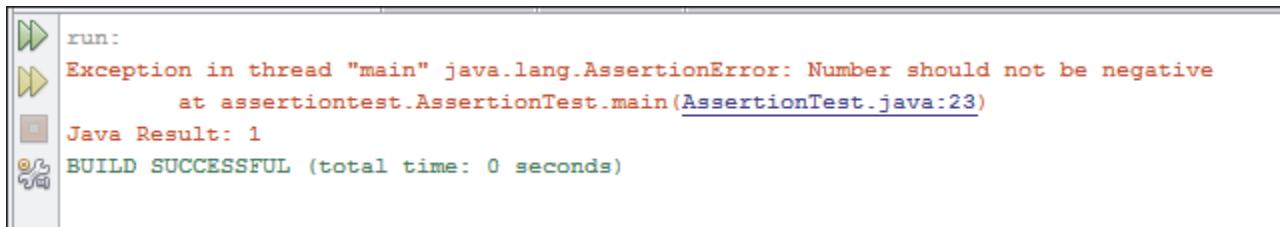
public static void main(String[] args) {
    int a=-1;

    if(a>0)
        System.out.println("Greater than zero");
    else{
        assert a==0:"Number should not be negative";
        System.out.println("Number is zero");
    }
}
}

```

In the code, the value of `a` has been set to `-1`. In the `else` block, an assert statement is provided which checks if `a` is equal to zero. If not, the detail message will be displayed to the user and the application will terminate.

Figure 1.7 shows the output of the code when assertions are enabled.



```
run:
Exception in thread "main" java.lang.AssertionError: Number should not be negative
    at assertiontest.AssertionTest.main(AssertionTest.java:23)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 1.7: AssertionError for Internal Invariants

Figure 1.7 shows the `AssertionError` along with the detail message specified by the programmer.

1.7.2 Control Flow Invariants

Assertions can also be applied to control flow invariants such as a `switch` statement that has no `default` case. The absence of a `default` case is indicative of the belief that one of the cases will always be executed. The assumption that a particular variable will surely have any one of a small set of values is an invariant that needs to be checked with an assertion. For example, suppose a `switch` statement appears in a program that checks days of a week as shown in Code Snippet 15.

Code Snippet 15:

```
public class ControlFlowTest {

    public static void main(String[] args) {

        String day=args[0];

        switch (day) {
            case "Sun":
                // do this
                break;

            case "Mon":
                // do this
                break;

            case "Tue":
                // do this
        }
    }
}
```

```

        break;

    case "Wed":
        // do this
        break;

    case "Thu":
        // do this
        break;

    case "Fri":
        // do this
        break;

    case "Sat":
        // do this
        break;
    }
}
}
}

```

The code probably indicates an assumption that the `day` variable will have only one of the seven values. To test this assumption, one can add the following default case:

```

default:
    assert false : day + " is incorrect";

```

If the `day` variable takes on any other value and assertions are enabled, the assertion will fail and an `AssertionError` will be thrown as shown in figure 1.8.



Figure 1.8: `AssertionError` in Control Flow Variants

1.7.3 PreCondition, PostCondition, and Class Invariants

While the `assert` construct is not a complete help in itself, it can help support an informal design-by-contract style of programming. One can use assertions for:

- **Preconditions:** what must be true when a method is invoked?

- **Postconditions:** what must be true after a method executes successfully?
- **Class invariants:** what must be true about each instance of a class?
- **Preconditions**

By convention, preconditions on `public` methods are enforced by making explicit condition checks that throw particular, specified exceptions. For example, consider the code given in Code Snippet 16.

Code Snippet 16:

```
// method to set the refresh rate and
//throw IllegalArgumentException if rate <=0 or rate > MAX_RATE

public void setRate(int rate) {
    // Apply the specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);
    setInterval(1000/rate);
}
```

In the code, the method `setRate()` takes the refresh rate as a parameter and checks if the value is less than zero and greater than `MAX_RATE`. If not, it throws an `IllegalArgumentException` and the rest of the code is not executed.

In this kind of setup, there is no use of adding an assert statement. Therefore, do not use assertions to check the parameters of a `public` method. An assert statement is inappropriate in this case because the method itself guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, the assert construct cannot throw an exception of a specific type. It can throw only an `AssertionError`.

However, one can use an assertion to test a precondition of a non-public method that is believed to be true no matter what a user does with the class. For example, an assertion is appropriate in the helper method `setInterval(int interval)` that is invoked by the `setRate()` method as shown in Code Snippet 17.

Code Snippet 17:

```
// Method to set the refresh interval (in milliseconds) which
// must correspond to a legal frame rate
```

```
private void setInterval(int interval) {
    // Verify the adherence to precondition in the non-public method
    assert interval > 0 && interval <= 1000/MAX_RATE : interval;
    // Set the refresh interval
    System.out.println("Interval is set to:" + interval);
}
```

In the code, the assertion will fail if **MAX RATE** is greater than 1000 and the client selects the value of **rate** greater than 1000. This would surely indicate an error in the library.

→ Postconditions

Postconditions can be checked with assertions in both public and non-public methods. For example, the public method **pop()** in Code Snippet 18 uses an assert statement to check a postcondition.

Code Snippet 18:

```
public class PostconditionTest{
    ArrayList values = new ArrayList();

    public PostconditionTest() {
        values.add("one");
        values.add("two");
        values.add("three");
        values.add("four");
    }

    public Object pop() {
        int size = values.size(); // line 1
        if (size == 0)
            throw new RuntimeException("List is empty!!");

        Object result = values.remove(0);
    }
}
```

```
// verify the postcondition
assert(values.size() == size - 1); // line 2

return result;
}

}
```

In the code, a `pop()` method is used to pop elements from a list. First, the size of the list is determined on line 1 with the help of the `size()` method. Next, if the size equals zero, the `RuntimeException` is thrown. Further, the code to pop the element is provided. Now, before popping the next element, a check is made using assertion on line 2. Here, if the `size()` method does not return a value equal to `size-1`, the assertion fails and throws an `AssertionError`.

→ Class Invariants

A class invariant is a type of internal invariant that is applied to every instance of a class. It is applicable at all times except when the instance is transiting from one consistent state to another. A class invariant can be used to specify the relationships among multiple attributes. Also, it should be true before and after any method completes. For example, in case of a program that implements a balanced tree data structure, a class invariant might be that the tree is balanced and properly ordered.

The assertion mechanism does not adopt any specific style for checking invariants. However, it is sometimes convenient and advisable to combine the expressions that verify the required constraints into a single internal method that can be called by assertions. With respect to the balanced tree example, it would be better to implement a `private` method that checked that the tree was indeed balanced as per the rules of the data structure as shown in Code Snippet 19.

Code Snippet 19:

```
// Returns true if this tree is properly balanced
private boolean balanced() {
    // code to check if the tree is balanced
}
```

Since this method is used to check a constraint that should be true before and after any method completes, each `public` method and constructor should contain the line, `assert balanced();` immediately prior to its return.

It is usually not required to place similar checks at the head of each `public` method unless the data structure is implemented by native methods. A failure of the assertion at the head of such a method would indicate that such memory corruption had occurred. Similarly, it is advisable to include class invariant checks at the heads of methods in classes whose state is likely to be modified by other classes.

1.7.4 Inappropriate Use of Assertions

Assertions can be disabled at runtime. Hence,

- Do not use assertion to check the parameters of a public method.

This is because argument checking is typically part of the published specifications or contract of a method. These specifications must be followed whether assertions are enabled or disabled. Another problem with using assertions for parameter checking is that, as per standards, incorrect arguments should result in an appropriate runtime exception such as `IllegalArgumentException`, `NullPointerException`, or `IndexOutOfBoundsException`. An assertion failure will not throw the appropriate exception.

- Do not use methods that can cause side effects in the assertion check.
- Do not use assertions to do any task that the application requires for correct operation.

This is because assertions may be disabled. Hence, programs must not assume that the Boolean expression contained in an assertion will be evaluated especially if violating the rule can lead to dire consequences. For example, suppose a code is required to remove all of the null elements from a list, `cities`, and knew that the list contained one or more nulls. In this case, it would be incorrect to do this:

```
// broken code - as action is contained in assertion
assert cities.remove(null);
```

The program would work fine when assertions are enabled, but would fail when they were disabled, as it would not remove the null elements from the list. The correct way is to perform the action before the assertion and then assert that the action succeeded as follows:

```
// fixed code - action precedes assertion
Boolean removedNull = cities.remove(null);
assert removedNull;
```

As a rule, the expressions written in assertions should be free of side effects. That is, evaluating the expression should not affect any state of the program that is visible after the evaluation is complete. One exception to this rule is that assertions can modify a state that is used only from within other assertions.

1.8 Check Your Progress

1. _____ is a try statement that declares one or more resources.

(A)	try
(B)	try-catch
(C)	try-with-resources
(D)	try-catch-finally

(A)	A	(C)	C
(B)	B, C	(D)	B, D

2. Which of the following statements about assertions are true?

(A)	Assertions should be used to check the parameters of a public method
(B)	Do not use assertions to do any task that the application requires for correct operation.
(C)	An assertion is a statement in the Java that allows the programmer to test his/her assumptions about the program.
(D)	Assertion checking is enabled by default.

(A)	A	(C)	A, C
(B)	B, C	(D)	B, D

3. Match the following keywords with the corresponding descriptions with respect to exception handling.

Access Specifier		Description	
(A)	throw	1.	Used to test an assumption about a code in a program.
(B)	throws	2.	Used to throw the exception declared by a method.
(C)	assert	3.	Used to handle the appropriate exception raised in the <code>try</code> block.
(D)	catch	4.	Used to declare the exceptions that a method is liable to throw.

(A)	A-4, B-1, C-2, D-3	(C)	A-3, B-2, C-1, D-4
(B)	A-1, B-2, C-3, D-4	(D)	A-2, B-4, C-1, D-3

4. Which of the following is the correct syntax to create a user-defined exception class?

(A)	public class <ExceptionName> implements Exception {}
(B)	public static final class <ExceptionName> extends Exception {}
(C)	public class <ExceptionName> extends Throwable {}
(D)	public class <ExceptionName> extends Exception {}

(A)	A	(C)	C
(B)	B	(D)	D

5. Consider the following code:

```
public void testException(){ // line 1
    if(condition==true)
        System.out.println("True");
    else
        throw new MyException();
}
```

The code is giving the warning '**unreported exception project.MyException. Must be caught or declared to be thrown**'. What change must be made in line 1 to resolve the issue?

(A)	public void testException() throw MyException{
(B)	public void testException() throws MyException() {
(C)	public void testException() throws MyException{
(D)	public void throw MyException testException() {

(A)	A	(C)	C
(B)	B	(D)	D

6. Consider the following code:

```
public void addEmployee{  
    ArrayList empNames = new ArrayList();  
    try{  
        empNames.remove(0);  
    }  
    catch(IndexOutOfBoundsException|RuntimeException ex){  
        throw ex;  
    }  
}
```

The code consists of a method to remove employee names from an `ArrayList`. Which of the following programming concept(s) have been used in the code?

(A)	try block with single catch block
(B)	try block with multiple catch blocks
(C)	Single catch block for multiple exceptions
(D)	Assertion

(A)	A	(C)	A, C
(B)	B	(D)	C, D

1.8.1 Answers

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

```
g package main
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello " + name);
    }
}
```

Summary

- An exception is an event occurring during program execution that leads to disruption of the normal flow of the program's instructions.
- The finally block is executed even if an exception occurs in the try block.
- The throws clause is written after the method name and argument list and before the opening brace of the method.
- All methods use the throw statement to throw an exception. The throw statement takes a single argument which is a throwable object.
- To create a multiple exception catch block, the types of exceptions that catch block can handle are specified separated by a vertical bar (|).
- The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it.
- To create a user-defined exception class, the class must inherit from the Exception class.
- Exception wrapping is catching an exception, wrapping it in a different exception, and throwing the wrapper exception.
- An assertion is a statement in the Java that allows the programmer to test his/her assumptions about the program.
- Assertions should not be used to check the parameters of a public method.



To add on to your knowledge of the subject,
visit the **REFERENCES** page



Session 2

java.lang Package

Welcome to the Session, **java.lang Package**.

This session explains the use of the java.lang Package and its classes. It also explains the use of Strings, regular expressions, pattern, and matcher. Further, this session explains the use of String literal and Character classes. Lastly, the session explains the use of quantifiers, capturing groups, and boundary matchers.

In this Session, you will learn to:

- Describe the java.lang package
- Explain the various classes of java.lang package
- Explain how to use and manipulate Strings
- Explain regular expressions, pattern, and matcher
- Explain String literal and Character classes
- Explain the use of quantifiers, capturing groups, and boundary matchers



2.1 Introduction

While writing programs in Java, it is often required to perform certain tasks on the data specified by the user. The data could be in any format such as strings, numbers, characters, and so on. To manipulate such data, special classes and methods are required. They are provided by a special package in Java called the `java.lang` package.

2.2 Overview of the `java.lang` Package

The `java.lang` package provides classes that are fundamental for the creation of a Java program. This includes the root classes that form the class hierarchy, basic exceptions, types tied to the language definition, threading, math functions, security functions, and also, information on the underlying native system. The most important classes are as follows:

- `Object`: which is the root of the class hierarchy.
- `Class`: instances of this class represent classes at run time.

Classes in `java.lang` are automatically imported into every source file. Hence, an explicit `import` statement is not required for using this package.

The other important classes and interfaces in `java.lang` package are as follows:

- `Enum`: It is the base class for all enumeration classes.
- `Throwable`: It is the base class of the exception class hierarchy.
- `Error`, `Exception`, and `RuntimeException`: These are base classes for each exception type.
- Exception classes thrown for language-level and other common exceptions.
- `Thread`: It is the class that allows manipulation of threads.
- `String`: It is the class used for creating and manipulating strings and string literals.
- `StringBuffer` and `StringBuilder`: These classes are used for performing string manipulation.
- `Comparable`: This is an interface that allows generic comparison and ordering of objects.
- `Iterable`: This is an interface that allows generic iteration using the enhanced for loop.
- `Process`, `ClassLoader`, `Runtime`, `System`, and `SecurityManager`: These classes provide system operations for managing the creation of external processes, dynamic loading of classes, making environment inquiries such as the time of day, and enforcing of security policies.

- Math: This class provides basic math functions such as square root, sine, cosine, and so on.
- Wrapper classes that encapsulate primitive types as objects.

2.2.1 Working with Garbage Collection

Memory management is the process of identifying the objects in memory that are no longer required and de-allocating the memory used by such objects so that it can be used for subsequent allocations. In some programming languages, memory management is the programmer's responsibility. However, the complexity of memory management leads to several common errors that can cause erroneous or unexpected program behavior and crashes. This leads to a lot of time spent debugging and trying to correct such errors.

One major problem that can occur in a program with explicit memory management is dangling references. It may happen that the space allotted to an object is de-allocated but some other object still has a reference to it. If the object (now dangling) with that reference tries to access the original object, the result would be unpredictable if the space has been reallocated to a new object.

Another common problem with explicit memory management is memory leaks. Memory leak occurs when memory is allocated to an object that is no longer referenced, but is not released. For example, if while intending to free the space occupied by a linked list, the programmer simply de-allocates the first element of the list. In this case, the remaining list elements are no longer referenced. Also, they go out of the program's reach and can neither be used nor recovered. If such leaks continue to occur, they may keep consuming memory until all available memory is exhausted.

An alternate approach to explicit memory management adopted by many object-oriented languages is automatic memory management by a program called a garbage collector. Garbage collection helps to avoid the problem of dangling references because an object that is still referenced somewhere will never be garbage collected and so will not be considered free. Garbage collection also solves the problem of memory leak problem because it automatically frees all memory that is no longer referenced.

Thus, a garbage collector is responsible for:

- allocating memory
- ensuring that any object that has references should remain in memory
- reclaim memory occupied by objects that are no longer referenced

Objects that have references are considered to be live whereas objects that are no longer referenced are considered dead. Such objects are termed as garbage. The process of identifying and reclaiming the space occupied by these objects is known as garbage collection.

Garbage collection may solve many, but not all, memory allocation problems. It is also a complex process that takes time and resources of its own. The exact algorithm that is used for organizing memory and allocating/de-allocating memory is handled by the garbage collector and hidden from the programmer. Space is mostly allocated from a large pool of memory referred to as the heap.

An object becomes eligible for garbage collection if it is not reachable from any live threads or any static references. In other words, it becomes eligible for garbage collection if all its references are null. It should be noted that, cyclic dependencies are not considered as reference. Hence, if Object A references object B and object B references Object A, but they do not have any other live reference. In this case, both Objects A and B will be eligible for garbage collection.

Thus, an object becomes eligible for garbage collection in the following cases:

1. All references of that object are explicitly set to null, for example, `object = null`
2. An object is created inside a block and its reference goes out of scope once control exits that block.
3. Parent object is set to null. If an object holds reference to another object and when the container object's reference is set to null, the child object automatically becomes eligible for garbage collection.

The time for garbage collection is decided by the garbage collector. Usually, the entire heap or a part of it is collected either when it gets filled or when it reaches a threshold percentage of occupancy.

Some of the features of a garbage collector are as follows:

- Must be both safe and comprehensive.
- Should ensure that live data is never erroneously freed and garbage does not remain unclaimed for more than a small number of collection cycles.
- Should operate efficiently, without taking long pauses during which the application is not running. However, there are often trade-offs between time, space, and frequency. For example, for a small heap size, collection will be fast but the heap will fill up more quickly, thus requiring more frequent collections. On the other hand, a large heap will take longer to fill up and thus collections will be less frequent, but they may take longer.
- Should limit fragmentation of memory when the memory for garbage objects is freed. This is because if the freed memory is in small chunks, it might not be enough to be used for allocation of a large object.
- Should handle the scalability issue also multithreaded applications on multiprocessor systems.

A number of parameters must be studied while designing or selecting a garbage collection algorithm:

→ **Serial versus Parallel**

With serial collection, only one thing happens at a time. For example, even when multiple CPUs are available, only one will be utilized to perform the collection. Whereas, in parallel collection, the task of garbage collection is divided into subparts that are executed simultaneously on different CPUs. The simultaneous collection is speedy but leads to additional complexity and potential fragmentation.

→ Concurrent versus Stop-the-world

In the stop-the-world garbage collection approach, during garbage collection, application execution is completely suspended. Whereas, in concurrent approach, one or more garbage collection tasks can be executed concurrently, that is, simultaneously, with the execution of the application. However, this incurs some overhead on the concurrent collectors and affects performance due to larger heap size requirement.

→ Compacting versus Non-compacting versus Copying

In the compacting approach, once a garbage collector has determined which objects in memory are garbage, it can compact the memory by moving all the live objects together and completely reclaiming the memory of the unreferenced objects. After compaction, it becomes easier and quicker to allocate a new object at the first free location. This can be done by using a simple pointer to keep track of the next location available for object allocation.

In contrast, a non-compacting collector releases the space utilized by garbage objects in-place. That is, it does not move all live objects together to create a large reclaimed region like a compacting collector does. The benefit is faster completion of garbage collection, but the drawback is potential fragmentation.

Generally, it is more expensive to allocate from a heap with in-place de-allocation than from a compacted heap. This is because, it may be necessary to search the entire heap for a contiguous area of memory large enough to accommodate the new object.

In the copying approach, the collector copies or evacuates live objects to a different memory area. The advantage is that the source area becomes empty and can be used for faster and easier subsequent allocations. However, the drawback is that additional time is required for copying as well as the extra space that may be required to move the objects.

Following metrics can be utilized to evaluate the performance of a garbage collector:

- **Throughput:** It is the percentage of total time not spent in garbage collection, considering a longer time period.
- **Garbage collection overhead:** It is the inverse of throughput. That is, the percentage of total time spent in garbage collection.
- **Pause time:** It is the amount of time during which application execution is suspended while garbage collection is occurring.
- **Frequency of collection:** It is a measure of how often collection occurs in relation to application execution.
- **Footprint:** It is a measure of size, such as heap size.

- **Promptness:** It is the time span between the time an object becomes garbage and the time when its memory becomes available.

An important method for garbage collection is the `finalize()` method.

finalize() method

The `finalize()` method is called by the garbage collector on an object when it is identified to have no more references pointing to it. A subclass overrides the `finalize()` method for disposing the system resources or to perform other cleanup.

The syntax of `finalize()` method is as follows:

Syntax:

```
protected void finalize() throws Throwable
```

The usual purpose of `finalize()` method is to perform cleanup actions before the object is irrevocably discarded. However, the `finalize()` method may take any action, including making this object available again to other threads. For example, the `finalize()` method for an object representing an input/output connection might perform explicit I/O transactions for breaking the connection before the object is permanently discarded. The `finalize()` method of class `Object` does not perform any special action; it simply returns normally. However, subclasses of `Object` may override this definition as per the requirement.

After the `finalize()` method is invoked for an object, no further action is taken until the JVM determines that there are no live threads trying to access the same object, after which, the object may be discarded. The `finalize()` method is never invoked more than once by the JVM for any given object. Also, any exception thrown by the `finalize()` method leads to suspension of the finalization of this object. The `finalize()` method throws the `Throwable` exception.

At times, an object might need to perform some task when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font. In this case, one might want to make sure these resources are released before the object is destroyed. To handle one can define the specific actions in the `finalize()` method. When the JVM calls that method while recycling an object of that class, actions specified in the `finalize()` methods will be performed before the object is destroyed.

However, it is important to note that `finalize()` method is only invoked just prior to garbage collection but not when an object goes out-of-scope. This means that when exactly the `finalize()` method will be executed is not known. Therefore, it is advisable to provide other means of releasing system resources used by the object instead of depending on the `finalize()` method.

To understand automatic garbage collection, consider the example given in Code Snippet 1.

Code Snippet 1:

```
class TestGC{  
    int num1;  
    int num2;  
  
    public void setNum(int num1,int num2) {  
        this.num1=num1;  
        this.num2=num2;  
    }  
    public void showNum() {  
        System.out.println("Value of num1 is " + num1);  
        System.out.println("Value of num2 is " + num2);  
    }  
    public static void main(String args[]) {  
        TestGC obj1 = new TestGC();  
        TestGC obj2 = new TestGC();  
        obj1.setNum(2,3);  
        obj2.setNum(4,5);  
        obj1.showNum();  
        obj2.showNum();  
        //TestGC obj3; // line 1  
        //obj3=obj2; // line 2  
        //obj3.showNum(); // line 3  
        //obj2=null; // line 4  
        //obj3.showNum(); // line 5  
        //obj3=null; // line 6  
        //obj3.showNum(); // line 7  
    }  
}
```

The code shows a class `TestGC` with two variables that are initialized in the `setNum()` method and displayed using the `showNum()` method. Next, two objects `obj1` and `obj2` of the `TestGC` class have been created. Now, to understand garbage collection, execute the code. Figure 2.1 shows the in-memory representation of the objects created on execution.

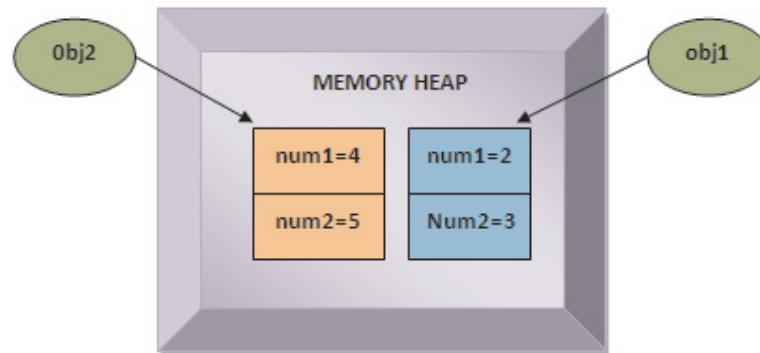


Figure 2.1: In-memory Representation of Objects

Now, if lines 1, 2, and 3 are uncommented and the code is re-run, two reference variables will point to the same object as shown in figure 2.2.

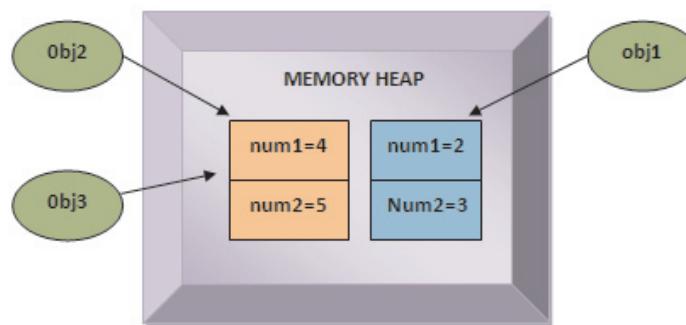


Figure 2.2: Two Reference Variables Pointing to an Object

Next, when lines 4 and 5 are uncommented and code is re-run, `obj2` becomes null, but `obj3` still points to the object as shown in figure 2.3. Therefore, the object is still not eligible for garbage collection.

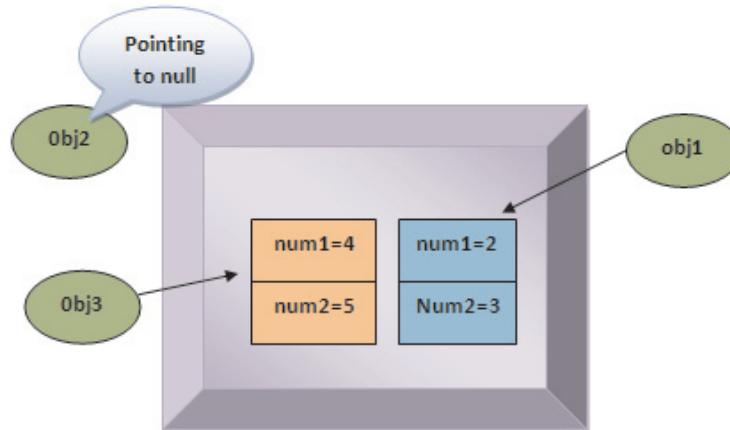


Figure 2.3: Reference Pointing to null

Now, if lines 6 and 7 are uncommented and the code is re-run, `obj3` also becomes null. At this point, there are no references pointing to the object and hence, it becomes eligible for garbage collection. It will be removed from memory by the garbage collector and cannot be retrieved again. This is shown in figure 2.4.

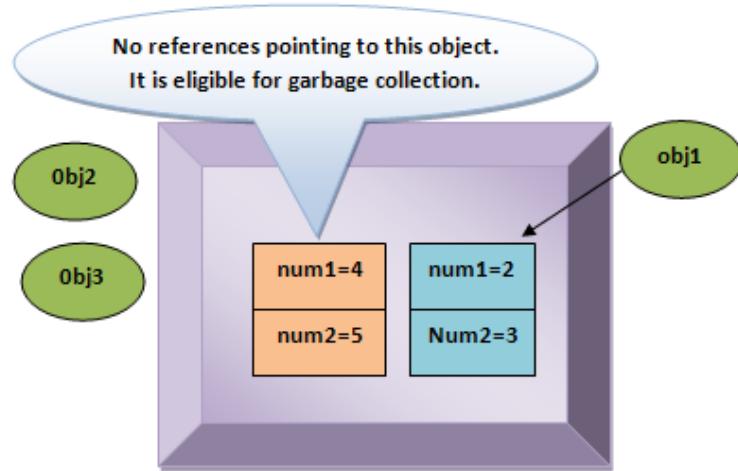


Figure 2.4: Object Ready for Garbage Collection

Thus, some important points about garbage collection are as follows:

1. To make an object eligible for garbage collection, its reference variable should be set to null.
2. It should be noted that, primitive types are not objects. Hence, they cannot be assigned null. That is, `int x = null` is wrong.

2.3 Classes of `java.lang` Package

The various classes provided under `java.lang` package helps a developer to accomplish common tasks quickly and efficiently without having to write unnecessary logic. Some important classes of the package are discussed in this section.

2.3.1 Wrapper Classes

At times, it is necessary to represent a primitive type as if it were an object. The wrapper classes such as `Boolean`, `Character`, `Integer`, `Float`, `Long`, and `Double` serve this purpose. A wrapper type of each primitive type is available. An object of type `Integer`, for example, contains a field whose type is `int`. It represents that value in such a way that a reference to it can be stored in a variable of reference type. The wrapper classes also provide a number of methods for processing variables of specified data type to another type. For example, to convert a `String` to an `Integer`, one can write as follows:

Here, the `parseInteger()` method is used to convert the string '1234' into an integer variable. Also, the direct assignment of `Integer` type to primitive `int` type is also allowed and the conversion happens implicitly. They also support such standard methods as `equals()` and `hashCode()`. The `Void` class cannot be instantiated and holds a reference to a `Class` object representing the primitive type `void`.

2.3.2 Math Class

The Math class contains methods for performing basic mathematical/numeric operations such as square root, trigonometric functions, elementary exponential, logarithm, and so on.

By default many of the Math methods simply call the equivalent method of the StrictMath class for their implementation. Table 2.1 lists some of the commonly used methods of the Math class.

Method	Description
static double abs(double a)	The method returns the absolute value of a double value.
static float abs(float a)	The method returns the absolute value of a float value.
static int abs(int a)	The method returns the absolute value of an int value.
static long abs(long a)	The method returns the absolute value of a long value.
static double ceil(double a)	The method returns the smallest double value that is greater than or equal to the argument.
static double cos(double a)	The method returns the trigonometric cosine of an angle.
static double exp(double a)	The method returns Euler's number e raised to the power of a double value.
static double floor(double a)	The method returns the largest double value that is less than or equal to the argument.
static double log(double a)	The method returns the natural logarithm (base e) of a double value.
static double max(double a, double b)	The method returns the greater of two double values.
static float max(float a, float b)	The method returns the greater of two float values.
static int max(int a, int b)	The method returns the greater of two int values.
static long max(long a, long b)	The method returns the greater of two long values.
static double min(double a, double b)	The method returns the smaller of two double values.
static float min(float a, float b)	The method returns the smaller of two float values.
static int min(int a, int b)	The method returns the smaller of two int values.
static long min(long a, long b)	The method returns the smaller of two long values.
static double pow(double a, double b)	The method returns the value of the first argument raised to the power of the second argument.
static double random()	The method returns a positive double value, greater than or equal to 0.0 and less than 1.0.
static long round(double a)	The method returns the closest long to the argument.
static int round(float a)	The method returns the closest int to the argument.

Method	Description
static double sin(double a)	The method returns the trigonometric sine of an angle.
static double sqrt(double a)	The method returns the correctly rounded positive square root of a double value.
static double tan(double a)	The method returns the trigonometric tangent of an angle.

Table 2.1: Methods of the Math Class

Code Snippet 2 shows the use of some of the methods of `Math` class.

Code Snippet 2:

```
// creating a class to use Math class methods

class MathClass {

    int num1; // declaring variables
    int num2;

    // declaring constructors
    public MathClass () { }

    public MathClass (int num1, int num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    // method to use max()
    public void doMax () {
        System.out.println("Maximum is: " + Math.max(num1, num2));
    }

    // method to use min()
    public void doMin () {
        System.out.println("Minimum is: " + Math.min(num1, num2));
    }

    // method to use pow()
}
```

```

public void doPow() {
    System.out.println("Result of power is: " + Math.pow(num1, num2));
}

// method to use random()
public void getRandom() {
    System.out.println("Random generated is: " + Math.random());
}

// method to use sqrt()
public void doSquareRoot() {
    System.out.println("Square Root of " + num1 + " is: " + Math.sqrt(num1));
}
}

public class TestMath {
    public static void main(String[] args) {
        MathClass objMath = new MathClass(4, 5);
        objMath.doMax();
        objMath.doMin();
        objMath.doPow();
        objMath.getRandom();
        objMath.doSquareRoot();
    }
}

```

The code shows the use of the `max()`, `min()`, `pow()`, `random()`, and `sqrt()` methods of the `Math` class. Figure 2.5 shows the output of the code.

```

run:
Maximum is: 5
Minimum is: 4
Result of power is: 1024.0
Random generated is: 0.1786669750639418
Square Root of 4 is: 2.0
BUILD SUCCESSFUL (total time: 2 seconds)

```

Figure 2.5: Result of Math Class Methods

2.3.3 System Class

The `System` class provides several useful class fields and methods. However, it cannot be instantiated. It provides several facilities such as standard input, standard output, and error output streams; a means of loading files and libraries; access to externally defined properties and environment variables; and a utility method for quickly copying a part of an array. Table 2.2 lists some of the commonly used methods of the `System` class.

Method	Description
<code>static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)</code>	The method copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
<code>static long currentTimeMillis()</code>	The method returns the current time in milliseconds.
<code>static void exit(int status)</code>	The method terminates the currently running JVM.
<code>static void gc()</code>	The method runs the garbage collector.
<code>static String getenv(String name)</code>	The method gets the value of the specified environment variable.
<code>static Properties getProperties()</code>	The method determines the current system properties.
<code>static void loadLibrary(String libname)</code>	The method loads the system library specified by the libname argument.
<code>static void setSecurityManager(SecurityManager s)</code>	The method sets the system security.

Table 2.2: Methods of System Class

Code Snippet 3 shows the use of some of the methods of `System` class.

Code Snippet 3:

```
class SystemClass {

    int arr1[] = {1, 3, 2, 4};
    int arr2[] = {6, 7, 8, 0};

    public void getTime()
    {
        System.out.println("Current time in milliseconds is: " + System.
currentTimeMillis());
    }
}
```

```

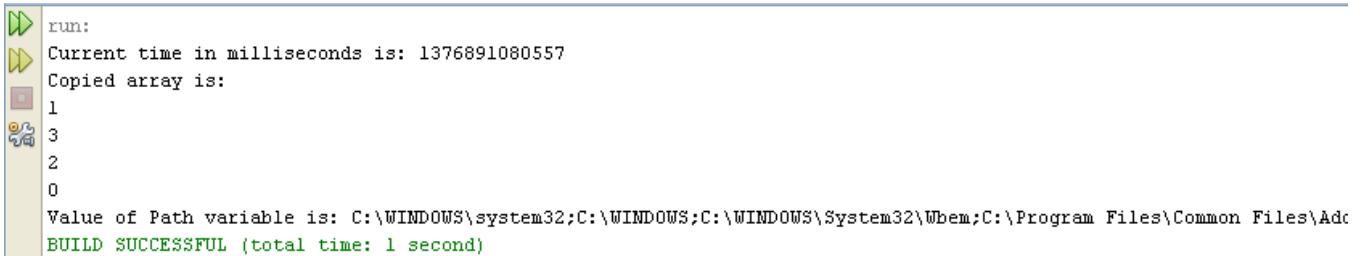
public void copyArray()
{
    System.arraycopy(arr1, 0, arr2, 0, 3);
    System.out.println("Copied array is: ");
    for(int i=0; i<4; i++)
        System.out.println(arr2[i]);
}

public void getPath(String variable)
{
    System.out.println("Value of Path variable is: " + System.getenv(variable));
}
}

public class TestSystem {
    public static void main(String[] args) {
        SystemClass objSys = new SystemClass();
        objSys.getTime();
        objSys.copyArray();
        objSys.getPath("Path");
    }
}

```

The code shows the use of the `currentTimeMillis()`, `arraycopy()`, and `getenv()` methods of the `System` class. Figure 2.6 shows the output of the code.



```

run:
Current time in milliseconds is: 1376891080557
Copied array is:
1
3
2
0
Value of Path variable is: C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\WBem;C:\Program Files\Common Files\Adc
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 2.6: Result of System Class Methods

2.3.4 Object Class

`Object` class is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of the `Object` class. Table 2.3 lists some of the commonly used methods of the `Object` class.

Method	Description
<code>protected Object clone()</code>	The method creates and returns a copy of this object.
<code>boolean equals(Object obj)</code>	The method indicates whether some other object is ‘equal to’ this one.
<code>protected void finalize()</code>	The method called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<code>Class<? extends Object> getClass()</code>	The method returns the runtime class of an object.
<code>int hashCode()</code>	The method returns a hash code value for the object.
<code>void notify()</code>	The method wakes up a single thread that is waiting on this object’s monitor.
<code>void notifyAll()</code>	The method wakes up all threads that are waiting on this object’s monitor.
<code>String toString()</code>	The method returns a string representation of the object.
<code>void wait()</code>	The method causes current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.
<code>void wait(long timeout)</code>	The method causes current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.
<code>void wait(long timeout, int nanos)</code>	The method causes current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Table 2.3: Methods of the Object Class

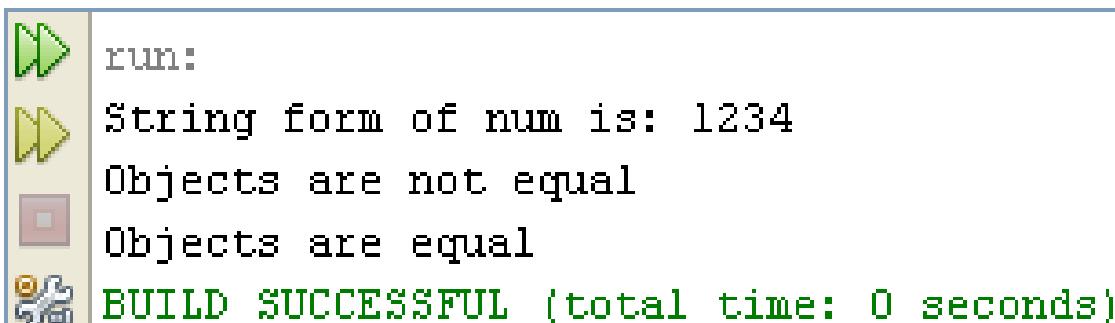
Note - A thread is a thread of execution in a program. The JVM allows an application to have multiple threads of execution running concurrently.

Code Snippet 4 shows the use of some of the methods of `Object` class.

Code Snippet 4:

```
class ObjectClass {  
    Integer num;  
    public ObjectClass () {}  
    public ObjectClass (Integer num) {  
        this.num = num;  
    }  
    // method to use the toString() method  
    public void getStringForm () {  
        System.out.println("String form of num is: " + num.toString());  
    }  
}  
  
public class TestObject {  
    // creating objects of ObjectClass class  
    ObjectClass obj1 = new ObjectClass (1234);  
    ObjectClass obj2 = new ObjectClass (1234);  
    obj1.getStringForm();  
    // checking for equality of objects  
    if (obj1.equals(obj2))  
        System.out.println("Objects are equal");  
    else  
        System.out.println("Objects are not equal");  
    obj2=obj1; // assigning reference of obj1 to obj2  
    // checking the equality of objects  
    if (obj1.equals(obj2))  
        System.out.println("Objects are equal");  
    else  
        System.out.println("Objects are not equal");  
}
```

The code shows the use of the `toString()` and `equals()` methods of the `Object` class. Figure 2.7 shows the output of the code.



```

run:
String form of num is: 1234
Objects are not equal
Objects are equal
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 2.7: Result of Object Class Methods

Note that after the reference of `obj1` is assigned to `obj2`, the `equals()` method returns true, otherwise it returns false even if the values passed for `num` were same for both the objects.

2.3.5 Class Class

In an executing Java program, instances of the `Class` class represent classes and interfaces. An enum is a kind of class and an annotation is a kind of interface. Even an array belongs to a `class` that is reflected as a `Class` object that is shared by all arrays with the same element type and number of dimensions. The primitive Java data types such as `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double` as well as the keyword `void` are also represented as `Class` objects.

`Class` class has no public constructor. Instead, `Class` objects are constructed automatically by the JVM as classes are loaded and by calls to the `defineClass()` method in the class loader.

Table 2.4 lists some of the commonly used methods of the `Class` class.

Method	Description
<code>static Class forName(String className)</code>	The method returns the <code>Class</code> object associated with the class or interface with the given string name.
<code>static Class forName(String name, boolean initialize, ClassLoader loader)</code>	The method returns the <code>Class</code> object associated with the class or interface with the given string name, using the given class loader.
<code>Class[] getClasses()</code>	The method returns an array containing <code>Class</code> objects representing all the public classes and interfaces that are members of the class represented by this <code>Class</code> object.
<code>Field getField(String name)</code>	The method returns a <code>Field</code> object that reflects the specified public member field of the class or interface represented by this <code>Class</code> object.
<code>Class[] getInterfaces()</code>	The method determines the interfaces implemented by the class or interface represented by this object.

Method	Description
Method <code>getMethod(String name, Class[] parameterTypes)</code>	The method returns a <code>Method</code> object that reflects the specified public member method of the class or interface represented by this <code>Class</code> object.
<code>int getModifiers()</code>	The method returns the Java language modifiers for this class or interface, encoded in an integer.
<code>String getName()</code>	The method returns the name of the entity (class, interface, array class, primitive type, or void) represented by this <code>Class</code> object, as a <code>String</code> .
<code>URL getResource(String name)</code>	The method finds a resource with a given name.
<code>Class getSuperclass()</code>	The method returns the <code>Class</code> representing the superclass of the entity (class, interface, primitive type, or void) represented by this <code>Class</code> .
<code>boolean isArray()</code>	The method determines if this <code>Class</code> object represents an array class.
<code>boolean isInstance(Object obj)</code>	Determines if the specified <code>Object</code> is assignment-compatible with the object represented by this <code>Class</code> .
<code>boolean isInterface()</code>	The method determines if the specified <code>Class</code> object represents an interface type.
<code>String toString()</code>	The method converts the object to a string.

Table 2.4: Methods of Class in Java API

Code Snippet 5 shows the use of some of the methods of `Class` class.

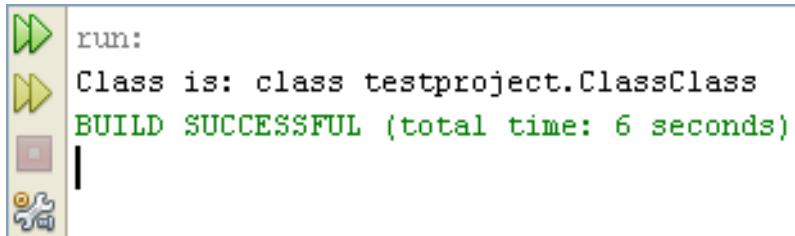
Code Snippet 5:

```
class ClassClass extends MathClass{
    public ClassClass() {}
}

public class TestClass {
    public static void main(String[] args) {
        ClassClass obj = new ClassClass();
        System.out.println("Class is: " + obj.getClass());
    }
}
```

The code shows the use of the `getClass()` method of the `Class` class.

Figure 2.8 shows the output of the code.



```
run:
Class is: class testproject.ClassClass
BUILD SUCCESSFUL (total time: 6 seconds)
```

Figure 2.8: Output of Code Snippet 5

2.3.6 ThreadGroup Class

A thread group represents a set of threads. Besides this, a thread group can also include other thread groups. The thread groups forms a tree in which all the thread group except the initial thread group has a parent.

A thread can access information about its own thread group, but has no right to access information about its thread group's parent thread group or any other thread groups.

Table 2.5 lists some of the commonly used methods of the ThreadGroup class.

Method	Description
int activeCount()	The method returns an estimate of the number of active threads in this thread group and its subgroups.
int activeGroupCount()	The method returns an estimate of the number of active groups in this thread group and its subgroups.
void checkAccess()	The method determines if the currently running thread has permission to modify this thread group.
void destroy()	The method destroys this thread group and all of its subgroups.
int enumerate(Thread[] list)	The method copies into the specified array every active thread in this thread group and its subgroups.
int enumerate(ThreadGroup[] list)	The method copies into the specified array references to every active subgroup in this thread group and its subgroups.
int getMaxPriority()	The method returns the maximum priority of this thread group.
String getName()	The method returns the name of this thread group.
ThreadGroup getParent()	The method returns the parent of this thread group.
void interrupt()	The method interrupts all threads in this thread group.
boolean isDaemon()	The method tests if this thread group is a daemon thread group.
boolean isDestroyed()	The method tests if this thread group has been destroyed.

Method	Description
<code>void list()</code>	The method prints information about this thread group to the standard output.
<code>boolean parentOf(ThreadGroup g)</code>	The method tests if this thread group is either the thread group argument or one of its ancestor thread groups.
<code>void setDaemon(boolean daemon)</code>	The method changes the daemon status of this thread group.
<code>void setMaxPriority(int pri)</code>	The method sets the maximum priority of the group.
<code>String toString()</code>	The method returns a string representation of this Thread group.

Table 2.5: Methods of the ThreadGroup Class

2.3.7 Runtime Class

There is a single instance of class `Runtime` for every Java application allowing the application to interface with the environment in which it is running. The current runtime is obtained by invoking the `getRuntime()` method. An application cannot create its own instance of this class.

Table 2.6 lists some of the methods of the `Runtime` class.

Method	Description
<code>int availableProcessors()</code>	The method returns the number of processors available to the Java virtual machine.
<code>Process exec(String command)</code>	The method executes the specified string command in a separate process.
<code>void exit(int status)</code>	The method terminates the currently running Java virtual machine by initiating its shutdown sequence.
<code>long freeMemory()</code>	The method returns the amount of free memory in the JVM.
<code>void gc()</code>	The method executes the garbage collector.
<code>static Runtime getRuntime()</code>	The method returns the runtime object associated with the current Java application.
<code>void halt(int status)</code>	The method forcibly terminates the currently running JVM.
<code>void load(String filename)</code>	The method loads the specified filename as a dynamic library.
<code>void loadLibrary(String libname)</code>	The method loads the dynamic library with the specified library name.
<code>long maxMemory()</code>	The method returns the maximum amount of memory that the JVM will attempt to use.
<code>void runFinalization()</code>	The method executes the finalization methods of any objects pending finalization.

Method	Description
long totalMemory()	The method returns the total amount of memory in the JVM.

Table 2.6: Methods of the Runtime Class

2.4 Strings

Strings are widely used in Java programming. Strings are nothing but a sequence of characters. In the Java programming language, strings are objects. The Java platform provides the `String` class to create and manipulate strings. For example, the most direct way to create a string is to write:

```
String name = "John";
```

In this case, “John” is a string literal, that is, a series of characters enclosed in double quotes.

Whenever a string literal is encountered in a code, the compiler creates a `String` object with its value, in this case, `John`.

2.4.1 String Class

The `String` class represents character strings. All string literals in Java programs, such as ‘xyz’, are implemented as instances of the `String` class.

The syntax of `String` class is as follows:

Syntax:

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

Strings are constant, that is, their values cannot be changed once created. However, string buffers support mutable strings. Since, `String` objects are immutable they can be shared. For example:

```
String name = "Kate";
```

is equivalent to:

```
char [] name = {'K', 'a', 't', 'e'};  
String strName = new String(name);
```

Similar to other objects, a `String` object can be created by using the `new` keyword and a constructor. The `String` class has thirteen overloaded constructors that allow specifying the initial value of the string using different sources, such as an array of characters as shown in Code Snippet 6.

Code Snippet 6:

```
char[] name = { 'J', 'o', 'h', 'n' };
String nameStr = new String(name);
System.out.println(nameStr);
```

The last line of the code will display **John**. Following are some more examples of strings:

```
System.out.println("Kate");
String lname = "Wilson";
System.out.println("Kate" + lname); // concatenation of strings
```

The **String** class provides methods for manipulating individual characters of the sequence, extracting substrings, comparing strings, searching strings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the **Character** class.

The Java language provides support for the string concatenation operator (+), and the conversion of other objects to strings. String concatenation can also be implemented through the **StringBuilder** or **StringBuffer** class and its append method.

String conversions can be done through the **toString()** method that is inherited from the **Object** class. The **Object** class is the root class and is inherited by all classes in Java. In general, passing a null argument to a constructor or method of this class will throw a **NullPointerException**.

The **String** class provides methods for dealing with Unicode code points, that is, characters, as well as for dealing with Unicode code units, that is, char values.

2.4.2 String Methods

String class provides several methods for manipulating strings. Table 2.7 lists some commonly used methods of the **String** class.

Method	Description
<code>char charAt(int index)</code>	The method returns the char value at the specified index.
<code>int compareTo(String anotherString)</code>	The method compares two strings lexicographically.
<code>String concat(String str)</code>	The method concatenates the specified string to the end of this string.
<code>boolean contains(CharSequence s)</code>	The method returns true if and only if this string contains the specified sequence of char values.
<code>boolean endsWith(String suffix)</code>	The method tests if this string ends with the specified suffix.
<code>boolean equals(Object anObject)</code>	The method compares this string to the specified object.

Method	Description
boolean equalsIgnoreCase(String anotherString)	The method compares this String to another String, ignoring the case.
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	The method copies characters from this string into the destination character array.
int indexOf(int ch)	The method returns the index within this string of the first occurrence of the specified character.
boolean isEmpty()	The method returns true if, and only if, length() is 0.
int lastIndexOf(int ch)	The method returns the index within this string of the last occurrence of the specified character.
int length()	The method returns the length of this string.
boolean matches(String regex)	The method tells whether or not this string matches the given regular expression.
String replace(char oldChar, char newChar)	The method returns a new string resulting from replacing all occurrences of oldChar in this string with newChar .
String[] split(String regex)	The method splits this string around matches of the given regular expression.
String substring(int beginIndex)	The method returns a new string that is a substring of this string.
char[] toCharArray()	The method converts this string to a new character array.
String toLowerCase()	The method converts all characters in the String to lower case using the rules of the default locale.
String toString()	The method returns the object which is already a string itself.
String toUpperCase()	The method converts all characters in the String to upper case using the rules of the default locale.
String trim()	The method returns a copy of the string, with leading and trailing whitespace omitted.

Table 2.7: Methods of the String Class

Apart from these, there is the `valueOf(<type>)` method that has several overloaded versions of all the primitive types including `char`.

Code Snippet 7 shows an example using String methods.

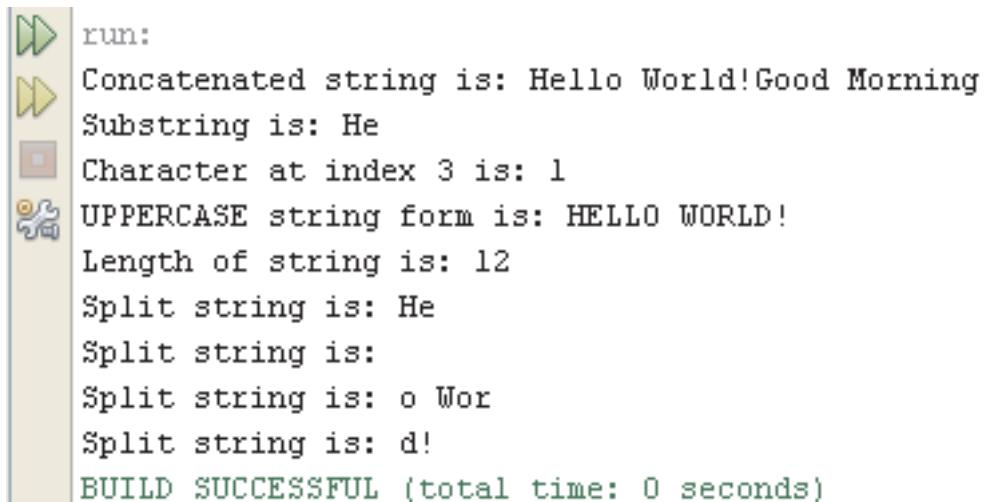
Code Snippet 7:

```
class StringClass {  
    public void manipulateStrings(String str1, String str2)  
    {  
        // concat strings  
        System.out.println("Concatenated string is: " + str1.concat(str2));  
  
        // get substring  
        System.out.println("Substring is: " + str1.substring(0, 2));  
  
        // get character at an index  
        System.out.println("Character at index 3 is: " + str1.charAt(3));  
  
        // convert string to uppercase  
        System.out.println("UPPERCASE string form is: " + str1.toUpperCase());  
  
        // get length  
        System.out.println("Length of string is: " + str1.length());  
        // split string  
        String[] splitted = str1.split("l");  
        for (int i = 0; i < splitted.length; i++)  
            System.out.println("Split string is: " + splitted[i]);  
    }  
}  
  
public class TestString {  
    public static void main(String[] args) {  
        StringClass objStr = new StringClass();  
        objStr.manipulateStrings("HelloWorld!", "GoodMorning");  
    }  
}
```

```
}
```

```
}
```

The code shows the use of `concat()`, `substring()`, `charAt()`, `toUpperCase()`, `length()`, and `split()` methods of the `String` class. Figure 2.9 shows the output of the code.



```

run:
Concatenated string is: Hello World!Good Morning
Substring is: He
Character at index 3 is: l
UPPERCASE string form is: HELLO WORLD!
Length of string is: 12
Split string is: He
Split string is:
Split string is: o Wor
Split string is: d!
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 2.9: Result of String Class Methods

2.4.3 *StringBuilder and StringBuffer Classes*

`StringBuilder` objects are like `String` objects, except that they are mutable. Internally, the runtime treats these objects like variable-length arrays containing a sequence of characters. The length and content of the sequence can be modified at any point through certain method calls.

It is advisable to use `Strings` unless string builders offer an advantage in terms of simpler code or better performance. For example, when it is required to concatenate several strings, appending them to a `StringBuilder` object is more efficient rather than using concatenation operator.

The `StringBuilder` class also has a `length()` method that returns the length of the character sequence in the builder. Unlike strings, a `StringBuilder` also has a capacity that indicates the number of character spaces that have been allocated. The capacity is returned by the `capacity()` method. It is always greater than or equal to the length. The capacity is automatically expanded as necessary to accommodate additions to the `StringBuilder`. Table 2.8 lists the constructors of `StringBuilder` class.

Constructor	Description
<code>StringBuilder()</code>	The method creates an empty string builder with a capacity of 16.
<code>StringBuilder(CharSequence cs)</code>	The method constructs a string builder containing the same characters as the specified <code>CharSequence</code> , plus an extra 16 empty elements trailing the <code>CharSequence</code> .

Constructor	Description
StringBuilder(int initCapacity)	The method creates an empty string builder with the specified initial capacity.
StringBuilder(String s)	The method creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

Table 2.8: Constructors of StringBuilder Class

Code Snippet 8 explains the use of `StringBuilder`.

Code Snippet 8:

```
class StringBuild {

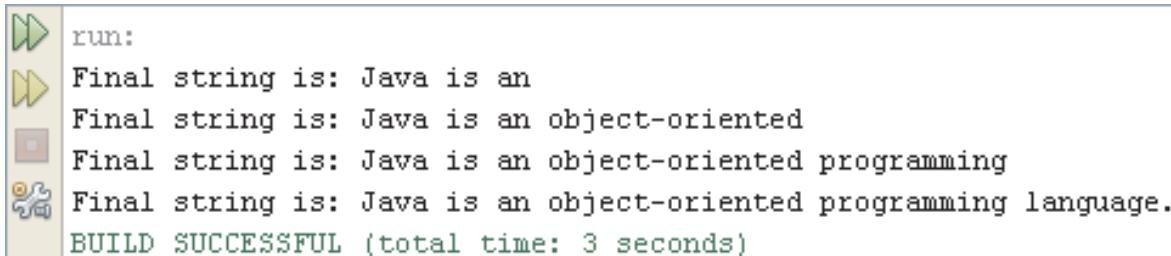
    // creating string builder
    StringBuilder sb = new StringBuilder(); // line 1

    public void addString(String str) {
        // appending string to stringbuilder
        sb.append(str); // line 2
        System.out.println("Final string is: " + sb.toString());
    }
}

public class TestStringBuild {
    public static void main(String[] args) {
        StringBuild sb = new StringBuild();
        sb.addString("Java is an");
        sb.addString("object-oriented");
        sb.addString("programming");
        sb.addString("language.");
    }
}
```

The code shows the creation of the `StringBuilder` object on line 1. Next, a method `addString()` is created to accept the string at runtime. Within the method, the `append()` method is used to add the new string to the `StringBuffer` object. The `TestStringBuild` class consists of the `main()` method within which the `StringBuild` class is instantiated and the `addString()` method is invoked multiple

times. Figure 2.10 shows the output of the code.



```

run:
Final string is: Java is an
Final string is: Java is an object-oriented
Final string is: Java is an object-oriented programming
Final string is: Java is an object-oriented programming language.
BUILD SUCCESSFUL (total time: 3 seconds)

```

Figure 2.10: StringBuilder Output

The `StringBuilder` class provides some methods related to length and capacity which are not available with the `String` class. These are listed in table 2.9.

Method	Description
<code>void setLength(int newLength)</code>	<p>The method sets the length of the character sequence.</p> <ul style="list-style-type: none"> If <code>newLength</code> is less than <code>length()</code>, the last characters in the character sequence are truncated. If <code>newLength</code> is greater than <code>length()</code>, null characters are added at the end of the character sequence.
<code>void ensureCapacity(int minCapacity)</code>	The method ensures that the capacity is at least equal to the specified minimum.

Table 2.9: Methods of `StringBuilder` Class

A number of operations such as `append()`, `insert()`, or `setLength()` can be used to increase the length of the character sequence in the `StringBuilder` class. This will lead to the resultant length becoming greater than the current capacity. In such a case, the capacity is automatically increased.

The main operations on a `StringBuilder` class that the `String` class does not possess, are the `append()` and `insert()` methods. These methods have overloaded versions so as to accept data of any type. However, each method converts its argument to a string and then appends or inserts the characters of the new string to the existing character sequence in the string builder. The `append()` method always adds these characters at the end of the existing character sequence, whereas the `insert()` method adds the characters at a specified index.

→ **StringBuffer**

The `StringBuffer` creates a thread-safe, mutable sequence of characters. Since JDK 5, this class has been supplemented with an equivalent class designed for use by a single thread, `StringBuilder`. The `StringBuilder` class should be preferred over `StringBuffer`, as it supports all of the same operations but it is faster since it performs no synchronization.

The `StringBuffer` class declaration is as follows:

```
public final class StringBuffer
```

```
extends Object
implements Serializable, CharSequence
```

A string buffer is similar to a `String` class except that it is mutable just like the `StringBuilder`. At any instant, it may contain some particular sequence of characters. However, the length and content of the sequence can be modified by invoking certain methods.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary. This causes all the operations on any particular instance to behave as if they occur in some serial order. That is, they are consistent with the order of the method calls made by each of the individual threads that are involved.

All operations that can be performed on `StringBuilder` class are also applicable to `StringBuffer` class. In other words, the `append()` and `insert()` methods that are used with `StringBuffer` class also will result in the same output when used with `StringBuilder` class. However, whenever an operation such as `append` or `insert` occurs that involves a source sequence, this class synchronizes only on the string buffer performing the operation and not on the source. Similar, to `StringBuilder`, every string buffer also has a capacity. Also, if the internal buffer overflows, it automatically increases in size.

2.4.4 Parsing of Text Using StringTokenizer Class

There are different ways of parsing text. The usual tools are as follows:

- `String.split()` method
- `StringTokenizer` and `StreamTokenizer` classes
- `Scanner` class
- `Pattern` and `Matcher` classes, which implement regular expressions
- For the most complex parsing tasks, tools such as **JavaCC** can be used

The `StringTokenizer` class belongs to the `java.util` package and is used to break a string into tokens. The class is declared as follows:

```
public class StringTokenizer
extends Object
implements Enumeration
```

The method for tokenization is much simpler than that used by the `StreamTokenizer` class. The `StringTokenizer` class methods do not differentiate between numbers, identifiers, and quoted strings. Also, they do not recognize and skip comments.

The set of delimiters, that is, the characters that separate tokens, can be specified either at creation time

or on a per-token basis.

Table 2.10 lists the constructors of StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	The method constructs a string tokenizer for the specified string.
StringTokenizer(String str, String delim)	The method creates an empty string tokenizer for the specified string. The <code>delim</code> argument specifies the delimiter to be used for tokenization.
StringTokenizer(String str, String delim, boolean returnDelims)	The method creates an empty string tokenizer for the specified string. The <code>delim</code> argument specifies the delimiter to be used for tokenization. If <code>returnDelims</code> is set to true, the delimiter characters are also returned as tokens.

Table 2.10: Constructors of StringTokenizer Class

A StringTokenizer object behaves in one of the following ways, depending on the value of the `returnDelims` flag, when it was created:

- If the flag is `false`, delimiter characters serve to separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters.
- If the flag is `true`, delimiter characters are themselves considered to be tokens. A token is thus either one delimiter character, or a maximal sequence of consecutive characters that are not delimiters.

An instance of StringTokenizer class internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the instance of the StringTokenizer class.

Table 2.11 lists some of the methods of StringTokenizer class.

Method	Description
<code>int countTokens()</code>	The method calculates the number of times that this tokenizer's <code>nextToken</code> method can be called before it generates an exception.
<code>boolean hasMoreElements()</code>	The method returns the same value as the <code>hasMoreTokens</code> method.
<code>boolean hasMoreTokens()</code>	The method tests if there are more tokens available from the tokenizer's string.
<code>Object nextElement()</code>	The method returns the same value as the <code>nextToken</code> method, except that its declared return value is <code>Object</code> rather than <code>String</code> .
<code>String nextToken()</code>	The method returns the next token from this string tokenizer.

Method	Description
String nextToken(String delim)	The method returns the next token in this string tokenizer's string.

Table 2.11: Methods of StringTokenizer Class

Code Snippet 9 shows the use of StringTokenizer.

Code Snippet 9:

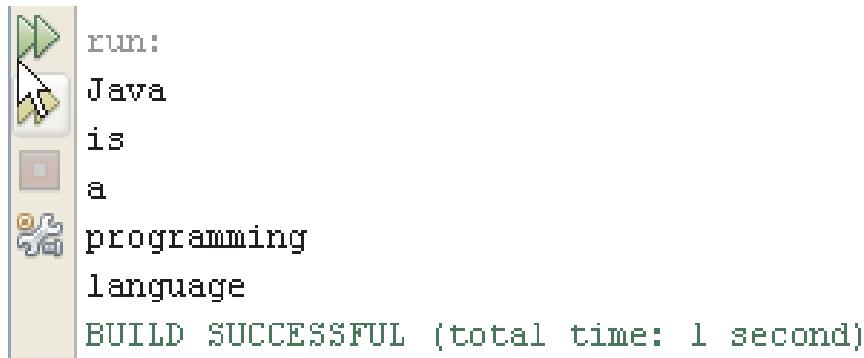
```
import java.util.StringTokenizer;

class StringTokenizer {
    public void tokenizeString(String str, String delim) {
        StringTokenizer st = new StringTokenizer(str, delim);
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}

public class TestProject {
    public static void main(String[] args) {
        StringTokenizer objST = new StringTokenizer();
        objST.tokenizeString("Java,is,a,programming,language", ",");
    }
}
```

The code shows a class StringTokenizer with the tokenizeString() method that takes a source string and a delimiter as parameters. Within the method, an object of StringTokenizer is created and the source string and delimiter are passed as arguments. Next, the hasMoreTokens() method is used to iterate through the StringTokenizer object to print the tokenized strings.

Figure 2.11 shows the output of the code.



```
run:
Java
is
a
programming
language
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 2.11: StringTokenizer Output

`StringTokenizer` is a legacy class that has been retained for compatibility reasons. Its methods do not distinguish between identifiers, numbers, and quoted strings nor do it recognize and skip comments. However, its use is discouraged in new code. It is advisable to use the `split()` method of `String` class or the `java.util.regex` package for tokenization rather than using `StringTokenizer`.

2.5 Regular Expression

Regular expressions are used to describe a set of strings based on the common characteristics shared by individual strings in the set. They are used to edit, search, or manipulate text and data. To create regular expressions, one must learn a particular syntax that goes beyond the normal syntax of the Java. Regular expressions differ in complexity, but once the basics of their creation are understood, it is easy to decipher or create any regular expression.

2.5.1 Regular Expression API

For creating regular expressions, there are many different options available such as Perl, grep, Python, Tcl, Python, awk, and PHP. In Java, one can use `java.util.regex API` to create regular expressions. The syntax for regular expression in the `java.util.regex` API is very similar to that of Perl.

There are primarily three classes in the `java.util.regex` package that are required for creation of regular expression. They are as follows:

- **Pattern:** A `Pattern` object is a compiled form of a regular expression. There are **no public constructors available in the Pattern class**. To create a pattern, it is required to first invoke one of its `public static compile()` methods. These methods will then return an instance of `Pattern` class. The first argument of these methods is a regular expression.
- **Matcher:** A `Matcher` object is used to **interpret the pattern and perform match operations against an input string**. Similar to the `Pattern` class, the `Matcher` class also provides no public constructors. To obtain a `Matcher` object, it is required to invoke the `matcher()` method on a `Pattern` object.

- **PatternSyntaxException:** A PatternSyntaxException object is an unchecked exception used to indicate a syntax error in a regular expression pattern.

2.5.2 Pattern Class

Any regular expression that is specified as a string must first be compiled into an instance of the Pattern class. The resulting pattern object can then be used to create a Matcher object. Once the Matcher object is obtained, the Matcher object can then match arbitrary character sequences against the regular expression. All the different state involved in performing a match resides in the matcher; so several matchers can share the same pattern. The declaration of the Pattern class is as follows:

Syntax:

```
public final class Pattern
extends Object
implements Serializable
```

A Pattern is a compiled representation of a regular expression.

A typical invocation sequence would be:

```
// creating the Pattern object with the regular expression
Pattern p = Pattern.compile("a*b");
// creating the Matcher object using the Pattern object with the input
//string
Matcher m = p.matcher("aaaaab");
// checking for a match
boolean b = m.matches();
```

The matches() method of the Matcher class is defined for use when a regular expression is used just once. This method will compile an expression and match it against an input sequence in a single invocation. The statement:

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

is an equivalent single statement for the three statements used earlier. However, for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

Instances of the Pattern class are immutable and therefore, safe for use by multiple concurrent threads whereas instances of the Matcher class are not safe for such use.

2.5.3 Matcher Class

A Matcher object is created from a pattern by invoking the matcher() method on the Pattern object. A Matcher object is the engine that performs the match operations on a character sequence by interpreting a Pattern.

The declaration of the `Matcher` class is as follows:

Declaration:

```
public final class Matcher
extends Object
implements MatchResult
```

After creation, a `Matcher` object can be used to perform three different types of match operations:

- The `matches()` method is used to **match the entire input** sequence against the pattern.
- The `lookingAt()` method is used to **match the input sequence**, starting at the **beginning**, against the pattern.
- The `find()` method is used to **scan** the **input** sequence **looking for** the next **subsequence** that **matches the pattern**.

The return type of each of these methods is a `boolean` indicating success or failure. Additional information about a successful match can be obtained by querying the state of the matcher.

A matcher finds matches in a subset of its input. This input is called the region. By default, the region contains all of the matcher's input. The region can be modified by using the `region()` method. To query the region, one can use the `regionStart()` and `regionEnd()` methods. Also, one can change the way that the region boundaries interact with some pattern constructs.

Matcher class consists of index methods that provide useful index values that can be used to indicate exactly where the match was found in the input string. These are as follows:

- `public int start():` Returns the start index of the previous match.
- `public int start(int group):` Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end():` Returns the offset after the last character matched.
- `public int end(int group):` Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Table 2.12 lists other important methods of the `Matcher` class.

Method	Description
<code>Matcher appendReplacement(StringBuffer sb, String replacement)</code>	The method implements a non-terminal append-and-replace step.

Method	Description
<code>StringBuffer appendTail(StringBuffer sb)</code>	The method implements a terminal append-and-replace step.
<code>boolean find()</code>	The method finds the next subsequence of the input sequence that matches the pattern.
<code>boolean find(int start)</code>	The method resets this matcher and then finds the next subsequence of the input sequence that matches the pattern, starting at the specified index.
<code>String group()</code>	The method returns the input subsequence matched by the previous match.
<code>String group(int group)</code>	The method returns the input subsequence captured by the given group during the previous match operation.
<code>String group(String name)</code>	The method returns the input subsequence captured by the given named-capturing group during the previous match operation.
<code>int groupCount()</code>	The method returns the number of capturing groups in this matcher's pattern.
<code>boolean hitEnd()</code>	The method returns true if the end of input was hit by the search engine in the last match operation performed by this matcher.
<code>boolean lookingAt()</code>	The method matches the input sequence against the pattern, starting at the beginning of the region.
<code>boolean matches()</code>	The method matches the entire region against the pattern.
<code>Pattern pattern()</code>	The method returns the pattern that is interpreted by this matcher.
<code>static String quoteReplacement(String s)</code>	The method returns a literal replacement String for the specified String.
<code>Matcher region(int start, int end)</code>	The method sets the limits of this matcher's region.
<code>int regionEnd()</code>	The method reports the end index (exclusive) of this matcher's region.
<code>int regionStart()</code>	The method reports the start index of this matcher's region.
<code>String replaceAll(String replacement)</code>	The method replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
<code>String replaceFirst(String replacement)</code>	The method replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
<code>boolean requireEnd()</code>	The method returns true if more input could change a positive match into a negative one.
<code>Matcher reset()</code>	The method resets this matcher.

Method	Description
Matcher reset(CharSequence input)	The method resets this matcher with a new input sequence.
int start()	The method returns the start index of the previous match.
int start(int group)	The method returns the start index of the subsequence captured by the given group during the previous match operation.
MatchResult toMatchResult()	The method returns the match state of this matcher as a MatchResult.
String toString()	The method returns the string representation of this matcher.
Matcher usePattern(Pattern newPattern)	The method changes the Pattern that this Matcher uses to find matches with.

Table 2.12: Methods of Matcher Class

The explicit state of a matcher includes:

- the start and end indices of the most recent successful match.
- the start and end indices of the input subsequence captured by each capturing group in the pattern.
- the total count of such subsequences.

Methods are also provided with the facility for returning these captured subsequences in string form.

The explicit state of a matcher is initially undefined. At this point, an attempt to query any part of matcher match will cause an `IllegalStateException` to be thrown. The explicit state of a matcher is recomputed with every match operation.

The implicit state of a matcher includes:

- input character sequence.
- append position, which is initially zero. It is updated by the `appendReplacement()` method.

The `reset()` method helps the matcher to be explicitly reset. If a new input sequence is desired, the `reset(CharSequence)` method can be invoked. The reset operation on a matcher discards its explicit state information and sets the append position to zero.

Instances of the `Matcher` class are not safe for use by multiple concurrent threads.

Code Snippet 10 explains the use of `Pattern` and `Matcher` for creating and evaluating regular expressions:

Code Snippet 10:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest{
    public static void main(String[] args) {
        String flag;

        while (true) {

            Pattern pattern1 =
                Pattern.compile(System.console().readLine("%nEnter expression: "));
            Matcher matcher1 =
                pattern1.matcher(System.console().readLine("Enter string to search:
"));
            boolean found=false;
            while (matcher1.find()) {
                System.console().format("Found the text" + "\"%s\" starting at " +
"index %d and ending at index %d.%n", matcher1.group(),
                matcher1.start(), matcher1.end());
                found=true;
            }

            if(!found) {
                System.console().format("No match found.%n");
            }
            // code to exit the application
            System.console().format("Press x to exit or y to continue");
        }
    }
}
```

```

flag=System.console().readLine("%nEnter your choice: ");

if(flag.equals("x"))

    System.exit(0);

else

    continue;

}

}

}

```

In the code, a while loop has been created inside the `RegexTest` class. Within the loop, a `Pattern` object is created and initialized with the regular expression specified at runtime using the `System.console().readLine()` method. Similarly, the `Matcher` object has been created and initialized with the input string specified at runtime. Next, another while loop has been created to iterate till the `find()` method returns true. Within the loop, the `group()`, `start()`, and `end()` methods are used to display the text, start index, and end index respectively. The output has been formatted using the `format()` method.

The variable `found` is set to true, when a match is found. When no match is found, appropriate message is displayed to the user.

Lastly, the user is asked if he/she wishes to continue. If the user provides x as input, the application is terminated. Otherwise, the outer while loop continues. Figure 2.12 shows the output of the code.

```

C:\WINDOWS\system32\cmd.exe
E:\>javac RegexTest.java
E:\>java RegexTest
Enter expression: abc
Enter string to search: abcdefabcghi
Found the text "abc" starting at index 0 and ending at index 3.
Found the text "abc" starting at index 6 and ending at index 9.
Press x to exit or y to continue
Enter your choice: x
E:\>_

```

Figure 2.12: Output of Matcher

The output shows the expression `abc` found in the search string `abcdefabcghi` along with the start and end index where it was found.

2.6 String Literal

The most basic form of pattern matching supported by the `java.util.regex` API is the match of a string literal. For example, consider the regular expression as `abc` and the input string as `abcdefabcghi` in Code Snippet 10. The match will succeed because the regular expression is found in the string.

Note that in the match, the start index is counted from 0. By convention, ranges are inclusive of the beginning index and exclusive of the end index. Each character in the string resides in its own cell, with the index positions pointing between each cell as shown in figure 2.13.

Cell	0	1	2	3	4	5	6	7	8	9	10	11
	a	b	c	d	e	f	a	b	c	g	h	i
Index	0	1	2	3	4	5	6	7	8	9	10	11

Figure 2.13: The String Literal with Index and Cell Numbers

The string **abcdefabcghi** starts at index 0 and ends at index 12, even though the characters themselves only occupy cells 0, 1, ..., and 11.

If the search string is changed to **abcababcabc**, with subsequent matches, some overlapping is noticed. The start index for the next match is the same as the end index of the previous match as shown in figure 2.14.

```
C:\WINDOWS\system32\cmd.exe - java RegexTest
E:\>java RegexTest
Enter expression: abc
Enter string to search: abcabcabcabc
Found the text "abc" starting at index 0 and ending at index 3.
Found the text "abc" starting at index 3 and ending at index 6.
Found the text "abc" starting at index 6 and ending at index 9.
Found the text "abc" starting at index 9 and ending at index 12.
Press x to exit or y to continue
Enter your choice: -
```

Figure 2.14: Overlapping Indices

Figure 2.14 shows the overlapping of start and end indexes for each subsequent match.

2.6.1 Metacharacters

This API also supports many special characters. This affects the way a pattern is matched. For example, if the regular expression is changed to **abc.** and the input string to **abcd**, the output will appear as shown in figure 2.15.

```
C:\WINDOWS\system32\cmd.exe - java RegexTest
E:\>java RegexTest
Enter expression: abc.
Enter string to search: abcd
Found the text "abcd" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: _
```

Figure 2.15: Metacharacter Ignored

The match still succeeds, even though the dot '.' is not present in the input string. This is because the dot is a metacharacter, that is, a character with special meaning as interpreted by the matcher. For the matcher, the metacharacter '.' stands for 'any character'. This is why the match succeeds in the example. The metacharacters supported by the API are: < ([{ \^-=\\$!|] }) ?*+.>

Note - These special characters may not be treated as metacharacters in certain situation. However, this list can be used to check whether or not a specific character will ever be considered a metacharacter. For example, the characters @ and # will never carry a special meaning.

One can force metacharacters to be treated as an ordinary character in one of the following ways:

- By preceding the metacharacters with a backslash.
- By enclosing it within \Q (starts the quote) and \E (ends the quote). The \Q and \E can be placed at any location within the expression. However, the \Q must comes first.

Figure 2.16 shows the use of backslash to force the metacharacter to be treated as an ordinary character.

```
C:\WINDOWS\system32\cmd.exe - java RegexTest
E:\>java RegexTest
Enter expression: abc\.
Enter string to search: abcd
No match found.
Press x to exit or y to continue
Enter your choice: _
```

Figure 2.16: Metacharacter Considered for Match

Figure 2.16 shows that the match fails, since '.' is not found in the search string.

2.7 Character Classes

The word ‘class’ in ‘character class’ phrase does not mean a .class file. With respect to regular expressions, a character class is a set of characters enclosed within square brackets. It indicates the characters that will successfully match a single character from a given input string. Table 2.13 summarizes the supported regular expression constructs in ‘Character Classes’.

Construct	Type	Description
[abc]	Simple class	a, b, or c
[^abc]	Negation	Any character except a, b, or c
[a-zA-Z]	Range	a through z, or A through Z (inclusive)
[a-d[m-p]]	Union	a through d, or m through p: [a-dm-p]
[a-z&&[def]]	Intersection	d, e, or f
[a-z&&[^bc]]	Subtraction	a through z, except for b and c: [ad-z]
[a-z&&[^m-p]]	Subtraction	a through z, and not m through p: [a-lq-z]

Table 2.13: Regular Expression Constructs

2.7.1 Simple Classes

This is the most basic form of a character class. It is created by specifying a set of characters side-by-side within square brackets. For example, the regular expression [fmc]at will match the words ‘fat’, ‘mat’, or ‘cat’. This is because the class defines a character class accepting either ‘f’, ‘m’, or ‘c’ as the first character. Figure 2.17 shows the output when the expression and input strings are modified in Code Snippet 10.

```

C:\WINDOWS\system32\cmd.exe

E:\>java RegexTest

Enter expression: [fmc]at
Enter string to search: fat
Found the text "fat" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [fmc]at
Enter string to search: mat
Found the text "mat" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [fmc]at
Enter string to search: cat
Found the text "cat" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [fmc]at
Enter string to search: rat
No match found.
Press x to exit or y to continue
Enter your choice: x

```

Figure 2.17: Using Simple Classes

Figure 2.17 shows that in all the cases, the overall match succeeds only when the first letter matches any one of the characters defined by the character class.

2.7.2 Negation

Negation is used to match all characters except those listed in the brackets. The '^' metacharacter is inserted at the beginning of the character class to implement Negation. Figure 2.18 shows the use of Negation.

```
C:\WINDOWS\system32\cmd.exe - java RegexTest
E:\>java RegexTest
Enter expression: [^fmc]at
Enter string to search: fat
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [^fmc]at
Enter string to search: mat
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [^fmc]at
Enter string to search: cat
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [^fmc]at
Enter string to search: rat
Found the text "rat" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: _
```

Figure 2.18: Using Negation

The figure shows that the match is successful only if the first character of the input string does not match any of the characters defined by the character class.

2.7.3 Ranges

At times, it may be required to define a character class that includes a range of values, such as the letters 'a to f' or numbers '1 to 5'. A range can be specified by simply inserting the '-' metacharacter between the first and last character to be matched. For example, [a-h] or [1-5] can be used for a range. One can also place different ranges next to each other within the class in order to further expand the match possibilities. For example, [a-zA-Z] will match any letter of the alphabet from a to z (lowercase) or A to Z (uppercase). Figure 2.19 shows the use of Range and Negation.

```
E:\>java RegexTest
Enter expression: [p-t]
Enter string to search: s
Found the text "s" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [p-t]
Enter string to search: q
Found the text "q" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: rno[5-9]
Enter string to search: rno7
Found the text "rno7" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [x-z]
Enter string to search: a
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: rno[5-9]
Enter string to search: rno2
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: rno[^5-9]
Enter string to search: rno2
Found the text "rno2" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [1-5]
Enter string to search: 5
Found the text "5" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Figure 2.19: Using Range and Negation

2.7.4 Unions

Unions can be used to create a single character class comprising two or more separate character classes. This can be done by simply nesting one class within the other. For example, the union **[a-d[f-h]]** creates a single character class that matches the characters **a, b, c, d, f, g, and h**.

Figure 2.20 shows the use of Unions.

```
E:\>java RegexTest
Enter expression: [a-d[f-h]]
Enter string to search: c
Found the text "c" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [a-d[f-h]]
Enter string to search: g
Found the text "g" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [a-d[f-h]]
Enter string to search: e
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [a-d[f-h]]
Enter string to search: i
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.20: Using Unions

Note that when 'e' is specified in the search string, no match is found. This is because the first range includes a to d and the second one includes f to h.

2.7.5 Intersections

Intersection is used to create a single character class that matches only the characters which are common to all of its nested classes. This is done by using the &&, such as in `[0-6&&[234]]`. This creates a single character class that will match only the **numbers common to both character classes**, that is, 2, 3, and 4. Figure 2.21 shows the use of Intersections.

```
E:\>java RegexTest
Enter expression: [0-6&&[234]]
Enter string to search: 3
Found the text "3" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[234]]
Enter string to search: 2
Found the text "2" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[234]]
Enter string to search: 4
Found the text "4" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[234]]
Enter string to search: 5
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.21: Using Intersections

Figure 2.21 shows that even though 5 is part of the range 0-6, it gives the output as no match found. This is because, `&&` has been used with `0-6` and `[234]`, and `5` does not satisfy this condition.

2.7.6 Subtraction

Subtraction can be used to negate one or more nested character classes, such as `[0-6&&[^234]]`. In this case, the character class will match everything from 0 to 6, except the numbers 2, 3, and 4. Figure 2.22 shows the use of Subtraction.

```
E:\>java RegexTest
Enter expression: [0-6&&[^234]]
Enter string to search: 2
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[^234]]
Enter string to search: 3
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[^234]]
Enter string to search: 4
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[^234]]
Enter string to search: 5
Found the text "5" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Figure 2.22: Using Subtraction

2.7.7 Pre-defined Character Classes

There are several predefined character classes in the Pattern API. These classes provide convenient shortcuts for commonly used regular expressions. Table 2.14 lists the pre-defined character classes:

Construct	Description
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Table 2.14: Pre-defined Character Classes

Here, `\d` is used for specifying a range of digits (0–9) and `\w` means a word character, that is, any lowercase letter, any uppercase letter, the underscore character, or any digit. Using the predefined classes whenever applicable helps to make the code easier to read and eliminates errors introduced by malformed character classes.

Constructs beginning with a backslash are called escaped constructs. When escaped constructs are used within a string literal, **the backslash must be preceded with another backslash for the string to compile**. For example:

```
private final String EXPR = "\\d"; // a single digit
```

Here, `\d` is the regular expression and the **extra backslash is required for the code to compile**. However, when the code reads the expressions directly from the console, the extra backslash is not required.

Figure 2.23 shows the use of pre-defined character classes.

```
E:\>java RegexTest
Enter expression: .
Enter string to search: #
Found the text "#" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: \d
Enter string to search: 5
Found the text "5" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: \D
Enter string to search: 6
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: \s
Enter string to search: d
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: \S
Enter string to search: d
Found the text "d" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: \w
Enter string to search: !
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: \W
Enter string to search: !
Found the text "!" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.23: Using Pre-defined Character Classes

2.8 Quantifiers

Quantifiers can be used to specify the number of occurrences to match against. Table 2.15 shows the greedy, reluctant, and possessive quantifiers. At first glance it may appear that the quantifiers $X?$, $X??$, and $X?+$ do exactly the same thing, since they all promise to match x , once or not at all. However, there are subtle differences so far as implementation is concerned between each of these quantifiers.

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

Table 2.15: List of Quantifiers

2.8.1 Using the Greedy, Reluctant, and Possessive Quantifiers

The use of Quantifiers can be understood from the following test cases:

→ **Case 1**

Consider three regular expressions created with the letter ‘b’ followed by either ?, *, or +. Figure 2.24 shows what happens when these expressions are applied to an empty input string “”:

```
E:\>java RegexTest
Enter expression: b?
Enter string to search:
Found the text "" starting at index 0 and ending at index 0.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b*
Enter string to search:
Found the text "" starting at index 0 and ending at index 0.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b+
Enter string to search:
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.24: Using Quantifiers

Figure 2.24 shows that the match is successful in the first two cases. This is because, both the expressions `b?` and `b*`, allow for zero occurrences of the letter b whereas `b+` requires at least one occurrence. Also, notice that both the start and end indices are zero. This is because, the empty input string "" has no length and therefore, the test simply matches nothing at index 0. These types of matches are known as zero-length matches.

A zero-length match can occur:

- in an empty input string
- at the beginning of an input string
- after the last character of an input string
- in between any two characters of an input string

Zero-length matches can be easily identified because they always start and end at the same index position.

→ Case 2

Change the input string to a single letter 'b'. Figure 2.25 shows the result of using quantifiers on it.

```
E:\>java RegexTest
Enter expression: b?
Enter string to search: b
Found the text "b" starting at index 0 and ending at index 1.
Found the text "" starting at index 1 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b*
Enter string to search: b
Found the text "b" starting at index 0 and ending at index 1.
Found the text "" starting at index 1 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b+
Enter string to search: b
Found the text "b" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Figure 2.25: Using Quantifiers with a Single Letter

In this case, all the three quantifiers found the letter 'b'. However, the first two quantifiers also found a zero-length match at index 1; that is, after the last character of the input string. This is because, the matcher sees the character 'b' as sitting in the cell between index 0 and index 1, and the code loops until it can no longer find a match. Based on the quantifier used, the presence of 'nothing' at the index after the last character may or may not trigger a match.

→ Case 3

Change the input string to ‘**bbbbbb**’ and the result will be as shown in figure 2.26.

```
E:\>java RegexTest
Enter expression: b?
Enter string to search: bbbbb
Found the text "b" starting at index 0 and ending at index 1.
Found the text "b" starting at index 1 and ending at index 2.
Found the text "b" starting at index 2 and ending at index 3.
Found the text "b" starting at index 3 and ending at index 4.
Found the text "b" starting at index 4 and ending at index 5.
Found the text "" starting at index 5 and ending at index 5.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b*
Enter string to search: bbbbb
Found the text "bbbbbb" starting at index 0 and ending at index 5.
Found the text "" starting at index 5 and ending at index 5.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b+
Enter string to search: bbbbb
Found the text "bbbbbb" starting at index 0 and ending at index 5.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Figure 2.26: Using Quantifiers with Repetitive Letters

Note that the expression **b?** finds an individual match for each character. This is because, it matches when ‘**b**’ appears zero or one times.

The expression **b*** could find two separate matches, that is, all of the letters ‘**b**’ in the first match and then, the zero-length match after the last character at index 5.

Finally, **b+** matched all occurrences of the letter ‘**b**’ but ignored the presence of ‘nothing’ at the last index.

→ Case 4

Now, the question is, what would be the result if the first two quantifiers encounter a letter other than ‘**b**’. For example, what would happen if it encounters the letter ‘**c**’, as in ‘**bcbcbbbb**’?

Figure 2.27 shows the result.

```
E:\>java RegexTest
Enter expression: b?
Enter string to search: bcbcbbbb
Found the text "b" starting at index 0 and ending at index 1.
Found the text "" starting at index 1 and ending at index 1.
Found the text "b" starting at index 2 and ending at index 3.
Found the text "" starting at index 3 and ending at index 3.
Found the text "b" starting at index 4 and ending at index 5.
Found the text "b" starting at index 5 and ending at index 6.
Found the text "b" starting at index 6 and ending at index 7.
Found the text "b" starting at index 7 and ending at index 8.
Found the text "" starting at index 8 and ending at index 8.
Found the text "" starting at index 9 and ending at index 9.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b*
Enter string to search: bcbcbbbb
Found the text "b" starting at index 0 and ending at index 1.
Found the text "" starting at index 1 and ending at index 1.
Found the text "b" starting at index 2 and ending at index 3.
Found the text "" starting at index 3 and ending at index 3.
Found the text "bbbb" starting at index 4 and ending at index 8.
Found the text "" starting at index 8 and ending at index 8.
Found the text "" starting at index 9 and ending at index 9.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b+
Enter string to search: bcbcbbbb
Found the text "b" starting at index 0 and ending at index 1.
Found the text "b" starting at index 2 and ending at index 3.
Found the text "bbbb" starting at index 4 and ending at index 8.
Press x to exit or y to continue
Enter your choice: x
E:\>
```

Figure 2.27: Using Quantifiers with Modified String

Notice that, even though letter ‘c’ is present in cells 1, 3, and 8, the result displays a zero-length match at those locations. This is because, the regular expression **b?** is not specifically looking for the letter ‘b’. It is simply looking for the presence or absence of the letter ‘b’. If the quantifier allows a match of ‘b’ zero times, anything in the input string that is not ‘b’ will be displayed as a zero-length match. The remaining b’s will be matched according to the rules of the quantifier.

→ Case 5

To match a pattern exactly n number of times, specify the number inside a set of braces. The result is shown in figure 2.28.

```
E:\>java RegexTest

Enter expression: b{3}
Enter string to search: bb
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b{3}
Enter string to search: bbb
Found the text "bbb" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: y

Enter expression: b{3}
Enter string to search: bbbb
Found the text "bbb" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Figure 2.28: Matching a Pattern Exactly n Times

In this case, the regular expression `b{3}` looks for three occurrences of the letter 'b' in a row. Now, the first match fails because there aren't enough b's in the input string. The second input string triggers a match because it contains exactly 3 b's. The third input string also matches because at the beginning of the input string there are exactly 3 b's that are required to match. Also, anything following that is of no relevance to the first match. However, if the pattern appears again after that point, it would trigger subsequent matches as shown in figure 2.29.

```
E:\>java RegexTest

Enter expression: b{3}
Enter string to search: bbbbbbbb
Found the text "bbb" starting at index 0 and ending at index 3.
Found the text "bbb" starting at index 3 and ending at index 6.
Found the text "bbb" starting at index 6 and ending at index 9.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Figure 2.29: Matching a Pattern Exactly n Times Repeatedly

To make a pattern that appears at least n times, a comma is added after the number as shown in figure 2.30.

```
E:\>java RegexTest
Enter expression: b{3,}
Enter string to search: bbbbbbbbbb
Found the text "bbbbbbbbb" starting at index 0 and ending at index 9.
Press x to exit or y to continue
Enter your choice: x
E:\>_
```

Figure 2.30: Matching a Pattern Atleast n Times

Note that with the same input string, this class finds only one match, because the 9 b's in a row fulfill the condition of 'at least' 3 b's.

To specify an upper limit on the number of occurrences, a second number is specified inside the braces as shown in figure 2.31.

```
E:\>java RegexTest
Enter expression: b{3,5}
Enter string to search: bbbbbbbbbb
Found the text "bbbbb" starting at index 0 and ending at index 5.
Found the text "bbbb" starting at index 5 and ending at index 9.
Press x to exit or y to continue
Enter your choice: x
E:\>
```

Figure 2.31: Specifying the Upper Limit for a Pattern

Here, **b{3,5}** means find atleast 2 but not more than 5 b's in a row. Therefore, the first match is forced to stop at the upper limit of 5 characters. The second match includes whatever is left over. Since 4 b's are left, it satisfies the upper limit condition and so they are for this match. If the input string had been two characters shorter, the second match would have failed since only two b's would have been left after the first match.

→ Case 6

So far, the quantifiers have been tested only on input strings containing one character. In fact, quantifiers can attach to only one character at a time. Therefore, the regular expression '**xyz+**' would mean 'x', followed by 'y', followed by 'z' one or more times. It would not mean '**xyz**' one or more times. However, quantifiers can also be attached to **Character Classes and Capturing Groups**, such as **[xyz]+**, that is, **x or y, or z, one or more times**. Also, **(xyz)+**, that is, the group 'xyz', one or more times. Figure 2.32 shows an example of using Capturing Groups with quantifiers.

```
E:\>java RegexTest

Enter expression: <bat>{3}
Enter string to search: batbatbatbatbatbat
Found the text "batbatbat" starting at index 0 and ending at index 9.
Found the text "batbatbat" starting at index 9 and ending at index 18.
Press x to exit or y to continue
Enter your choice: y

Enter expression: bat{3}
Enter string to search: batbatbatbatbatbat
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.32: Quantifiers with Capturing Groups

In this case, the first example finds two matches because the quantifier applies to the entire capturing group. In the second example, the parentheses are removed and the match fails. This is because the quantifier {3} now applies only to the letter 't'.

Similarly, figure 2.33 shows how to apply a quantifier to an entire character class.

```
E:\>java RegexTest

Enter expression: [xyz]{3}
Enter string to search: xyzzxyxxxzzyyyz
Found the text "xyz" starting at index 0 and ending at index 3.
Found the text "zxy" starting at index 3 and ending at index 6.
Found the text "xxx" starting at index 6 and ending at index 9.
Found the text "zyy" starting at index 9 and ending at index 12.
Found the text "yyz" starting at index 12 and ending at index 15.
Press x to exit or y to continue
Enter your choice: y

Enter expression: xyz{3}
Enter string to search: xyzzxyxxxzzyyyz
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.33: Quantifiers with Character Classes

In this case, the quantifier {3} applies to the entire character class in the first example, but only to the letter 'z' in the second one.

2.8.2 Differences Among the Quantifiers

Table 2.16 lists the differences between the greedy, reluctant, and possessive quantifiers.

Greedy	Reluctant	Possessive
The greedy quantifiers are termed 'greedy' because they force the matcher to read the entire input string before attempting the first match. giọng possessive nhung lap lai nhieu lan	The reluctant quantifiers take the opposite approach. tim match it nhat	The possessive quantifiers always eat the entire input string, trying once and only once for a match. cd ".++f" tim xong .++ roi moi tim f
If in the first attempt to match the entire input string, fails, then the matcher backs off the input string by one character and tries again.	They start at the beginning of the input string and then, reluctantly read one character at a time looking for a match.	Unlike the greedy quantifiers, they never back off , even if doing so would allow the overall match to succeed.
It repeats the process until a match is found or there are no more characters left to back off from.	The last thing they try is to match the entire input string.	
Depending on the quantifier used in the expression, the last thing it will attempt is to try to match against 1 or 0 characters.		

Table 2.16: Differences between Quantifiers

To understand the difference, consider the input string **xbatxxxxxxxxbat**.

The result of using the three quantifiers is shown in figure 2.34.

```
E:\>java RegexTest

Enter expression: .*bat
Enter string to search: xbatxxxxxbat
Found the text "xbatxxxxxbat" starting at index 0 and ending at index 13.
Press x to exit or y to continue
Enter your choice: y

Enter expression: .*?bat
Enter string to search: xbatxxxxxbat
Found the text "xbat" starting at index 0 and ending at index 4.
Found the text "xxxxxbat" starting at index 4 and ending at index 13.
Press x to exit or y to continue
Enter your choice: y

Enter expression: .*+bat
Enter string to search: xbatxxxxxbat
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Figure 2.34: Differentiating the Quantifiers

Here, the first example the greedy quantifier `.*` is used to find ‘anything’, zero or more times, followed by the letters ‘b’, ‘a’, and ‘t’. Since the quantifier is greedy, the `.*` portion of the expression first reads the entire input string. At this point, the overall expression does not succeed, because the last three letters (‘b’, ‘a’, and ‘t’) have already been consumed. Therefore, the matcher slowly backs off one letter at a time until the rightmost occurrence of ‘bat’ is re-encountered. Once it finds the entire ‘bat’, the match succeeds and the search ends.

The second example is of the reluctant quantifier. It starts by first consuming ‘nothing’. Since ‘bat’ does not appear at the beginning of the string, it is forced to swallow the first letter (‘x’). This triggers the first match at 0 and 4. The code then continues the process until the input string is exhausted. It finds another match at 4 and 13.

The third example fails to find a match because the quantifier is possessive. In this case, the entire input string is consumed by `.*+`, leaving nothing left over to satisfy the ‘bat’ at the end of the expression. Therefore, the possessive quantifier should be used in situations where it is necessary to seize all of something without ever backing off. It will do better than the equivalent greedy quantifier in cases where the match is not immediately found.

2.9 Capturing Groups

Capturing groups allows the programmer to consider multiple characters as a single unit. This is done by placing the characters to be grouped inside a set of parentheses. For example, the regular expression `(bat)` creates a single group. The group contains the letters ‘b’, ‘a’, and ‘t’. The part of the input string that matches the capturing group will be saved in memory to be recalled later using backreferences.

2.9.1 Numbering

Capturing groups are numbered by counting their opening parentheses from left to right. For example, in the expression `((X)(Y(Z)))`, there are four such groups namely, `((X))`, `(X)`, `(Y(Z))`, and `(Z)`.

The `groupCount()` method can be invoked on the matcher object to find out how many groups are present in the expression. This method will return an `int` value indicating the number of capturing groups present in the matcher's pattern. Therefore, in this example, the `groupCount()` method would return `4`, indicating that the pattern contains four capturing groups.

There is another special group, group 0, which always represents the entire expression. However, this group is not counted in the total returned by `groupCount()`. Groups beginning with the character '?' are pure, non-capturing groups as they do not capture text and also do not count towards the group total.

It is required to have a clear understanding of how groups are numbered because some `Matcher` methods accept an `int` value. The value specifies a particular group number as a parameter. These are as follows:

- `public int start(int group)`: Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end (int group)`: Returns the index of the last character, plus one, of the subsequence captured by the given group during the previous match operation.
- `public String group (int group)`: Returns the input subsequence captured by the given group during the previous match operation.

An example of using `groupCount()` is shown in Code Snippet 11.

Code Snippet 11:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest1{

    public static void main(String[] args) {
        Pattern pattern1 =
            Pattern.compile("((X)(Y(Z))))";
        Matcher matcher1 =
```

```

pattern1.matcher("((X)(Y(Z)) )");
System.console().format("Group count is: %d", matcher1.groupCount());
}
}

```

Figure 2.35 shows the output of the code.

```

E:\>javac RegexTest1.java
E:\>java RegexTest1
Group count is: 4

```

Figure 2.35: Using the groupCount() Method

2.9.2 Backreferences

`pattern((\d\d)\\\1)`

The portion of the input string that matches the capturing group(s) is saved in memory for later recall with the help of backreference. A **backreference** is specified in the regular expression as a **backslash** (\) followed by a digit indicating the number of the group to be recalled. For example, the expression `(\d\d)` defines one capturing group matching two digits in a row, which can be recalled later in the expression by using the backreference `\1`.

Figure 2.36 shows an example for using backreferences.

```

E:\>java RegexTest
Enter expression: (\d\d)\\\1
Enter string to search: 2323
Found the text "2323" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: y

Enter expression: (\d\d)\\\1
Enter string to search: 2312
No match found.
Press x to exit or y to continue
Enter your choice: x
E:\>_

```

Figure 2.36: Using Backreferences

The first example matches any 2 digits, followed by the exact same two digits. The match succeeds in this case. In the second example, the last two digits are changed, so the match fails.

For nested capturing groups also backreferencing works in exactly the same way. This is done by specifying a backslash followed by the number of the group to be recalled.

2.10 Boundary Matchers

Pattern matches can be made more precise by specifying the information with boundary matchers. For example, finding a specific word only if it appears at the beginning or end of a line or maybe to verify if the match is occurring on a word boundary, or at the end of the previous match.

Table 2.17 lists the boundary matchers.

Boundary Construct	Description
^	The beginning of a line
\$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Table 2.17: Boundary Constructs

The following examples demonstrate the use of boundary matchers ^ and \$. Remember that ^ matches the beginning of a line and \$ matches the end.

Enter expression: ^bat\$

Enter string to search: bat

I found the text "bat" starting at index 0 and ending at index 3.

Enter expression: ^bat\$

Enter string to search: bat

No match found.

Enter expression: \s*bat\$

Enter string to search: bat

I found the text "bat" starting at index 0 and ending at index 15.

Enter expression: ^bat\w*

Enter string to search: batblueblue

I found the text "batblueblue" starting at index 0 and ending at index 11.

The observations are as follows:

- The first match succeeds because the pattern occupies the entire input string.
- The second match fails because the input string contains extra whitespaces at the beginning.
- The third match succeeds because the expression allows for unlimited white space, followed by 'bat' present at the end of the line.
- The fourth match requires 'bat' to be present at the beginning of a line followed by an unlimited number of word characters. The match succeeds as the input string satisfies the condition.

Following are examples to check if a pattern begins and ends on a word boundary by using the boundary matcher \b on either side.

Enter expression: \bbat\b

Enter string to search: The bat is in the closet.

I found the text "bat" starting at index 4 and ending at index 7.

Enter expression: \bbat\b

Enter string to search: The batter is in the closet.

No match found.

Following are examples to match the expression on a non-word boundary by using \B:

Enter expression: \bbat\B

Enter string to search: The bat is in the closet.

No match found.

Enter expression: \bbat\B

Enter string to search: The batter is in the closet.

I found the text "bat" starting at index 4 and ending at index 7.

Following are the examples to verify the match occurring only at the end of the previous match by using \G:

Enter expression: bat

Enter string to search: bat bat

I found the text "bat" starting at index 0 and ending at index 3.

I found the text "bat" starting at index 4 and ending at index 7.

Enter expression: \Gbat

Enter string to search: bat bat

I found the text "bat" starting at index 0 and ending at index 3.

Here the second example finds only one match, because the second occurrence of 'bat' does not start at the end of the previous match.

2.11 Additional Methods of the Pattern Class

Until now, the `RegexTest` class has been used to create `Pattern` objects in their most basic form. One can also use advanced techniques such as creating patterns with flags and using embedded flag expressions. Also, one can use the additional useful methods of the `Pattern` class.

2.11.1 Creating a Pattern with Flags

The `Pattern` class provides an alternate `compile()` method that accepts a set of flags. These flags affect the way the pattern is matched. The `flags` parameter is a bit mask including any of the following public static fields:

- **`Pattern.CANON_EQ`**: Allows canonical equivalence. If this flag is specified then it will match two characters if and only if their full canonical decompositions match. For example, the expression '`a\ u030A`' will match the string '`\u00E5`' when this flag is specified. By default, matching does not consider canonical equivalence. Using this flag results in a performance penalty.
- **`Pattern.CASE_INSENSITIVE`**: Allows `case-insensitive matching`. By default, case-insensitive matching assumes that matching is done only on characters present in the US-ASCII charset. To enable Unicode-aware case-insensitive matching, one needs to specify the `UNICODE_CASE` flag along with this flag. Enabling case-insensitive matching can also be accomplished by using the embedded flag expression `(?i)`. Using this flag may result in a slight performance penalty.
- **`Pattern.COMMENTS`**: Allows `comments and whitespace in the pattern`. In this matching mode, a whitespace is ignored, and embedded comments starting with # are ignored till the end of a line. One can also use the embedded flag expression `(?x)` to enable comments.
- **`Pattern.DOTALL`**: Enables the Dotall mode. In Dotall mode, the expression '.' matches any character which can also include a line terminator. This mode does not match line terminators, by default. One can also use the embedded flag expression `(?s)` to enable the Dotall mode. The 's' is a mnemonic for 'single-line' mode.
- **`Pattern.LITERAL`**: Allows literal parsing of the pattern. When specifying this flag, then the input string specifying the pattern is treated as a sequence of literal characters. The metacharacters or escape sequences used in the input sequence will not have any special meaning. The flags `CASE_INSENSITIVE` and `UNICODE_CASE` when used in conjunction with this flag retain their matching. The other flags become superfluous. There is no embedded flag character for enabling literal parsing.

- **Pattern.MULTILINE**: Enables multiline mode. In this mode the expressions ^ and \$ matches just after or just before, a line terminator or the end of the input sequence respectively. These expressions, by default only match at the beginning and the end of the entire input sequence. Multiline mode is enabled using the embedded flag expression (?m).
- **Pattern.UNICODE_CASE**: Enables Unicode-aware case folding. When this flag is specified then case-insensitive matching, is performed in a manner consistent with the Unicode Standard. This will happen when enabled by the CASE_INSENSITIVE flag, by default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case folding is enabled using the embedded flag expression (?u). Specifying this flag may result in a performance penalty.
- **Pattern.UNIX_LINES**: Enables Unix lines mode. In this mode, only the '\n' line terminator is recognized in the behavior of ., ^, and \$. Unix lines mode is enabled using the embedded flag expression (?d).

To understand the use of flags, first modify the code of **RegexTest.java** class to invoke the alternate version of compile as shown in Code Snippet 12.

Code Snippet 12:

```
...
Pattern pattern =
Pattern.compile(console.readLine("%nEnter expression:"), Pattern.CASE_
INSENSITIVE);
...
```

Figure 2.37 shows creation of a pattern with case-insensitive matching.

```
E:\>javac RegexTest.java
E:\>java RegexTest
Enter expression: bat
Enter string to search: BaTBAt
Found the text "BaT" starting at index 0 and ending at index 3.
Found the text "BAt" starting at index 3 and ending at index 6.
Press x to exit or y to continue
Enter your choice: x
E:\>
```

Figure 2.37: Case-insensitive Matching

Note that the string literal 'bat' matches both occurrences, irrespective of case.

Now, to compile a pattern with multiple flags, use the bitwise OR operator ‘|’ to separate the flags as follows:

```
pattern = Pattern.compile("[az]$", Pattern.MULTILINE | Pattern.UNIX_LINES);
```

An int variable can also be used of specifically writing the flags as follows:

```
final int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;
Pattern pattern = Pattern.compile("aa", flags);
```

2.11.2 Embedded Flag Expressions

Embedded flag expressions can also be used to enable various flags. They are an alternative to the two-argument version of `compile()` method. They are specified in the regular expression itself. Following example uses the original `RegexTest.java` class with the embedded flag expression (?i) to enable case-insensitive matching.

```
Enter your regex: (?i)bat
```

```
Enter input string to search: BATbatBaTbaT
```

I found the text "BAT" starting at index 0 and ending at index 3.

I found the text "bat" starting at index 3 and ending at index 6.

I found the text "BaT" starting at index 6 and ending at index 9.

I found the text "baT" starting at index 9 and ending at index 12.

Once again, all matches succeed irrespective of case.

Table 2.18 lists the embedded flag expressions that correspond to `Pattern` class' publicly accessible fields.

Constant	Embedded Flag Expression
Pattern.CANON_EQ	None
Pattern.CASE_INSENSITIVE	(?i)
Pattern.COMMENTS	(?x)
Pattern.MULTILINE	(?m)
Pattern.DOTALL	(?s)
Pattern.LITERAL	None
Pattern.UNICODE_CASE	(?u)
Pattern.UNIX_LINES	(?d)

Table 2.18: Embedded Flag Expressions

2.11.3 *matches(String, CharSequence) Method*

The `Pattern` class defines the `matches()` method that allows the programmer to quickly check if a pattern is present in a given input string. Similar, to all public static methods, the `matches()` method is invoked by its class name, that is, `Pattern.matches("\\d", "1")`. In this case, the method will return true, because the digit '1' matches the regular expression '\d'.

2.11.4 *split(String) Method*

The `split()` method of `Pattern` class is used for obtaining the text that lies on either side of the pattern being matched. Consider the `SplitTest.java` class shown in Code Snippet 13.

Code Snippet 13:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitTest{
    private static final String REGEX = ":";

    private static final String DAYS = "Sun:Mon:Tue:Wed:Thu:Fri:Sat";

    public static void main(String[] args) {
        Pattern objP1 = Pattern.compile(REGEX);
        String[] days = objP1.split(DAYS);
        for(String s : days) {
            System.out.println(s);
        }
    }
}
```

In the code, the `split()` method is used to extract the words 'Sun Mon Tue Wed Thu Fri Sat' from the string '`Mon:Tue:Wed:Thu:Fri:Sat`'. Figure 2.38 shows the output of the code.

```
E:\>javac SplitTest.java
E:\>java SplitTest
Sun
Mon
Tue
Wed
Thu
Fri
Sat
```

Figure 2.38: Using the split() Method

The string literal colon (:) has been used instead of a complex regular expression.

The `split()` method can also be used to get the text that falls on either side of any regular expression. Code Snippet 14 explains the example to split a string on digits.

Code Snippet 14:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitTest{

    private static final String REGEX = "\\d";
    private static final String DAYS = "Sun1Mon2Tue3Wed4Thu5Fri6Sat";

    public static void main(String[] args) {

        Pattern objP1 = Pattern.compile(REGEX);
        String[] days = objP1.split(DAYS);
        for(String s : days) {
            System.out.println(s);
        }
    }
}
```

Figure 2.39 shows the output of the code.

```
E:\>javac SplitTest.java
E:\>java SplitTest
Sun
Mon
Tue
Wed
Thu
Fri
Sat
```

Figure 2.39: Using the split() Method with Regular Expression

The regular expression \\d is used to specify the pattern.

2.11.5 Other Useful Methods

Following are some methods that can also be used in some cases:

- **public static String quote(String s):** This method returns a literal pattern String for the specified String argument. This String produced by this method can be used to create a Pattern that would match the argument String s as if it were a literal pattern. Metacharacters or escape sequences in the input string will hold no special meaning.
- **public String toString():** Returns the String representation of this pattern.

2.12 Check Your Progress

1. The _____ method is called by the garbage collector on an object when it is identified to have no more references pointing to it.

(A)	final()	(C)	destroy()
(B)	finalize()	(D)	notify()

(A)	A	(C)	C
(B)	B	(D)	D

2. Which of the following statements about Quantifiers are true?

(A)	The greedy quantifiers are termed 'greedy' because they force the matcher to read the entire input string before attempting the first match.
(B)	The possessive quantifiers start at the beginning of the input string and then, reluctantly read one character at a time looking for a match.
(C)	The last thing that Greedy quantifiers try is the entire input string.
(D)	The possessive quantifiers always eat the entire input string, trying recursively for a match.

(A)	B,C	(C)	A,C
(B)	B,D	(D)	A,B

3. Match the following Pattern constants with the corresponding embedded flag expressions.

Constant		Embedded Flag Expressions	
a.	Pattern.CASE_INSENSITIVE	1.	(?m)
b.	Pattern.COMMENTS	2.	(?s)
c.	Pattern.MULTILINE	3.	(?i)
d.	Pattern.DOTALL	4.	(?x)

(A)	a-4, b-3, c-2, d-1	(C)	a-3, b-4, c-1, d-2
(B)	a-3, b-1, c-4, d-1	(D)	a-2, b-4, c-1, d-3

4. Which of the following input strings will find a match for the regular expression `b[^at]\s?`

(A)	bar	(C)	beg
(B)	bet	(D)	bat

(A)	B,C	(C)	A,C
(B)	B,D	(D)	A,B

5. Consider the following code.

```
import java.util.StringTokenizer;

public class Tokenizer{

    public static void main(String[] args) {

        StringTokenizer st = new StringTokenizer("Java,is,object:
oriented", ":");

        while (st.hasMoreTokens())
            System.out.println(st.nextToken());
    }
}
```

What will be the output of the code?

(A)	Java,is,object,oriented	(C)	Java,is,object oriented
(B)	Java is object oriented	(D)	Java,is Object-oriented

6. Match the following garbage collection approaches with the corresponding description.

GC Approach		Description	
a.	Serial	1.	Copies or evacuates live objects to a different memory area.
b.	Concurrent	2.	Works on only one thing at a time.
c.	Non-compacting	3.	Executes one or more garbage collection tasks simultaneously with the execution of the application.
d.	Copying	4.	Releases the space utilized by garbage objects in-place.

(A)	a-4, b-3, c-2, d-1	(C)	a-3, b-4, c-1, d-2
(B)	a-3, b-1, c-4, d-1	(D)	a-2, b-3, c-4, d-1

2.12.1 Answers

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

Summary

- The java.lang package provides classes that are fundamental for the creation of a Java program.
- Garbage collection solves the problem of memory leak because it automatically frees all memory that is no longer referenced.
- In the stop-the-world garbage collection approach, during garbage collection, application execution is completely suspended.
- The finalize() method is called by the garbage collector on an object when it is identified to have no more references pointing to it.
- Object class is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of the Object class.
- StringBuilder objects are like String objects, except that they are mutable. Internally, the runtime treats these objects like variable-length arrays containing a sequence of characters.
- The StringTokenizer class belongs to the java.util package and is used to break a string into tokens.
- Any regular expression that is specified as a string must first be compiled into an instance of the Pattern class.
- A Matcher object is the engine that performs the match operations on a character sequence by interpreting a Pattern.
- Intersection is used to create a single character class that matches only the characters which are common to all of its nested classes.
- The greedy quantifiers are termed ‘greedy’ because they force the matcher to read the entire input string before attempting the first match.

Tutor Contributed Value-ads For Students



For further reading, logon to

Session 3

Collections API

Welcome to the Session, **Collections API**.

This session will give you an insight to the Collections Application Programming Interface (API). Collections API is a unified architecture for representing and manipulating collections.

In this Session, you will learn to:

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



3.1 Introduction

The Collections Framework consists of collection interfaces which are primary means by collections are manipulated. They also have wrapper and general purpose implementations. Adapter implementation helps to adapt one collection over other. Besides these there are convenience implementations and legacy implementations.

3.2 java.util Package

The `java.util` package contains the definition of a number of useful classes providing a broad range of functionality. The package mainly contains collection classes that are useful for working with groups of objects. The package also contains the definition of classes that provides date and time facilities and many other utilities, such as calendar and dictionary. It also contains a list of classes and interfaces to manage a collection of data in memory. Figure 3.1 displays some of the classes present in `java.util` package.

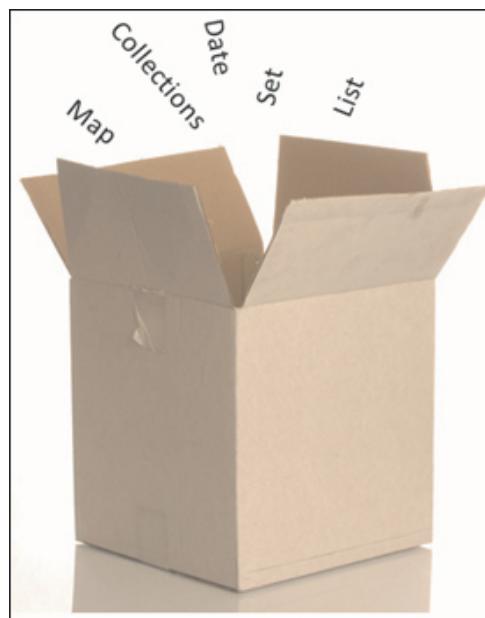


Figure 3.1: `java.util` Package

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

→ Date Class, its Constructors, and Methods

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

Constructors of `Date` class are listed in table 3.1.

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

Table 3.1: Constructors of the Date Class

→ Calendar Class, Its Constructors and Methods

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

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

→ Random Class

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

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

3.2.1 Collections Framework

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

The Collections Framework represents and manipulates collections. It includes the following:

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

Note - The same method can be applied on different implementations of the appropriate collection interface.

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

In Java, the Collections Framework provides a set of interfaces and classes for storing and manipulating groups of objects in a standardized way. The Collections Framework was developed with the following objectives in mind:

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

3.2.2 Collection Interface

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

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

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

JDK provides implementation of specific sub interfaces such as `Set` and `List` which are used to pass collections and manipulate them wherever it is required to be general.

The `Collection` interface is extended by the following sub interfaces:

- `Set`
- `List`
- `Queue`

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

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

Figure 3.2 displays the Collection interface.

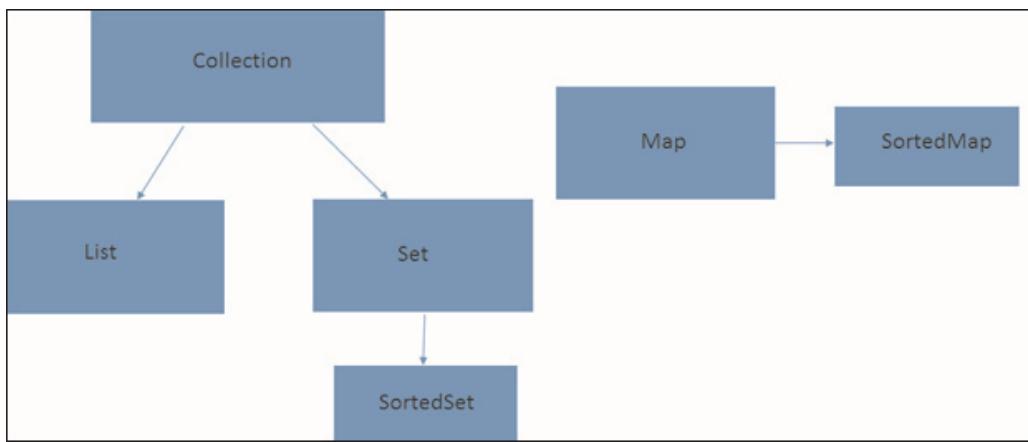


Figure 3.2: Collection Interface

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

3.2.3 Methods of Collection Interface

A Collection interface helps to pass collection of objects with maximum generality. A collection class holds a number of items in the data structure and provides various operations to manipulate the contents of the collection, such as add item, remove item, and find the number of elements.

The interface includes the following methods:

- `size`, `isEmpty`: Use these to inform about the number of elements that exist in the collection.
- `contains`: Use this to check if a given object is in the collection.
- `add`, `remove`: Use these to add and remove an element from the collection.
- `iterator`: Use this to provide an iterator over the collection.

Some of the other important methods supported by the `Collection` interface are listed in table 3.2.

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

Table 3.2: Methods Supported by Collection Interface

3.2.4 Traversing Collections

The following section describes the two ways to traverse collections:

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

Code Snippet 1:

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

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

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

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

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

The following describes the `Iterator` interface:

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

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

3.2.5 Bulk Operations

Bulk operations perform shorthand operations on an entire collection using the basic operations. Table 3.3 describes the methods for bulk operations.

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

Table 3.3: Bulk Operation Methods

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

3.3 Lists

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

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

3.3.1 Methods of List Interface

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

→ **`add(int index, E element)`**

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

Syntax:

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

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

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

Syntax:

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

→ **get (int index)**

The method retrieves element from the specified index position.

Syntax:

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

→ **set (int index, E element):**

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

Syntax:

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

→ **remove(int index):**

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

Syntax:

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

→ **subList(int start, int end):**

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

Syntax:

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

Some of the other methods supported by the List interface are listed in table 3.4.

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

Table 3.4: Methods of List Interface

3.3.2 ArrayList Class

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

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

The ArrayList class includes all elements, including null. In addition to implementing the methods of the List interface, this class provides methods to change the size of the array that is used internally to store the list.

Each ArrayList instance includes a capacity that represents the size of the array. A capacity stores the elements in the list and grows automatically as elements are added to an ArrayList.

ArrayList class is best suited for random access without inserting or removing elements from any place other than the end.

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

ArrayList class is best suited for random access without inserting or removing elements from any place other than the end.

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

→ **ArrayList()**

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

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

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

→ **ArrayList(int initialCapacity)**

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

Code Snippet 2 displays the creation of an instance of the `ArrayList` class.

Code Snippet 2:

```
...
List<String> listObj = new ArrayList<String> ();
System.out.println("The size is : " + listObj.size());
for (int ctr=1; ctr <= 10; ctr++)
{
    listObj.add("Value is : " + new Integer(ctr));
}
...
```

In Code Snippet 2, an `ArrayList` object is created and the first ten numbers are added to the list as objects. The primitive `int` data type is wrapped by the respective wrapper class.

3.3.3 Methods of `ArrayList` Class

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

→ **`add(E obj)`**

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

Syntax:

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

→ **`trimToSize()`**

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

Syntax:

```
public void trimToSize()
```

→ **`ensureCapacity(int minCap)`**

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

Syntax:

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

→ **clear()**

The method removes all the elements from this list.

Syntax:

```
public void clear()
```

→ **contains(Object obj)**

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

Syntax:

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

→ **size()**

The method returns the number of elements in this list.

Syntax:

```
public int size()
```

Code Snippet 3 displays the use of `ArrayList` class.

Code Snippet 3:

```
...
List<String> listObj = new ArrayList<String> ();
System.out.println("The size is : " + listObj.size());
for (int ctr=1; ctr <= 10; ctr++)
{
    listObj.add("Value is : " + new Integer(ctr));
}
listObj.set(5, "Hello World");
System.out.println("Value is: " +(String)listObj.get(5));
...
```

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

3.3.4 Vector Class

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

The elements in the Vector can be accessed using an integer index.

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

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

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

The constructors of this class are as follows:

→ **Vector()**

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

Syntax:

```
public Vector()
```

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

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

Syntax:

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

→ **Vector(int initCapacity)**

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

Syntax:

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

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

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

Syntax:

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

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

Code Snippet 4:

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

3.3.5 Methods of `Vector` Class

`Vector` class defines three protected members. They are as follows:

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

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

→ **`addElement(E obj)`**

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

Syntax:

```
public void addElement(E obj)
```

→ **`capacity()`**

The method returns the present capacity of the vector.

Syntax:

```
public int capacity()
```

→ **`toArray()`**

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

Syntax:

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

→ **elementAt(int pos)**

The method retrieves the object stored at the specified location.

Syntax:

```
public Object elementAt(int pos)
```

→ **removeElement(Object obj)**

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

Syntax:

```
public boolean removeElement(Object obj)
```

→ **clear()**

The method removes all the elements of the vector.

Syntax:

```
public void clear()
```

Code Snippet 5 displays the use of the `Vector` class.

Code Snippet 5:

```
...
Vector<Object> vecObj = new Vector<Object>();
vecObj.addElement(new Integer(5));
vecObj.addElement(new Integer(7));
vecObj.addElement(new Integer(45));
vecObj.addElement(new Float(9.95));
vecObj.addElement(new Float(6.085));
System.out.println("The value is: " + (Object) vecObj.elementAt(3));
...
```

In the code, a `Vector` is created and initialized with `int` and `float` values after converting them to their respective object types. Finally, the value at the third position is retrieved and displayed.

3.3.6 *LinkedList Class*

An ordered collection is called a list. It is also called a **sequence**. Following are the features of a list:

- The user has the control over each element in the list.

- The user can access and search for elements in the list using integer index.
- Duplicate elements can exist in a list.
- Multiple null elements can exist.
- Integer index can be used to access elements and search for them in the list.
- Similar to Java arrays, lists are zero based.
- The `List` interface includes four methods for indexed access to list elements.

Note - For certain implementations, such operations can execute in time proportional to the index value.

The `List` interface includes additional declarations using the `iterator`, `remove`, `add`, `hashCode`, and `equals` methods. The `equals()` method helps to check the logical equality of two `List` objects. Two `List` objects are equal if they contain the same elements in the same order. It can include declarations for other inherited methods also.

The `List` interface provides `ListIterator`, which is a special iterator. This is used to insert and replace an element. This is also used for bidirectional access along with the normal operations. A method is used to get a list iterator, which starts at a particular position in the list.

There are two methods in the `List` interface to search for a specified object. However, these methods can execute costly linear searches.

The `List` interface also includes two methods to insert and remove multiple elements at an arbitrary point in the list.

`LinkedList` class implements the `List` interface. An array stores objects in consecutive memory locations, whereas a linked list stores object as a separate link. It provides a linked-list data structure. A linked-list is a list of objects having a link to the next object. There is usually a data element followed by an address element that contains the address of the next element in the list in a sequence. Each such item is referred as a node.

Linked lists allow insertion and removal of nodes at any position in the list, but do not allow random access. There are several different types of linked lists - singly-linked lists, doubly-linked lists, and circularly-linked lists.

Java provides the `LinkedList` class in the `java.util` package to implement linked lists.

- `LinkedList()`

The `LinkedList()` constructor creates an empty linked list.

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

The `LinkedList(Collection <? extends E>c)` constructor creates a linked list, which contains the elements of a specified collection, in the order they are returned by the collection's iterator.

Figure 3.3 displays a linked list.

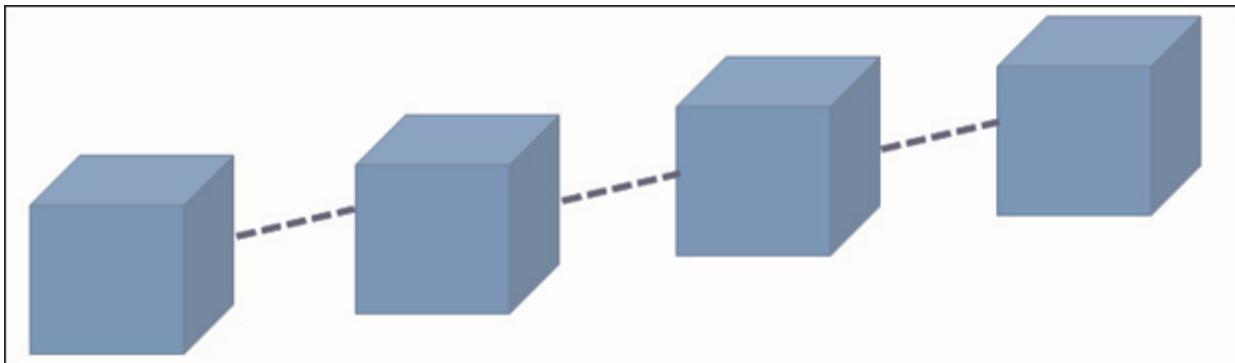


Figure 3.3: LinkedList

Code Snippet 6 displays the creation of an instance of the `LinkedList` class.

Code Snippet 6:

```
...
LinkedList<String> lisObj = new LinkedList<List>();
...
```

3.3.7 Methods of `LinkedList` Class

In addition to the methods defined by the `List` interface, the other methods of the `LinkedList` class are as follows:

→ **addFirst(E obj)**

This method adds the given object at the beginning of the list.

Syntax:

```
public void addFirst(E obj)
```

→ **addLast(E obj)**

This method appends the given object at the end of the list.

Syntax:

```
public Object addLast(E obj)
```

→ **getFirst()**

The method retrieves the first element from the list.

Syntax:

```
public E getFirst()
```

→ **getLast()**

The method retrieves the last element from the list.

Syntax:

```
public E getLast()
```

→ **removeFirst()**

The method removes and returns the first element from the list.

Syntax:

```
public E removeFirst()
```

→ **removeLast()**

The method removes and returns the last element from the list.

Syntax:

```
public E removeLast()
```

Code Snippet 7 displays the use of the methods of the `LinkedList` class.

Code Snippet 7:

```
...
LinkedList<String> lisObj = new LinkedList<String>();
lisObj.add("John");
lisObj.add("Mary");
lisObj.add("Jack");
lisObj.add("Elvis");
lisObj.add("Martin");
System.out.println("Original content of the list: " + lisObj);
lisObj.removeFirst();
System.out.println("After removing content of the list: " + lisObj);
...
```

In the code, a `LinkedList` object has been created and instantiated with string values. Next, the original content of the list is displayed. After removing an object from the list the content of the list is again displayed.

3.3.8 AutoBoxing and Unboxing

The autoboxing and unboxing feature automates the process of using primitive value into a collection. Note that collections hold only object references. So, primitive values, such as `int` from `Integer`, have to be boxed into the appropriate wrapper class. If an `int` value is required, the `Integer` value must be unbox using the `intValue()` method. The autoboxing and unboxing feature helps to reduce the clutter in the code.

3.4 Sets

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

3.4.1 Set Interface

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

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

→ `HashSet`

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

→ `TreeSet`

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

→ `LinkedHashSet`

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

3.4.2 Comparison of Set with List

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

3.4.3 Methods of Set Interface

Set interface is best suited for carrying out bulk operations. The bulk operation methods supported by the Set interface are as follows:

- **containsAll(Collection<?> obj)**

This method returns true if the invoking set object contains all the elements of the specified collection.

Syntax:

```
public boolean containsAll(Collection<?> obj)
```

- **addAll(Collection<? extends E> obj)**

This method adds all the elements of the specified collection to the invoking set object.

Syntax:

```
public boolean addAll(Collection<? extends E> obj)
```

- **retainAll(Collection<?> obj)**

This method retains in the invoking set only those elements which are contained in the specified collection.

Syntax:

```
public boolean retainAll(Collection<?> obj)
```

- **removeAll(Collection<?> obj)**

This method removes all the elements from the invoking set that are contained in the specified collection.

Syntax:

```
public boolean removeAll(Collection<?> obj)
```

Note - Two Set instances are considered equal when they contain the same elements.

3.4.4 SortedSet Interface

The SortedSet interface extends the Set interface and its iterator traverses its elements in the ascending order. Elements can be ordered by natural ordering, or by using a Comparator that a user can provide while creating a sorted set.

SortedSet is used to create sorted lists of non-duplicate object references. The **ordering of a sorted set should be consistent with equals() method**. This is because the Set interface is specified in terms of the equals operation.

A sorted set performs all element comparisons using the `compareTo()` or `compare()` method. So, two elements considered equal by this method carry the same status for the sorted set.

Note - If the ordering is inconsistent with `equals`, a sorted set results in normal behavior but does not follow the general guidelines of the `Set` interface.

Typically, sorted set implementation classes provide the following standard constructors:

- **No argument (`void`) constructor:** This creates an empty sorted set which is sorted according to the natural ordering of its elements.
- **Single argument of type `Comparator` constructor:** This creates an empty sorted set sorted according to the defined comparator.
- **Single argument of type `Collection` constructor:** This creates a new sorted set with the elements that exists as its argument, sorted according to the natural ordering of the elements.
- **Single argument of type `SortedSet` constructor:** This creates a new sorted set with the elements and the same ordering that exists in the input sorted set.

The `SortedSet` defines some methods in addition to the methods inherited from the `Set` interface so as to make set processing convenient.

Some of the methods in this interface are as follows:

→ **`first()`**

This method returns the first or lowest element in the sorted set.

Syntax:

```
public E first()
```

→ **`last()`**

This method returns the last or highest element in the sorted set.

Syntax:

```
public E last()
```

→ **`headSet (E endElement)`**

This method returns a `SortedSet` instance containing all the elements from the beginning of the set up to the specified element, `endElement`, but not including the end element.

Syntax:

```
public SortedSet<E> headSet (E endElement)
```

→ **subSet (E startElement, E endElement)**

This method returns a `SortedSet` instance that includes the elements between `startElement` and `endElement-1`.

Syntax:

```
public SortedSet<E> subSet(E startElement, E endElement)
```

→ **tailSet (E fromElement)**

This method returns a `SortedSet` instance containing those elements greater than or equal to `fromElement` that are contained in the invoking sorted set.

Syntax:

```
public SortedSet<E> tailSet(E fromElement)
```

Note - Several methods throw `NoSuchElementException` when there are no elements in the set. When an object is incompatible with the elements in a set, it throws `ClassCastException` and `NullPointerException` is thrown if a null object is used.

3.4.5 HashSet Class

`HashSet` class implements the `Set` interface. A `HashSet` creates a collection that makes use of a hashtable for data storage. A hashtable is a data structure that stores information by mapping the key of each data element into an array position or index. A key is an identifier used to find or search a value in a hashtable. The specific meaning of keys in any hashtable is totally dependent on how the table is used and what data it contains. The transformation of a key into its hash code is done automatically. User's code cannot directly index the hashtable.

This `HashSet` class allows `null` element. However, there is no guarantee on the iteration order of the set and that the order remains constant over time.

This `HashSet` class provides constant time performance for the basic operations. For iteration, time should be proportional to the sum of the `HashSet` instance's size and the capacity of the backing `HashMap` instance.

Note - For good iteration performance, do not set the initial capacity too high or the load factor too low.

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

→ **HashSet()**

The constructor creates a default hash set. The default initial capacity and load factor are 16 and 0.75, respectively.

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

The constructor creates a hash set based on the elements given in the specified collection.

→ **HashSet(int size)**

The constructor creates a hash set with given size and with the default load factor of 0.75.

→ **HashSet(int size, float fillRatio)**

The constructor creates a hash set with given size and fill ratio. A fill ratio determines the capacity of the set before it is resized upward.

Code Snippet 8 displays the creation of an instance of HashSet class.

Code Snippet 8:

```
...
Set<String> words = new HashSet<String>();
...
```

In the code, an instance of HashSet class is created.

3.4.6 *LinkedHashSet Class*

The LinkedHashSet class creates a list of elements and maintains the order of the elements added to the Set. It includes hash table and linked list implementation of the Set interface. It differs from HashSet in the iteration order. Here, a double-linked list is maintained that runs through all of its entries.

This linked list defines the iteration ordering, which is the order in which elements were inserted into the set.

This class includes the following features:

- It provides all of the optional Set operations.
- It permits null elements.
- It provides constant-time performance for the basic operations such as add and remove.

Note - The performance of the LinkedHashSet class might not be good as HashSet because of the cost of maintaining the linked list.

For iteration over a LinkedHashSet, time should be proportional to the size of the set. This is regardless of its capacity. Iteration over a HashSet can be more expensive because time is required to be proportional to its capacity.

Initial capacity and load factor affect the performance of a linked hash set. Iteration times for `LinkedHashSet` are unaffected by capacity.

This class has no additional methods and has constructors that take identical parameters as the `HashSet` class.

The constructors of this class are as follows:

→ **`LinkedHashSet()`**

The constructor creates a default linked hash set. The default initial capacity and load factor are 16 and 0.75 respectively.

→ **`LinkedHashSet(Collection<? extends E> c)`**

The constructor constructs a new linked hash set with the same elements as the specified collection.

→ **`LinkedHashSet(int initial capacity)`**

The constructor constructs a new, empty linked hash set with the specified initial capacity. Its signature is as follows:

```
LinkedHashSet(int initialCapacity, float loadFactor)
```

The constructor constructs a new empty linked hash set with the initial capacity and load factor as specified.

3.4.7 TreeSet Class

`TreeSet` class implements the `NavigableSet` interface and uses a tree structure for data storage. The elements can be ordered by natural ordering, or by using a `Comparator` provided at the time of set creation.

Objects are stored in ascending order and therefore accessing and retrieving an object is much faster. `TreeSet` will be used when elements needs to be extracted quickly from the collection in a sorted manner.

→ **`TreeSet()`**

The constructor creates an empty tree set with the elements sorted in ascending order.

→ **`TreeSet(Collection<? extends E> c)`**

The constructor creates a new tree set containing the elements of the specified collection, sorted according to the elements order.

→ **`TreeSet(Comparator<? super E> c)`**

The constructor creates a new, empty tree set and will be sorted according to the specified comparator `c`.

→ **TreeSet (SortedSet<E> s)**

The constructor creates a new tree set containing the elements of the specified `SortedSet` in the same order.

Code Snippet 9 shows the creation of `TreeSet` object.

Code Snippet 9:

```
...
TreeSet tsObj = new TreeSet();
...
```

In the code, an instance of the `TreeSet` is created.

3.5 Maps

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

3.5.1 Map Interface

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

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

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

→ **HashMap**

→ **TreeMap**

→ **LinkedHashMap**

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

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

→ **put(K key, V value)**

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

Syntax:

```
V put(Object key, Object value)
```

→ **get(Object key)**

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

Syntax:

```
V get(Object key)
```

→ **containsKey(Object Key)**

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

Syntax:

```
public boolean containsKey(Object key)
```

→ **containsValue(Object Value)**

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

Syntax:

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

→ **size()**

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

Syntax:

```
public int size()
```

→ **values()**

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

Syntax:

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

3.5.2 HashMap Class

The `HashMap` class implements the `Map` interface and inherits all its methods. An instance of `HashMap` has two parameters: initial capacity and load factor. Initial capacity determines the number of objects that can be added to the `HashMap` at the time of the hashtable creation. The load factor determines how full the hashtable can get before its capacity is automatically increased.

The `HashMap` is very similar to the `Hashtable` with two main differences:

- The `HashMap` is not synchronized making access faster.
- The `HashMap` allows null values to be used as values or keys, which are disallowed in the `Hashtable` implementation. `HashMap` class does not guarantee the order of the map and it does not guarantee that the order will remain constant over time.

The constructors of this class are as follows:

→ **`HashMap()`**

The constructor constructs an empty `HashMap` with the default initial capacity and load factor of 17 and 0.75 respectively.

→ **`HashMap(int initialCapacity)`**

The constructor constructs an empty `HashMap` with the specified initial capacity and default load factor of 0.75.

→ **`HashMap(int initialCapacity, float loadFactor)`**

The constructor constructs an empty `HashMap` with the specified initial capacity and load factor.

→ **`HashMap(Map<? extends K,? extends V> m)`**

The constructor constructs a `HashMap` similar to the specified map `m`.

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

Code Snippet 10:

```
...
class EmployeeData
{
    public EmployeeData (String nm)
    {
        name = nm;
        salary = 5600;
    }
    public String toString()
    {
        return "[name=" + name + ", salary=" + salary + "]";
    }
}
```

```

public String toString()
{
    return "[name=" + name + ", salary=" + salary + "]";
}

...
}

public class MapTest
{
    public static void main(String[] args)
    {
        Map<String, EmployeeData> staffObj = new HashMap<String,
EmployeeData>();
        staffObj.put("101", new EmployeeData("Anna John"));
        staffObj.put("102", new EmployeeData("Harry Hacker"));
        staffObj.put("103", new EmployeeData("Joby Martin"));
        System.out.println(staffObj);
        staffObj.remove("103");
        staffObj.put("106", new EmployeeData("Joby Martin"));
        System.out.println(staffObj.get("106"));
        System.out.println(staffObj);
        ...
    }
}

```

In the code, an **EmployeeData** class and a **MapTest** is created. In the **MapTest** class an instance of **HashMap** class is created. In the **HashMap** object, data of the **EmployeeData** class is added, removed, and retrieved.

3.5.3 Hashtable Class

The **Hashtable** class implements the **Map** interface but stores elements as a key/value pairs in the hashtable. While using a **Hashtable**, **a key is specified to which a value is linked**. The key is than hashed and then the hash code is used as an index at which the value is stored. The class inherits all the methods of the **Map** interface. **To retrieve and store objects from a hashtable successfully, the objects used as keys must implement the `hashCode()` and `equals()` method.**

The constructors of this class are as follows:

→ **Hashtable()**

The constructor constructs a new, empty hashtable.

→ **Hashtable (int initCap)**

The constructor constructs a new, empty hashtable with the specified initial capacity.

→ **Hashtable (int intCap, float fillRatio)**

The constructor constructs a new, empty hashtable with the specified initial capacity and load factor.

→ **Hashtable (Map<? extends K,? extends V> m)**

The constructor constructs a new hashtable containing the same entries as the given Map.

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

Code Snippet 11:

```
...
Hashtable<String, String> bookHash = new Hashtable<String, String>();
bookHash.put("115-355N", "A Guide to Advanced Java");
bookHash.put("116-455A", "Learn Java by Example");
bookHash.put("116-466B", "Introduction to Solaris");
String str= (String) bookHash.get("116-455A");
System.out.println("Detail of a book "+str);
System.out.println("Is table empty "+bookHash.isEmpty());
System.out.println("Does table contains key? "+bookHash.containsKey("116-466B"));
Enumeration name=bookHash.keys();
while (name.hasMoreElements())
{
    String bkCode= (String)name.nextElement();
    System.out.println(bkCode+": "+(String)bookHash.get(bkCode));
}
...
```

The code demonstrates the use of `Hashtable` and some of its methods.

The `put()` and `get()` method is used to insert and retrieve values from the hashtable. The `isEmpty()` method checks whether the hashtable is empty and the `containsKey()` method is used to check whether a particular key exists or not.

To obtain all the book details, an `Enumeration` object is used. The keys are retrieved by using the `nextElement()` method of the `Enumeration` interface and later using the keys the value associated with each key are retrieved.

Note - `Enumeration` interface returns a series of elements and calling the `nextElement()` method returns the element present in the series one at a time.

3.5.4 TreeMap Class

The `TreeMap` class implements the `NavigableMap` interface but stores elements in a tree structure. The `TreeMap` returns keys in sorted order. If there is no need to retrieve `Map` elements sorted by key, then the `HashMap` would be a more practical structure to use.

The constructors of this class are as follows:

→ **`TreeMap()`**

The constructor constructs a new empty tree map.

→ **`TreeMap(Comparator<? super K> c)`**

The constructor constructs a tree map, where the keys are sorted according to the given comparator.

→ **`TreeMap(Map<? extends K,? extends V> m)`**

The constructor constructs a new tree map containing the same entries as the given map `m`. It orders according to the natural ordering of its key.

→ **`TreeMap(SortedMap<K,? extends V> m)`**

The constructor constructs a new tree map containing the same entries as the given `SortedMap` and uses the same comparator as the given `SortedMap`.

3.5.5 Methods of TreeMap Class

The important methods of the `TreeMap` class are as follows:

→ **`firstKey()`**

The method returns the first key in this sorted map.

Syntax:

```
public K firstKey()
```

→ lastKey()

The method returns the last key in this sorted map.

Syntax:

```
public K lastKey()
```

→ headMap(K toKey)

The method returns a portion of the map whose value is less than the value specified by the variable, `toKey`.

Syntax:

```
public SortedMap<K,V> headMap(K toKey)
```

→ tailMap(K fromKey)

The method returns a portion of this invoking map whose keys is greater than or equal to the value specified in the variable, `fromKey`.

Syntax:

```
public SortedMap<K,V> tailMap(K fromKey)
```

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

Code Snippet 12:

```
...
TreeMap<String, EmployeeData> staffObj = new TreeMap<String, EmployeeData>();
staffObj.put("101", new EmployeeData("Anna John"));
staffObj.put("102", new EmployeeData("Harry Hacker"));
staffObj.put("103", new EmployeeData("Joby Martin"));
System.out.println(staffObj);
staffObj.remove("103");
staffObj.put("104", new EmployeeData("John Luther"));
System.out.println(staffObj.get("104"));
Object firstKey=staffObj.firstKey();
```

```
System.out.println(firstKey.toString());
System.out.println((String)staffObj.firstKey());
System.out.println((String)(staffObj.lastKey()));
...

```

Code Snippet 12 demonstrates the use of `TreeMap` class. Data is added in the `TreeMap` and retrieved. The function `firstKey()` and `lastKey()` is used to obtain the first and the last key.

3.5.6 *LinkedHashMap Class*

`LinkedHashMap` class implements the concept of hashtable and the linked list in the `Map` interface. A `LinkedHashMap` maintains the values in the order they were inserted, so that the key/values will be returned in the same order that they were added to this `Map`.

The constructor of this class are as follows:

→ **`LinkedHashMap()`**

The constructor creates an empty `LinkedHashMap` with the default capacity and load factor of 16 and 0.75 respectively.

→ **`LinkedHashMap(int initialCapacity)`**

The constructor creates an empty `LinkedHashMap` with the user-defined initial capacity and a default load factor of 0.75.

→ **`LinkedHashMap(int initialCapacity, float loadFactor)`**

The constructor creates an empty `LinkedHashMap` with the user-defined initial capacity and load factor.

→ **`LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`**

The constructor creates an empty `LinkedHashMap` with the user-defined initial capacity, load factor, and ordering mode. The variable `accessOrder` defines the ordering mode. A boolean value of true specifies that the ordering mode is based on access-order and false specifies that ordering mode is based on insertion-order.

→ **`LinkedHashMap(Map<? extends K,? extends V> m)`**

The constructor creates a `LinkedHashMap` with the same mappings as the specified map and whose ordering mode is based on insertion-mode.

3.5.7 Methods of LinkedHashMap Class

The important methods in `LinkedHashMap` class are as follows:

→ **`clear()`**

The method removes all mappings from the invoking map.

Syntax:

```
public void clear()
```

→ **`containsValue(Object value)`**

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

Syntax:

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

→ **`get(Object key)`**

The method returns the value to which the key is mapped.

Syntax:

```
public V get(Object key)
```

→ **`removeEldestEntry(Map.Entry<K,V> eldest)`**

The method returns true if the map should remove its eldest key.

Syntax:

```
protected boolean removeEldestEntry( Map.Entry<K, V> eldest )
```

3.6 Stack and Queues

In the `Stack` class, the stack of objects results in a **last-in-first-out** (LIFO) behavior. It extends the `Vector` class to consider a vector as a stack.

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

→ **`empty()`:** This tests if the stack is empty and returns a boolean value of true and false.

→ **`peek()`:** This views the object at the top of the stack without removing it from the stack.

→ **`pop()`:** This removes the object at the top of this stack and returns that object as the value of the function.

- `push(E item)`: This pushes an item on the top of the stack.
- `int search(Object o)`: This returns the 1-based position where an object is on the stack.

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

3.6.1 Queue Interface

A `Queue` is a collection for holding elements that needs to be processed. In `Queue`, the elements are normally ordered in First-In-First-Out (FIFO) order. The priority queue orders the element according to their values. A queue can be arranged in other orders too. Every `Queue` implementation defines ordering properties. In a FIFO queue, new elements are inserted at the end of the queue. LIFO queues or stacks order the elements in LIFO pattern. However, in any form of ordering, a call to the `poll()` method removes the head of the queue.

Queues support the standard collection methods and also provide additional methods to add, remove, and review queue elements. A queue helps track asynchronous message requests while a stack helps traverse a directory tree structure.

3.6.2 Deque Interface

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

Note - `Deque` is pronounced as deck.

Usually, `Deque` implementations have no restrictions on the number of elements to include. However, it does support capacity-restricted deques.

The methods are inherited from the `Queue` interface. A deque when used as a queue results in FIFO behavior. Elements are added at the end of the deque and removed from the start.

A deque when used as LIFO stack should be used in preference with the legacy `Stack` class. A deque when used as a stack has their elements pushed and popped from the beginning of the deque. Stack methods are equivalent to `Deque` methods.

The `Deque` interface and its implementations when used with the `Stack` class provides a consistent set of LIFO stack operations. Code Snippet 13 displays this.

Code Snippet 13:

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

3.6.3 Methods Supported by the Queue Class

Some of the important methods supported by this class are as follows:

→ **poll()**

The method returns the value of the head of the queue and removes the head from the queue. The method returns `null` if the queue is empty.

Syntax:

```
E poll()
```

→ **peek()**

The method returns the value of the head of the queue but does not remove the head from the queue. The method returns `null` if the queue is empty.

Syntax:

```
E peek()
```

→ **remove()**

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

Syntax:

```
E remove()
```

→ **offer(E obj)**

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

Syntax:

```
public boolean offer(E obj)
```

→ **element()**

The method returns the value of the head of the queue but does not remove the head from the queue. The method throws an exception if the queue is empty.

Syntax:

```
E element()
```

Each of the described methods can throw an exception if the operation fails or return a `null` or `false` value.

3.6.4 PriorityQueue Class

Priority queues are similar to queues but the elements are **not arranged in FIFO structure**. They are arranged in a **user-defined manner**. The elements are ordered either by natural ordering or according to a comparator. A priority queue neither allows adding of non-comparable objects nor allows null elements. A priority queue is unbound and allows the queue to grow in capacity.

When the elements are added to a priority queue, its capacity grows automatically.

The constructors of this class are as follows:

→ **PriorityQueue()**

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

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

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

→ **PriorityQueue(int initialCapacity)**

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

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

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

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

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

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

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

3.6.5 Methods of PriorityQueue Class

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

The other methods supported by the `PriorityQueue` class are as follows:

→ **`add(E e)`**

The method adds the specific element to the priority queue and returns a boolean value.

→ **`clear()`**

The method removes all elements from the priority queue.

→ **`comparator()`**

The method returns the comparator used to order this collection. The method will return `null` if this collection is sorted according to its elements natural ordering.

→ **`contains (Object o)`**

The method returns a boolean value of `true` if the queue contains the specified element.

→ **`iterator()`**

The method returns an iterator over the elements in the queue.

→ **`toArray()`**

The method returns an array of objects containing all of the elements in the queue.

Code Snippet 14 displays the use of the `PriorityQueue` class.

Code Snippet 14:

```
...
PriorityQueue<String> queue = new PriorityQueue<String>();
queue.offer("New York");
queue.offer("Kansas");
queue.offer("California");
queue.offer("Alabama");
System.out.println("1. " + queue.poll()); // removes
System.out.println("2. " + queue.poll()); // removes
System.out.println("3. " + queue.peek());
System.out.println("4. " + queue.peek());
System.out.println("5. " + queue.remove()); // removes
```

```
System.out.println("6. " + queue.remove()); // removes
System.out.println("7. " + queue.peek());
System.out.println("8. " + queue.element()); // Throws Exception
...

```

Output:

```
1. Alabama
2. California
3. Kansas
4. Kansas
5. Kansas
6. New York
7. null
Exception in thread "main" java.util.NoSuchElementException
at java.util.AbstractQueue.element (Unknown Source)
at QueueTest.main (QueueTest.java:24)
```

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

3.6.6 Arrays Class

`Arrays` class provides a number of methods for working with arrays such as searching, sorting, and comparing arrays. The class has a static factory method that allows the array to be viewed as lists. The methods of this class throw an exception, `NullPointerException` if the array reference is `null`.

Each of the methods that are listed has an overloaded version that differs according to the type of the array or array arguments. Some of the important methods of this class are as follows:

→ **`equals(<type> arrObj1, <type> arrObj2)`**

The method compares two specified arrays of the same type for equality. The method returns true if each array holds the same number of elements and each element in the first array is equals to the corresponding value in the second array. There is one method of this type for each primitive data type and for `Object`.

Syntax:

```
public static boolean equals(byte[] a, byte[] b)
```

→ `fill(<type>[] array, <type> value)`

The method initializes an array by assigning the specified value to all elements in the array. There is one method of this type for each primitive data type and for `Object`.

Syntax:

```
public static void fill(boolean[] a, boolean v)
```

→ `fill (type[] array, int fromIndex, int toIndex, type value)`

The method initializes the elements in an array by assigning the specified value between the given indices.

Syntax:

```
public static void fill(boolean[] a, int fromIndex, int toIndex, boolean v)
```

→ `sort(<type>[] array)`

The method sorts the array in ascending order. There is one method of this type for each primitive data type except for `boolean` data types.

Syntax:

```
public static void sort(byte[] a)
```

→ `sort(<type> [] array, int startIndex, int endIndex)`

The method sorts the elements in the array between the given indices. There is one method of this type for each primitive data type.

Syntax:

```
void sort (type[] array, int startIndex, int endIndex)
```

→ `toString(<type>[] array)`

The method returns a string representation of the contents of an array. There is one method of this type for each primitive data type.

Syntax:

```
public String toString(type[] array)
```

3.7 Sorting Collections

Collection API provides the following two interfaces for ordering interfaces:

- **Comparable:** The Comparable interface imposes a total ordering on the objects of each class which implements it. Lists of objects implementing this interface are automatically sorted. It is sorted using `Collection.sort` or `Arrays.sort` method. sort theo 1 category

Note - Ordering is called the class's natural ordering. The `compareTo()` method for the class is referred as the natural comparison method.

Such objects can be used as keys in a sorted map or as elements in a comparator.

Consider the following for the natural ordering for a class C:

- If `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every e1 and e2 of class C, the natural ordering is consistent with equals.
- null is not an instance of any class.
- `e.compareTo(null)` throws a `NullPointerException` even when `e.equals(null)` returns false.

When sorted sets and maps without explicit comparators are used with elements or keys whose natural ordering is inconsistent with equals, this results in unusual behavior. Therefore, it is recommended that natural orderings should be consistent with equals.

- **Comparator:** This interface provides multiple sorting options and imposes a total ordering on some collection of objects. The use of comparators provides the following:
 - When use with a sort method provides control over the sort order. sort theo nhieu category
 - Control the order of certain data structures such as sorted maps.
 - An ordering for collections of objects that do not have a natural ordering.

It is important to note the following:

- If `c.compare(e1, e2)==0` has the same boolean value as `e1.equals(e2)` for every e1 and e2 in S, the ordering by a comparator c on a set of elements S is consistent with equals.
- If the ordering by c on S is inconsistent with equals, the sorted set or map will behave abnormal. For example, when two elements a and b are added such that `(a.equals(b) && c.compare(a, b) != 0)` to an empty TreeSet with comparator c. The second add operation will then return true. This is because a and b are not equivalent for the TreeSet.

Note - To serialize a data structure (TreeSet, TreeMap), comparators can implement `java.io.Serializable`.

3.8 Enhancements in Collections Classes

Java SE 6 introduced new Collections classes as part of Collection APIs. These are as follows:

→ **ArrayDeque**

The `ArrayDeque` class implements the `Deque` interface. This class is faster than stack and linkedlist when used as a queue. It does not put any restriction on capacity and does not allow null values. It is not thread safe. In other words, it does not allow simultaneous access by multiple threads.

The `ArrayDeque` class provides `addFirst(E e)`, `addLast(E e)`, `getFirst()`, `getLast()`, `removeFirst()`, and `removeLast()` methods to add, retrieve, and remove elements from the first and last positions respectively. The elements can be accessed and traversed in forward and backward directions respectively. The `ArrayDeque` class also provides the `contains()` method that returns `true` if the deque contains the specified element.

Code Snippet 15 shows the use of some of the methods available in the `ArrayDeque` class.

Code Snippet 15:

```
import java.util.ArrayDeque;
import java.util.Iterator;
...
public static void main(String args[]) {
    ArrayDeque arrDeque = new ArrayDeque();
    arrDeque.addLast("Mango");
    arrDeque.addLast("Apple");
    arrDeque.addFirst("Banana");
    for (Iterator iter = arrDeque.iterator(); iter.hasNext();) {
        System.out.println(iter.next());
    }
    for (Iterator descendingIter = arrDeque.descendingIterator();
        descendingIter.hasNext()) {
        System.out.println(descendingIter.next());
    }
    System.out.println("First Element : " + arrDeque.getFirst());
    System.out.println("Last Element : " + arrDeque.getLast());
    System.out.println("Contains \"Apple\" : " + arrDeque.
contains("Apple"));
}
```

```
}
```

```
...
```

The code creates an `ArrayDeque` instance, adds elements to it and then traverses in forward and backward directions. The methods, `getFirst()` and `getLast()` retrieves the first and the last elements, `Banana` and `Apple` respectively. The `contains()` method checks whether the `Arraydeque` contains the element, `Apple` and returns true after finding the specified element.

→ **ConcurrentSkipListSet**

The `ConcurrentSkipListSet` class implements the `NavigableSet` interface. The elements are sorted based on natural ordering or by a `Comparator`. The `Comparator` is an interface that uses the `compare()` method to sort objects that don't have a natural ordering. During the creation time of the set, the `Comparator` is provided. The `ConcurrentSkipListSet` class provides methods to return iterators in ascending or descending order. It also provides methods to return the closest matches of elements in a collection.

Syntax:

```
ConcurrentSkipListSet()
```

The constructor creates an instance with its elements sorted on natural order.

Table 3.5 lists and describes the different methods available in `ConcurrentSkipListSet` class.

Note that the argument `e` is the element to be matched in the list.

Method	Description
<code>ceiling(E e)</code>	Returns the least element greater than or equal to <code>e</code> or null, if there is no such element
<code>floor(E e)</code>	Returns the greatest element less than or equal to <code>e</code> or null, if there is no such element
<code>higher(E e)</code>	Returns the least element greater than <code>e</code> or null, if there is no such element
<code>lower(E e)</code>	Returns the greatest element less than <code>e</code> or null, if there is no such element

Table 3.5: Methods of `ConcurrentSkipListSet` class

Code Snippet 16 shows the use of some of the methods available in `ConcurrentSkipListSet` class.

Code Snippet 16:

```
import java.util.Iterator;
import java.util.concurrent.ConcurrentSkipListSet;
...
public static void main(String args[]) {
```

```

ConcurrentSkipListSet fruitSet = new ConcurrentSkipListSet();

fruitSet.add("Banana");
fruitSet.add("Peach");
fruitSet.add("Apple");
fruitSet.add("Mango");
fruitSet.add("Orange");

// Displays in ascending order
Iterator iterator = fruitSet.iterator();

System.out.print("In ascending order :");

while (iterator.hasNext())
    System.out.print(iterator.next() + " ");

// Displays in descending order
System.out.println("In descending order: " +
fruitSet.descendingSet() + "\n");

System.out.println("Lower element: " + fruitSet.lower("Mango"));

System.out.println("Higher element: " + fruitSet.higher("Apple"));

}

...

```

The code creates a `ConcurrentSkipListSet` instance and adds elements into it. It then sorts its elements in ascending and descending order, and finds lower and higher elements. Here, the elements are displayed in descending order using the `descendingSet()` method. Also, the methods, `lower()` and `higher()` returns the element before `Mango` and after `Apple` as `Banana` respectively.

→ `ConcurrentSkipListMap`

The `ConcurrentSkipListMap` class implements `ConcurrentNavigableMap` interface. It belongs to `java.util.concurrent` package. Like `ConcurrentHashMap` class, the `ConcurrentSkipListMap` class allows modification without locking the entire map.

Syntax:

`ConcurrentSkipListMap()`

The constructor creates a new empty map where, the map is sorted according to the natural ordering of the keys.

Table 3.6 lists and describes the different methods available in the `ConcurrentSkipListMap` class.

Method	Description
<code>descendingMap()</code>	Reverses all the data in the descending order
<code>firstEntry()</code>	Returns data present with the lowest key in the map
<code>ceilingEntry(K key)</code>	Returns closest value that is greater than or equal to the specified key, or null if there is no such key
<code>lastEntry()</code>	Returns data present with the greatest key in the map
<code>put(K key, V value)</code>	Inserts previous value associated with specified key, or null if no such value exists. Here, <code>key</code> is the key with which a specified value is associated, and <code>value</code> is the <code>value</code> with which a specified key is associated.

Table 3.6: Methods of `ConcurrentSkipListMap` Class

Code Snippet 17 shows the use of some of the methods available in `ConcurrentSkipListMap` class.

Code Snippet 17:

```
import java.util.concurrent.ConcurrentSkipListMap;
...
...
public static void main(String args[]) {
    ConcurrentSkipListMap fruits = new ConcurrentSkipListMap();
    fruits.put(1, "Apple");
    fruits.put(2, "Banana");
    fruits.put(3, "Mango");
    fruits.put(4, "Orange");
    fruits.put(5, "Peach");
    // Retrieves first data
    System.out.println("First data: " + fruits.firstEntry() + "\n");
    // Retrieves last data
}
```

```

System.out.println("Last data: " + fruits.lastEntry() + "\n");
// Displays all data in descending order
System.out.println("Data in reverse order: " + fruits.descendingMap());
}
...

```

The code inserts, retrieves, and reverses the data from a `ConcurrentSkipListMap` instance. The `firstEntry()` and `lastEntry()` methods displays the key-value mapping result for the first and the last elements respectively.

→ **LinkedBlockingDeque**

The `LinkedBlockingDeque` class implements the `BlockingDeque` interface. The class belongs to `java.util.concurrent` package. In this class, you can specify the capacity for storing the elements. If you did not specify the capacity then maximum capacity will have the value of `Integer.MAX_VALUE`. The class contains linked nodes that are dynamically created after each insertion. The syntax for the constructors are as follows:

Syntax:

- `LinkedBlockingDeque()`

The constructor creates an instance with default capacity as specified in `Integer.MAX_VALUE`.

- `LinkedBlockingDeque(int capacity)`

The constructor creates an instance with specified capacity.

Methods

Table 3.7 explains the different methods available in `LinkedBlockingDeque` class.

Note that the argument `e` is the element to be added to the deque.

Method	Description
<code>addFirst(E e)</code>	Inserts the specified element, <code>e</code> , at the beginning of the deque. It does not violate the capacity restriction
<code>pollFirst()</code>	Removes and returns the first element of the deque if available, otherwise it returns null
<code>peekFirst()</code>	Returns but does not removes the first element of deque if available, otherwise it returns null

Table 3.7: Methods of `LinkedBlockingDeque` Class

Code Snippet 18 shows the implementation of `LinkedBlockingDeque` class and use of some of its available methods.

Code Snippet 18:

```
/* ProducerDeque.Java */
import java.util.concurrent.BlockingDeque;
class ProducerDeque implements Runnable {
    private String name;
    private BlockingDeque blockDeque;
    public ProducerDeque(String name, BlockingDeque blockDeque) {
        this.name = name;
        this.blockDeque = blockDeque;
    }
    public void run() {
        for (int i = 1; i < 10; i++) {
            try {
                blockDeque.addFirst(i);
                System.out.println(name + " puts " + i);
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (IllegalStateException ex) {
                System.out.println("Deque filled upto the maximum capacity");
                System.exit(0);
            }
        }
    }
}
/* ConsumerDeque.Java */
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;
class ConsumerDeque implements Runnable {
    private String name;
```

```
private BlockingDeque blockDeque;  
public ConsumerDeque(String name, BlockingDeque blockDeque) {  
    this.name = name;  
    this.blockDeque = blockDeque;  
    public void run() {  
        for (int i = 1; i < 10; i++) {  
            try {  
                int j = (Integer) blockDeque.peekFirst();  
                System.out.println(name + " takes " + j);  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}  
/* LinkedBlockingDequeClass.Java */  
import java.util.concurrent.BlockingDeque;  
import java.util.concurrent.LinkedBlockingDeque;  
public class LinkedBlockingDequeClass {  
    public static void main(String[] args) {  
        BlockingDeque blockDeque = new LinkedBlockingDeque(5);  
        Runnable produce = new ProducerDeque("Producer", blockDeque);  
        Runnable consume = new ConsumerDeque("Consumer", blockDeque);  
        new Thread(produce).start();  
        new Thread(consume).start();  
    }  
}
```

The code creates three classes, `ProducerDeque`, `ConsumerDeque`, and `LinkedBlockingDequeClass`. The `ProducerDeque` class creates an instance of `BlockingDeque` that invokes the `addFirst()` method to insert 10 integers at the front of the deque. The `ConsumerDeque` class also creates an instance of `BlockingDeque` that invokes the `peekFirst()` method to retrieve and remove the last integer value present in the deque. In the main class, `LinkedBlockingDequeClass`, an object of `LinkedBlockingDeque` class is created that allows you to store a maximum of 5 elements in the deque. If you try to insert more than 5 elements in the deque, it will throw an `IllegalStateException`. The main class, on execution first runs the `Producer` thread and inserts the integer value 1 at the front of the deque.

Next, the `Consumer` thread runs and retrieves the integer value 1, but does not remove it from the head. Again, the `Producer` thread executes and inserts 2 at the front of the deque. Now, once again, the `Consumer` thread runs and retrieves the value 2 present at the front of the deque, but does not remove it from the head. This process repeats until the integer value 4 is retrieved from the head of the queue. Now, as soon as the integer value 5 tries to enter the deque a message is displayed that the deque is filled upto the maximum capacity. This is because the deque has reached its maximum capacity an `IllegalStateException` is thrown.

→ `AbstractMap.SimpleEntry`

The `AbstractMap.SimpleEntry` is static class nested inside `AbstractMap` class. This class is used to implement custom map. An instance of this class stores key-value pair of a single entry in a map. The value of the entry can be changed.

The `getKey()` method returns the key of an entry in the instance. The `getValue()` method returns the value of an entry. The `setValue()` method updates the value of an entry.

Syntax:

```
AbstractMap.SimpleEntry(K key, V value)
```

The constructor creates an instance with specified key-value pair that is denoted by `key` and `value`.

Code Snippet 19 shows the implementation of `AbstractMap.SimpleEntry` static class and the use of some of its available methods.

Code Snippet 19:

```
AbstractMap.SimpleEntry<String, String> se = new AbstractMap.SimpleEntry<String, String>("1", "Apple");
System.out.println(se.getKey());
System.out.println(se.getValue());
se.setValue("Orange");
System.out.println(se.getValue());
```

The code creates instances with an entry having key-value pair as **(1, Apple)**. It then retrieves key and value using `getKey()` and `getValue()` methods, and replaces the value of last entry from **Apple** to **Orange** using the `setValue()` method.

→ **AbstractMap.SimpleImmutableEntry**

The `AbstractMap.SimpleImmutableEntry` class is a static class nested inside the `AbstractMap` class. As the name suggests, it does not allow modifying a value in an entry. If any attempt to change a value is made, it results in throwing `UnsupportedOperationException`.

3.9 Check Your Progress

1. Which of these statements related to `java.util` package are true?

(A)	Collection is a container that helps to group multiple elements into a single unit.		
(B)	Collections Framework provides classes and interfaces for storing and manipulating groups of object in a standardized way.		
(C)	Collections Framework was developed so as to have low degree of interoperability.		
(D)	Collections Framework does not contain Collection interface at the top of the hierarchy.		

(A)	A, B	(C)	C, D
(B)	B	(D)	D

2. Which of these statements regarding Lists are true?

(A)	Lists allow access to elements based on their position.		
(B)	Lists do not have a method to search for a specific element.		
(C)	<code>ArrayList()</code> creates an empty array.		
(D)	In a <code>Vector</code> , the size of the vector can be increased or decreased.		

(A)	A	(C)	C
(B)	B	(D)	D

3. The `getFirst()` method _____.

(A)	Retrieves, but does not remove the last element of the list.		
(B)	Retrieves, but does not remove the first element of the list.		
(C)	Removes and returns the first element of the list.		
(D)	Removes and returns the last element of the list.		

(A)	A	(C)	C
(B)	B	(D)	D

4. Which of these statements about the different Set classes are true?

(A)	Set class does not permit duplication of elements.
(B)	SortedSet class allows duplication of elements.
(C)	HashSet class guarantee the order of elements.
(D)	LinkedHashSet maintains the order of the items added to the Set.

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of these statements related to the different classes of Map are true?

(A)	TreeMap is used to arrange the keys in a sorted order.
(B)	HashMap allows null values to be used as keys.
(C)	TreeMap stores elements in a tree structure.
(D)	LinkedHashMap is a combination of hash map and linked list.
(E)	LinkedHashMap returns values in the order they were added to the Map.

(A)	A	(C)	C
(B)	A, B	(D)	B, C

3.9.1 Answers

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

Summary

- The java.util package contains the definition of number of useful classes providing a broad range of functionality.
- The List interface is an extension of the Collection interface.
- The Set interface creates a list of unordered objects.
- A Map object stores data in the form of relationships between keys and values.
- A Queue is a collection for holding elements before processing.
- ArrayDeque class does not put any restriction on capacity and does not allow null values..
- AbstractMap.SimpleEntry is used for implementation of custom map.
- AbstractMap.SimpleImmutableEntry class is a static class and does not allow modification of values in an entry.

social media

Balanced Learner-Oriented Guide

for enriched learning available



www.onlinevarsity.com

Session 4

Generics

Welcome to the Session, **Generics**.

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

In this Session, you will learn to:

- Identify the need for Generics
- List the advantages and limitations of Generics
- Explain generic class declaration and instantiation
- Define and describe generic methods
- Describe the relationship between Collection and Generics
- Explain the wildcard argument
- Describe the use of inheritance with Generics
- Describe the use of legacy code in Generics and Generics in legacy code
- Explain type inference



4.1 Introduction

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

4.1.2 Generics Overview

Generics in Java code generates one compiled version of a generic class. The introduction of Generics in Java classes will help remove the explicit casting of a class object so the ClassCastException will not arise during compilation. Generics will help to remove type inconsistencies during compile time rather than at run time. Generics are added to the Java programming language because they enable:

- Getting more information about a collection's type.
- Keeping track of the type of elements a collection contains.
- Using casts all over the program.

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

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

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

Code Snippet 1 displays the code for non-generic code.

Code Snippet 1:

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

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

Code Snippet 2 displays the code for generic code.

Code Snippet 2:

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

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

4.1.3 Advantages and Limitations of Generics

The advantages of Generics are as follows:

- Generics allow flexibility of dynamic binding.
- Generic type helps the compiler to check for type correctness of the program at the compile time.
- In Generics, the compiler detected errors are less time consuming to fix than runtime errors.
- The code reviews are simpler in Generics as the ambiguity is less between containers.
- In Generics, codes contain lesser casts and thus help to improve readability and robustness.

The limitations of Generics are as follows:

- In Generics, you **cannot create generic constructors**.
- **A local variable cannot be declared where the key and value types are different from each other.**

4.2 Generic Classes

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

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

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

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

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

4.2.1 Declare and Instantiate Generic Class

To create an instance of the generic class, the `new` keyword is used along with the class name except that the type parameter argument is passed between the class name and the parentheses. The type parameter argument is replaced with the actual type when an object is created from a class. A generic class is shared among all its instances.

Syntax:

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

Code Snippet 3 displays the declaration of a generic class.

Code Snippet 3:

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

{}

...

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

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

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

- ➔ K - Key
- ➔ T - Type
- ➔ V - Value
- ➔ N - Number
- ➔ E - Element
- ➔ S, U, V, and, so on

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

Code Snippet 4:

```
import java.util.*;  
  
class TestQueue<DataType> {  
    private LinkedList<DataType> items = new LinkedList<DataType>();  
    public void enqueue(DataType item) {  
        items.addLast(item);  
    }  
    public DataType dequeue() {  
        return items.removeFirst();  
    }  
}
```

```

public boolean isEmpty() {
    return (items.size() == 0);
}

public static void main(String[] args) {
    TestQueue<String> testObj = new TestQueue<>();
    testObj.enqueue("Hello");
    testObj.enqueue("Java");
    System.out.println((String) testObj.dequeue());
}
}

```

In Java SE 7 and later, the required type arguments can be replaced to invoke the constructor of a generic class with an empty set of type arguments (`<>`). The pair of angle brackets (`<>`) is called the diamond. It is important to note that the compiler should determine the type arguments from the context when using the empty set of type arguments.

In Code Snippet 5, an instance of `TestQueue` will accept `String` as a type parameter.

Code Snippet 5:

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

In the code, a generic class is created that implements the concept of queue, and dequeue on any data types such as `Integer`, `String`, and `Double`. The type variable or type parameter, `<DataType>`, is used for the argument type and return type of the two methods. Type parameters can have any name. Type parameters can be compared to formal parameters in subroutines. The name will be replaced by the actual name when the class will be used to create an instance. Here, `<DataType>` has been replaced by `String` within the `main()` method while instantiating the class. Figure 4.1 displays the output.

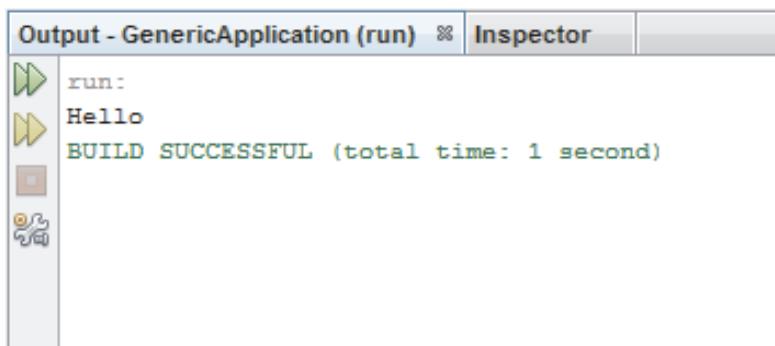


Figure 4.1: TestQueue - Output

Note - To understand the difference between the type variable and an ordinary class or interface name, type parameters are single, uppercase letters.

4.3 Generic Methods

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

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

The scope of the method's type parameter is the method declaration. The scope of type parameters can also be type parameters of other type parameters.

Syntax:

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

Code Snippet 6 displays the use of a generic method.

Code Snippet 6:

```
class NumberList <T>
{
    public<T> void display(T[] val)
    {
        for (T element : val)
        {
            System.out.printf("Values are: %s ", element);
        }
    }

    public static void main(String [] args)
    {
        Integer[] intValue = {1, 7, 9, 15};
        NumberList<Integer> listObj = new NumberList<> ();
        listObj.display(intValue);
    }
}
```

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

Figure 4.2 displays the output.

```
Output - GenericApplication (run)  ✘ Inspector
run:
Values are: 1 Values are: 7 Values are: 9 Values are: 15 BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 4.2: NumberList - Output

A generic class can have two or more type parameters.

Code Snippet 7 demonstrates how to declare a class with two type parameters.

Code Snippet 7:

```
import java.util.*;

public class TestQueue<DataType1, DataType2> {

    private final DataType2 num;

    private LinkedList<DataType1> items = new LinkedList<>();

    public TestQueue(DataType2 num) {
        this.num = num;
    }

    public void enqueue(DataType1 item) {
        items.addLast(item);
    }

    public DataType1 dequeue() {
        return items.removeLast();
    }
}
```

Like any other class, generic classes can also be subclassed by either generic or non-generic classes. A non-generic subclass can extend a superclass by specifying the required parameters and thus making it concrete.

Code Snippet 8 demonstrates how to declare a non-generic subclass.

Code Snippet 8:

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

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

Figure 4.3 displays the output.

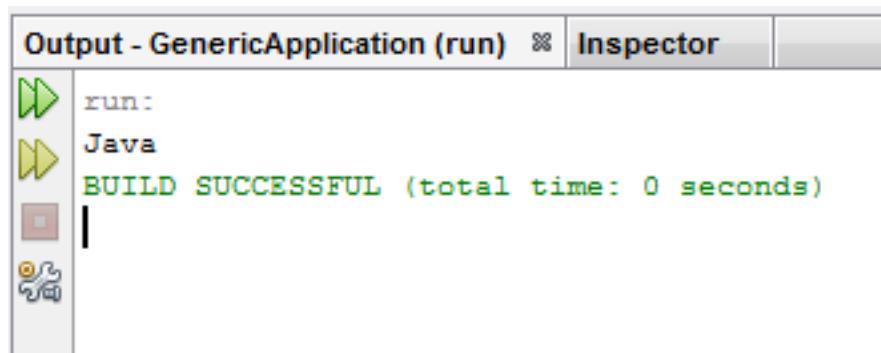


Figure 4.3: GenericApplication - Output

Code Snippet 9 demonstrates the creation of a generic subclass.

Code Snippet 9:

```
...
class MyTestQueue<DataType> extends TestQueue<DataType> {
    public static void main(String[] args) {
        MyTestQueue<String> test = new MyTestQueue<String>();
```

```

test.enqueue("Hello");
test.enqueue("Java");
System.out.println((String) test.dequeue());
}
}
...

```

4.3.1 Declare Generic Methods

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

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

Code Snippet 10:

```

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

```

The code shows two methods namely, `containsAll` and `addAll`. In both the methods, the type parameter `T` is used only once.

For constructors, the type parameters are not declared in the constructor but in the header that declares the class. The actual type parameter are passed while invoking the constructor.

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

Code Snippet 11:

```

import java.util.*;
class StudPair<T, U> {
    private T name;
    private U rollNumber;
}

```

```

public StudPair(T nmObj, U rollNo) {
    this.name = nmObj;
    this.rollNumber = rollNo;
}

public T displayName() {
    return name;
}

public U displayNumber() {
    return rollNumber;
}

public static void main(String [] args) {
    StudPair<String, Integer> studObj = new StudPair<>("John", 2);
    System.out.println(studObj.displayName());
    System.out.println(studObj.displayNumber());
}
}

```

In the code, a generic constructor is declared containing two generic type parameters separated by a comma. Figure 4.4 displays the output.

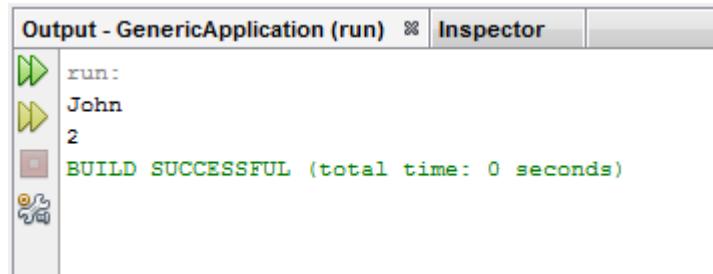


Figure 4.4: StudPair - Output

4.3.2 Accept Generic Parameters

A single generic method declaration can be called with arguments of different types. According to the types of the arguments passed, the compiler handles each method call. Generic methods can be defined based on the following rules:

- Each type parameter section includes one or more type parameters separated by commas. A type parameter is an identifier that specifies a generic type name.

Note - A type parameter is also called a type variable.

- All generic method declarations have a type parameter section delimited by angle brackets preceding the method's return type.
- A generic method's body should include type parameters that represent only reference types.

Note - A generic method is declared like any other method.

- The type parameters can be used to declare the return type. They are placeholders for the types of the arguments passed to the generic method. These arguments are called actual type arguments.

Code Snippet 12 displays a generic method declaration.

Code Snippet 12:

```
public class GenericAcceptReturn {

    public static<E> void displayArray(E[] acceptArray) {
        // Display array elements
        for (E element : acceptArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

    public static void main(String args[]) {
        // Create arrays of Integer, Double and Character
        Integer[] intArrayObj = {100, 200, 300, 400, 500};
        Double[] doubleArrayObj = {51.1, 52.2, 53.3, 54.4};
        Character[] charArrayObj = {'J', 'A', 'V', 'A'};

        System.out.println("Integer Array contains:");
        displayArray(intArrayObj);

        System.out.println("\nDouble Array contains:");
        displayArray(doubleArrayObj);
    }
}
```

```

        System.out.println("\nCharacter Array contains:");
        displayArray(charArrayObj);
    }
}

```

In the code, the `displayArray()` is a generic method declaration that accepts different type of arguments and displays them.

Figure 4.5 displays the output.

```

Output - GenericApplication (run) ✘ Inspector
run:
Integer Array contains:
100 200 300 400 500

Double Array contains:
51.1 52.2 53.3 54.4

Character Array contains:
J A V A

BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 4.5: GenericAcceptReturn - Output

4.3.3 Return Generic Types

A method can also return generic data type. Code Snippet 13 displays a method having a generic return type.

Code Snippet 13:

```

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

```

```

        if (val3.compareTo(maxValue) > 0)
            maxValue = val3;

        return maxValue;
    }

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

```

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

Figure 4.6 displays the output.

```

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

```

Figure 4.6: GenericReturnTest - Output

4.4 Type Inference

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

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

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

The type arguments required to invoke the constructor of a generic class can be replaced with an empty set of type parameters (<>) as long as the compiler infers the type arguments from the context.

Code Snippet 14 illustrates this:

Code Snippet 14:

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

In Java SE 7, the parameterized type of the constructor can be replaced with an empty set of type parameters as displayed in Code Snippet 15.

Code Snippet 15:

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

Note - Empty set of type parameters is important to use automatic type inference during generic class instantiation.

In Code Snippet 16, the compiler generates an unchecked conversion warning:

Code Snippet 16:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

In the code snippet, the `HashMap()` constructor refers to the `HashMap` raw type.

Java SE 7 supports limited type inference for generic instance creation. The type inference can be used only if the parameterized type of the constructor is apparent from the context. Code Snippet 17 illustrates this.

Code Snippet 17:

```
List<String> list = new ArrayList<>();
list.add("A");

// The following statement should fail since addAll expects
// Collection<? extends String>

list.addAll(new ArrayList<>());
```

The code does not compile.

Note - It is recommended to use the angle brackets for variable declarations.

Code Snippet 18 when executed compiles.

Code Snippet 18:

```
// The following statements compile:  
List<? extends String> list2 = new ArrayList<>();  
list.addAll(list2);
```

4.4.1 Generic Constructors of Generic and Non-Generic Classes

Constructors can declare their own formal type parameters in both generic and non-generic classes. Code Snippet 19 illustrates this.

Code Snippet 19:

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

Code Snippet 20 shows the instantiation of the class MyClass.

Code Snippet 20:

```
new MyClass<Integer>("")
```

This code is valid in Java SE 7 and prior releases. In the Code Snippet:

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

The compiler in Java SE 7 understands the actual type parameters of the generic class that is instantiated with the angle brackets. Prior to Java SE 7, compilers understand the actual type parameters of generic constructors just as generic methods understand.

Code Snippet 21 is valid for Java SE 7.

Code Snippet 21:

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

In the code, the compiler understands:

- The type Integer is for the formal type parameter, X, of the generic class MyClass<X>.
- The type String is for the formal type parameter, T, of the constructor of the generic class.

4.4.2 Java SE 7 Enhancements

- **Underscores in Numeric Literals:** Underscore characters (_) can be added anywhere between digits in a numerical literal to separate groups of digits in numeric literals. They improve the readability of the code.
- **Strings in switch Statements:** The String class can be used in the expression of a switch statement.
- **Binary Literals:** In Java SE 7, the integral types can be defined using the binary number system. Note, the integral types are of byte, short, int, and long type. To specify a binary literal, add the prefix 0b or 0B to the number.
- **Better Compiler Warnings and Errors with Non-Reifiable Formal Parameters:** The Java SE 7 compiler generates a warning at the declaration site of a varargs method or constructor with a non-reifiable varargs formal parameter. Java SE 7 suppresses these warnings using the compiler option -Xlint:varargs and the annotations @SafeVarargs and @SuppressWarnings({"unchecked", "varargs"}).
- **Type Inference for Generic Instance Creation:** The required type arguments can be replaced to invoke the constructor of a generic class with an empty set of type parameters as long as the compiler infers the type arguments from the context.
- **Catching Multiple Exception Types:** A single catch block handles many types of exception. Users can define specific exception types in the throws clause of a method declaration because the compiler executes accurate analysis of rethrown exceptions.
- **The try-with-resources Statement:** This declares one or more resources, which are objects that should be closed after the programs have finished working with them. Object that implements the new java.lang.AutoCloseable interface or the java.io.Closeable interface can be used as a resource. The statement ensures that each resource is closed at the end of the statement.

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

4.5 Collection and Generics

Collection is an object that manages a group of objects. Collection API depends on generics for its implementation. Code Snippet 22 illustrates this.

Code Snippet 22:

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

Figure 4.7 displays the output that displays the usage of collection API and generics.

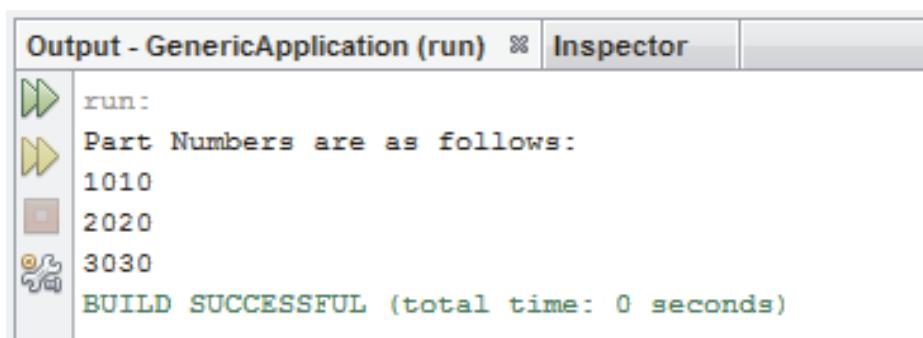


Figure 4.7: GenericArrayListExample - Output

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

4.5.1 Wildcards with Generics

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



?

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

Code Snippet 23 displays the use of unbounded wildcard.

Code Snippet 23:

```
public class Paper
{
    public void draw (Shape shapeObj)
    {
        shapeObj.draw(this);
    }

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

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



? extends Type

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

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

Code Snippet 24 displays the declaration of the bounded wildcard '`? extends Type`'.

Code Snippet 24:

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

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

→ `? super Type`

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

Code Snippet 25 displays the use of '`? super Type`' bounded wildcard.

Code Snippet 25:

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

In the method '`<? super T>`' indicates that the destination object, `destObj`, may have elements of any type which is a supertype of `T`. Similarly, the statement '`<? extends T>`' means that the source object, `srcObj`, may have elements of any type that is a subtype of `T`. The method `copyAll()` invokes the `copy()` method with the parameter type as `Object`. This method invocation

works because `srcObj` has type `List<Object>` which is the class itself. The object, `intObj` has type as `List<Integer>` which is a subtype of `Object`.

Figure 4.8 displays the different types of wildcards.

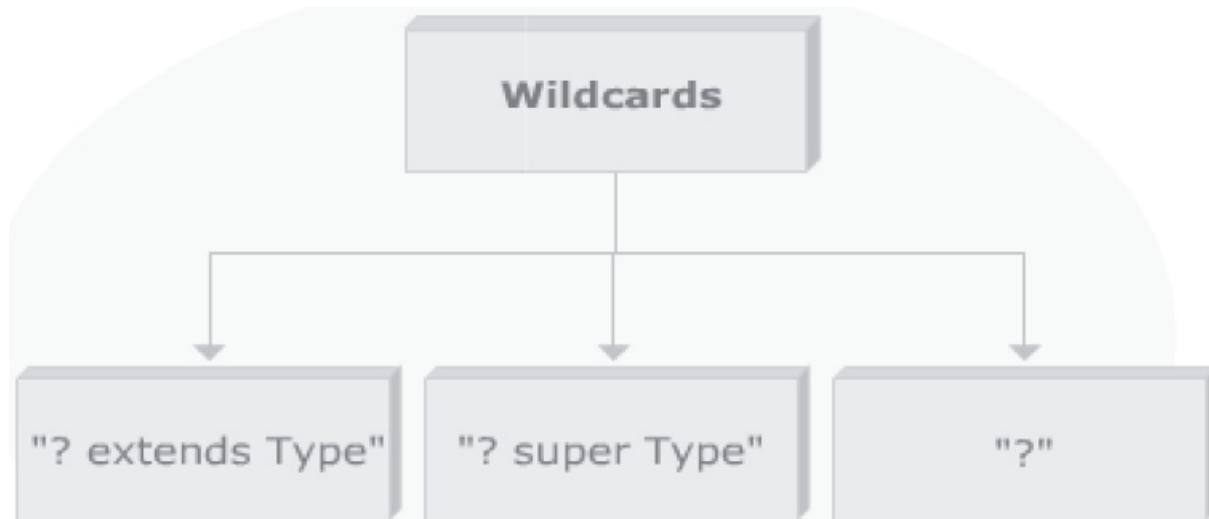


Figure 4.8: Wildcards

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

4.5.2 Exception Handling with Generics

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

In Generics, the type variable can be used in the `throws` clause of the method signature.

Figure 4.9 displays exception handling with Generics.

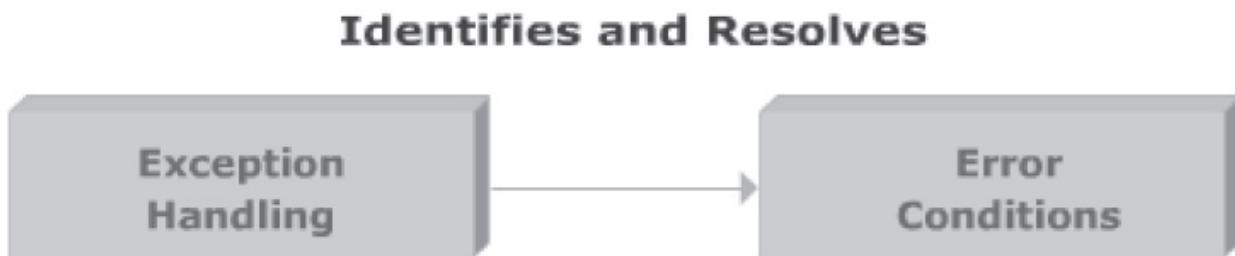


Figure 4.9: Exception Handling with Generics

Code Snippet 26 displays the use of generic type with exceptions.

Code Snippet 26:

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

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

The code shows how to use genericity in `Exception`. The code uses a type variable in the throws clause of the method signature. The usage of the type parameter `X` shows that the code may throw an exception.

4.5.3 Inheritance with Generics

Inheritance is a mechanism to derive new classes or interfaces from the existing ones. Object-oriented programming allows classes to inherit commonly used state and behavior from other classes.

Classes can extend generic classes, and provide values for type parameters or add new type parameters. A class cannot inherit from parametric type. Two instantiations of the same generic type cannot be used in inheritance.

Code Snippet 27 displays the use of generics with inheritance.

Code Snippet 27:

```
class Month<T>
{
    T monthObj;
    Month(T obj)
    {
        monthObj = obj;
    }
}
```

```
// Return monthObj
T getob()
{
    return monthObj;
}

// A subclass of Month that defines a second type parameter, called V.
class MonthArray<T, V> extends Month<T>
{
    V valObj;
    MonthArray(T obj, V obj2)
    {
        super(obj);
        valObj = obj2;
    }
    V getob2()
    {
        return valObj;
    }
}
// Create an object of type MonthArray
public class HierTest
{
    public static void main(String args[])
    {
        MonthArray<String, Integer> month;
        month = new MonthArray<>("Value is:", 99);
        System.out.print(month.getob());
        System.out.println(month.getob2());
    }
}
```

```
}
```

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

Figure 4.10 displays the output.

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

Figure 4.10: HierTest - Output

4.6 Interoperability with Generics

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

In Java, genericity ensures that the same class file is generated by both legacy and generic versions with some additional information about types. This is known as binary compatibility as the legacy class file can be replaced by the generic class file without recompiling. Figure 4.11 displays the adaptation between the legacy code and the new code.

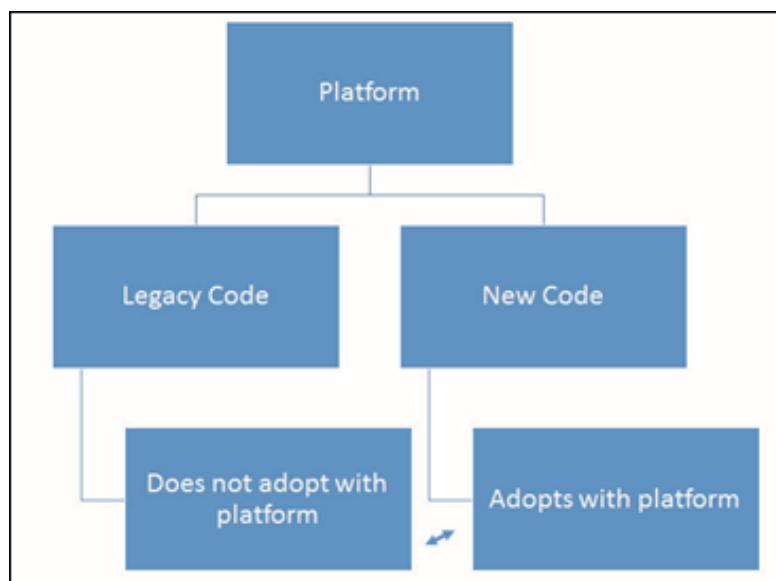


Figure 4.11: Legacy Code

Code Snippet 28 displays the use of legacy code with legacy client.

Code Snippet 28:

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

class NumArrayList implements NumStack
{
    private List listObj;
    public NumArrayList()
    {
        listObj = new ArrayList();
    }
    @Override
    public Object retrieve()
    {
        Object value = listObj.remove(listObj.size() - 1);
        return value;
    }
    @Override
    public String toString()
    {
        return "stack" + listObj.toString();
    }
}
public class Client
{
    public static void main(String[] args)
    {
```

```

NumStack stackObj = new NumArrayStack();

for (int ctr=0; ctr<4; ctr++)

{
    stackObj.push(new Integer(ctr));
}

assert stackObj.toString().equals("stack[0, 1, 2, 3]");
int top= ((Integer)stackObj.retrieve()).intValue();
System.out.println("Value is : " + top);
}
}

```

The code snippet displays the use of legacy code and a legacy client. In the code data is added to a stack and retrieved from the stack. This will compile well with JDK 1.4 version.

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

4.6.1 Generic Library with Legacy Client

In generic code, the classes are accompanied by a type parameter. When a generic type like collection is used without a type parameter, it is called a raw type. In the earlier example, the parameterized `NumStack<E>` corresponds to the raw type `NumStack` and parameterized `NumArrayStack<E>` corresponds to the raw type `NumArrayStack`.

A value of parameterized type can be passed to a raw type as parameterized type is a subtype of raw type. Java generates an unchecked conversion warning when a value of raw type is passed where a parameterized type is expected. In the same example, a value of `NumStack<E>` type can be assigned to a variable of `NumStack` type. But when the reverse is performed an unchecked conversion warning is displayed. Figure 4.12 displays generic library with legacy client.

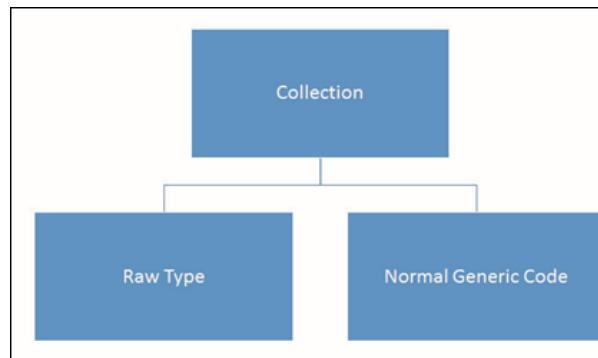


Figure 4.12: Generic Library with Legacy Client

Code Snippet 29 displays the use of generic library with legacy client.

Code Snippet 29:

```
import java.util.*;  
  
interface NumStack  
{  
    public boolean empty();  
    public void push(Object elt);  
    public Object retrieve();  
}  
  
class NumArrayList implements NumStack  
{  
    private List listObj;  
    public NumArrayList()  
    {  
        listObj = new ArrayList();  
    }  
    public boolean empty()  
    {  
        return listObj.size() == 0;  
    }  
    public void push(Object obj)  
    {  
        listObj.add(obj);  
    }  
    public Object retrieve()  
    {  
        Object value = listObj.remove(listObj.size() - 1);  
        return value;  
    }  
}
```

```

public String toString()
{
    return "stack"+listObj.toString();
}
}

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

```

When the code is compiled, an unchecked conversion warning is displayed as shown:

Note - Client.java uses unchecked or unsafe operation

Note - Recompile with -Xlint:unchecked for details.

If the code is compiled using the switch as suggested then the following message appears:

```

Client.java:21: warning: [unchecked] unchecked call to add(E) as a member of
the raw type java.util.List
listObj.add(obj);
^
1 warning

```

The warning is due to the use of the generic method add in the legacy method retrieve. The warnings can be turned off by using the switch -source 1.4 as shown:

```
javac -source 1.4 Client.java
```

Note - The unchecked conversion warning means that the same safety guarantees which are given to generic code while compiling cannot be offered.

4.6.2 Erasure

When you insert an integer into a list, and try to extract a `String` it is wrong. If you extract an element from list, and by casting that to `String` if you try to treat that as `String`, you will get `ClassCastException`. The reason is that Generics are implemented by the Java compiler as a front end conversion called erasure. Erasure removes all generic type information. All the type information between angle brackets is thrown out, so, a parameterized type like `List<String>` is converted into `List`. Type erasure maintains compatibility with Java libraries and applications which are created before generics. Figure 4.13 displays erasure.

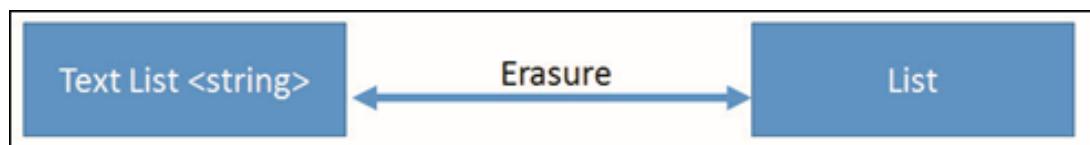


Figure 4.13: Erasure

4.6.3 Generics in Legacy Code

Sometimes it may be required to update the library not immediately but over a period of time. In such cases, the method signatures will change and will consist of the type parameters. The method body will not change. This change in the method signature can be performed by making minimum changes in the method, or by creating stub or by using wrappers. The minimum changes that have to be incorporated are:

- Adding type parameter to class or interface declarations
- Adding type parameters to the class or interface which has been extended or implemented
- Adding type parameters to the method signatures
- Adding cast where the return type contains a type parameter

Code Snippet 30 displays the use of generics.

Code Snippet 30:

```
import java.util.*;  
  
interface NumStack<E>  
{  
    public boolean empty();  
    public void push(E elt);  
    public E retrieve();  
}  
  
@SuppressWarnings("unchecked")  
class NumArrayList<E> implements NumStack<E>  
{  
    private List listObj;  
    public NumArrayList()  
    {  
        listObj = new ArrayList();  
    }  
    public boolean empty()  
    {  
        return listObj.size() == 0;  
    }  
    public void push(E obj)  
    {  
        listObj.add(obj);  
    }  
    public E retrieve()  
    {  
        Object value = listObj.remove(listObj.size() - 1);  
        return (E) value;  
    }  
}
```

```
}

public String toString()
{
    return "stack"+listObj.toString();
}

}

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

This code creates an unchecked warning.

4.6.4 Using Generics in Legacy Code

A generic library should be created when there is access to source code. Update the entire library source as well as the client code to eliminate potential unchecked warnings.

Code Snippet 31 shows the use of generics in legacy code.

Code Snippet 31:

```
import java.util.*;  
  
interface NumStack<E>  
{  
    public void push(E elt);  
    public E retrieve();  
}  
  
class NumArrayListStack<E> implements NumStack<E>  
{  
    private List<E> listObj;  
    public NumArrayListStack()  
    {  
        listObj = new ArrayList<E>();  
    }  
    public void push(E obj)  
    {  
        listObj.add(obj);  
    }  
    public E retrieve()  
    {  
        E value = listObj.remove(listObj.size() - 1);  
        return value;  
    }  
    public String toString()  
    {  
        return "stack" + listObj.toString();  
    }  
}
```

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

The interface and the implementing class use the type parameter. The type parameter <E> replaces the Object type from the `push()` and `retrieve()` method signature and method body. Appropriate type parameters are added to the client code.

4.7 Check Your Progress

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

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

(A)	A, C	(C)	A, D
(B)	B, C	(D)	C, D

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

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

(A)	A, C	(C)	A, D
(B)	B, C	(D)	C, D

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

A.	Java generates an unchecked conversion warning when a value of raw type is passed where a parameterized type is expected.		
B.	Erasure adds all generic type information.		
C.	When a generic type like collection is used without a type parameter, it is called a raw type.		

D.	Erasure removes all generic type information.		
E.	The change in method signature can be performed by making minimal change or by creating a skeleton.		

(A)	A	(C)	B
(B)	C	(D)	D

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

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

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

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

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

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

4.7.1 Answers

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

Summary

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

TechnoWise



Are you a

TECHIE GEEK

looking for updates?

Logon to

www.onlinevarsity.com

Session 5

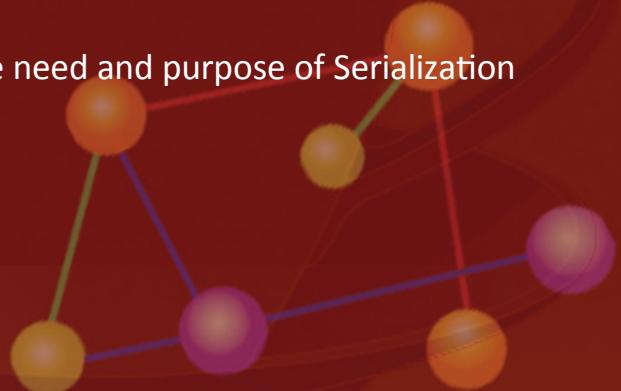
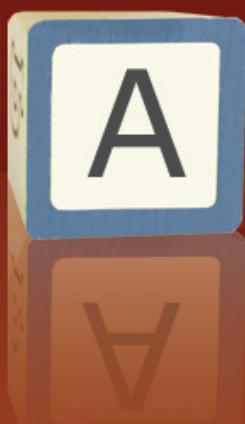
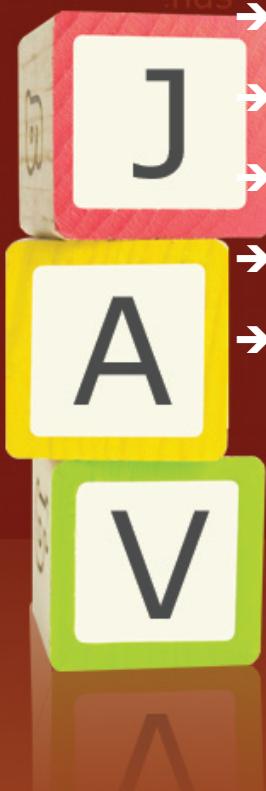
File Handling in Java

Welcome to the Session, **File Handling in Java**.

This session covers the `java.io` package that contains various classes known as Input/Output (I/O) streams. These streams manage the input and output operations to files, network connections and other I/O devices. The primary focus of this session is how to perform I/O operations using these streams.

In this Session, you will learn to:

- Define data streams
- Identify the need for streams
- Identify the purpose of the `File` class, its constructors, and methods
- Describe the `DataInput` and `DataOutput` interfaces
- Describe the byte stream and character stream in the `java.io` package
- Explain the `InputStream` and `OutputStream` classes
- Describe the `BufferedInputStream` and `BufferedOutputStream` classes
- Describe Character stream classes
- Describe the chaining of I/O systems
- Define Serialization and describe the need and purpose of Serialization



5.1 Introduction

Most of the application programs work with data stored in local files or coming from computers over the network. Java works with streams of data. A stream is a sequence of data. It can also be defined as a logical entity that produces or consumes information. A data stream is a channel through which data travels from a source to a destination. This source or destination can be an input or output device, storage media or network computers. A physical data storage is mapped to a logical stream and then a Java program reads data from this stream serially – one byte after another, or character after character. In other words, a physical file can be read using different types of streams, for example, `FileInputStream` or `FileReader`. Java uses such streams to perform various input and output operations.

5.2 Stream Classes

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

→ **in**

The standard input stream is used for reading characters of data. This stream responds to keyboard input or any other input source specified by the host environment or user. It has been defined as follows:

```
public static final InputStream in;
```

→ **out**

The standard output stream is used to typically display the output on the screen or any other output medium. It has been defined as follows:

```
public static final PrintStream out;
```

→ **err**

This is the standard error stream. By default, this is the user's console. It has been defined as follows:

```
public static final PrintStream err;
```

5.2.1 Need for Stream Classes

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

Thus, stream classes help in:

- Reading input from a stream
- Writing output to a stream

- Managing disk files
- Share data with a network of computers

To read or write data using Input/Output streams, the following steps need to be performed. They are:

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

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

5.3 File Class

Unlike other classes that work on streams, `File` class directly works with files and the file system. The files are named using the file-naming conventions of the host operating system. These conventions are encapsulated using the `File` class constants.

A pathname can be absolute or relative. In an absolute pathname, no other information is required in order to locate the required file as the pathname is complete. In a relative pathname, information is gathered from some other pathname. The classes in the `java.io` package resolve relative pathnames against the current user directory, which is named by the system property `user.dir`.

Note: This is typically the directory in which the Java Virtual Machine is invoked.

The parent of an abstract pathname is obtained by invoking the `getParent()` method of this class. This consists of the pathname's prefix and each name in the pathname's name sequence. Note that the last name is not included. Each directory's absolute pathname is an ancestor of any `File` object with an absolute abstract pathname. For example, '`/usr`' is an ancestor of the directory denoted by the pathname `/usr/local/bin`.

Note: In the pathname, `/usr/local/bin` pathname, `/usr` is an ancestor of the directory pathname `/usr/local/bin`.

The prefix concept is used to handle root directories on platforms such as UNIX. Table 5.1 describes the prefix that each platform uses.

Platform	Prefix Description
UNIX	“/” is the prefix for an absolute pathname. There is no prefix for relative pathnames. The abstract pathname indicating the root directory has the prefix “/” and an empty name sequence.
Microsoft Windows	The prefix of a pathname that contains a drive specifier includes a drive letter followed by “：“. This is followed by “\\” if the pathname is absolute. In a relative pathname, there is no prefix when a drive is not specified.

Table 5.1: Prefix Concept

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

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

The directory methods in the `File` class allow creating, deleting, renaming, and listing of directories.

The interfaces and classes defined by the `java.nio.file` package helps the Java virtual machine to access files, file systems, and file attributes. The `toPath()` method helps to obtain a `Path` that uses the abstract path. A `File` object uses this path to locate a file.

To diagnose errors when an operation on a file fails, the `Path` can be used with the `Files` class.

Note: When the `Path` is used with the `Files` class, there is generous access to additional I/O exceptions, file operations, and file attributes.

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

→ **`File(String dirpath)`**

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

```
public File(String dirpath)
```

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

The `File(String parent, String child)` constructor creates a `File` object with pathname of the file specified by the `String` variables `parent` and `child`.

If `parent` string is null, then the new `File` instance is created by using the given `child` pathname

string. This constructor will behave in a similar fashion as a single-argument `File` constructor on the given child pathname string.

Else, the parent pathname string is considered to point to a directory and the child pathname string is considered to point a directory or a file. If the child pathname string is absolute, it is converted into a relative pathname in a system-dependent way. Its signature is as follows:

```
public File(String parent, String child)
```

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

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

If `parent` is null, then the new `File` instance is created by using the given child pathname string. The single-argument `File` constructor on the given child pathname string gets invoked and the new `File` instance is created. If this does not occur, the parent abstract pathname is considered to point a directory and the child pathname string is considered to point a directory or a file.

The child pathname string is converted into a relative pathname in a system-dependent way if it is absolute. If `parent` is the empty abstract pathname, the new `File` instance is created. This occurs by converting `child` into an abstract pathname and resolving the result against a system-dependent default directory. If none of these occur, then, each pathname string is converted into an abstract pathname and the child abstract pathname is resolved against the parent. Its signature is as follows:

```
public File(File fileobj, String filename)
```

→ **`File(URL urlobj)`**

The `File(URL urlobj)` converts the given `file:URI` into an abstract pathname and creates a new `File` instance. The `file:URI` is system-dependent. Additionally, the transformation by the constructor will also be system-dependent. Its signature is as follows:

```
public File(URL urlobj)
```

Code Snippet 1 displays the creation of an instance of the `File` class.

Code Snippet 1:

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

5.3.1 Methods in File Class

The methods in `File` class help to manipulate the file on the file system. Some of the methods in the `File` class are as follows:

→ **`renameTo(File newname)`**

The `renameTo(File newname)` method will name the existing `File` object with the new name specified by the variable `newname`.

```
public boolean renameTo(File newname)
```

→ **`delete()`**

The `delete()` method deletes the file represented by the abstract path name.

```
public boolean delete()
```

→ **`exists()`**

The `exists()` method tests the existence of file or directory denoted by this abstract pathname.

```
public boolean exists()
```

→ **`getPath()`**

The `getPath()` method converts the abstract pathname into a pathname string.

```
public String getPath()
```

→ **`isFile()`**

The `isFile()` method checks whether the file denoted by this abstract pathname is a normal file.

```
public boolean isFile()
```

→ **`createNewFile()`**

The `createNewFile()` method creates a new empty file whose name is the path name for this file. It is only created when the file of similar name does not exist.

```
public boolean createNewFile() throws IOException
```

→ **`mkdir()`**

The `mkdir()` method creates the directory named by this abstract pathname.

```
public boolean mkdir()
```

→ **toPath()**

The `toPath()` method returns a `java.nio.file.Path` object constructed from the abstract path.

```
Path toPath()
```

→ **toURI()**

The `toURI()` method constructs a file, URI. This file represents this abstract pathname.

```
URI toURI()
```

Code Snippet 2 displays the use of methods of the `File` class.

Code Snippet 2:

```
...
File fileObj = new File("C:/Java/Hello.txt");
System.out.println("Path is: " + fileObj.getPath());
System.out.println("Name is: " + fileObj.getName());
System.out.println("File exists is: " + fileObj.exists());
System.out.println("File is: " + fileObj.isFile());
...
```

Code Snippet 2 displays the full path and the filename of the invoking `File` object. The code also checks for the existence of the file and returns true if the file exists and false if it does not. The `isFile()` method returns true if called on a file and returns false if called on a directory.

Figure 5.1 displays the output.



Figure 5.1: File Class Methods - Output

Code Snippet 3 displays the use `FilenameFilter` class to filter files with a specific extension.

Code Snippet 3:

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

The `FilenameFilter` interface defines an `accept()` method which is used to check if the specified file should be included in a file list. The method of this class returns true if the filename ends with `.java` extension as stored in the variable `ext`. The `list()` method restricts the visibility of the file and displays only those files which ends with the specified extension.

Figure 5.2 displays the output.

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

Figure 5.2: *FilenameFilter* Class - Output

5.3.2 *FileDescriptor* Class

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

The *FileDescriptor* class has the following public fields:

→ **static final FileDescriptor err**

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

→ **static final FileDescriptor in**

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

→ **static final FileDescriptor out**

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

5.3.3 Constructor and Methods of FileDescriptor

The constructor and methods of the `FileDescriptor` class are:

→ **`FileDescriptor()`**

The constructor creates an (invalid) `FileDescriptor` object.

```
public FileDescriptor()
```

→ **`sync()`**

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

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

→ **`valid()`**

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

```
public boolean valid()
```

Code Snippet 4 displays the use of the `FileDescriptor` object.

Code Snippet 4:

```
if (!fdobj.valid())
throw new SecurityException("Invalid FileDescriptor");
...
```

5.4 DataInput Interface and DataOutput Interface

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

The `DataInput` interface has methods for:

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

→ Converting data from Java modified Unicode Transmission Format (UTF)-8 format into string form.

UTF-8 is a special format for encoding 16 bit Unicode values.

Note: UTF-8 assumes that in most cases, the upper 8 bits of a Unicode will be 0 and optimizes accordingly.

The `DataOutput` interface has methods for:

- Converting data present in Java primitive type into a series of bytes and write them into a binary stream.
- Converting string data into Java-modified UTF-8 format and write it into a stream.

5.4.1 Methods of `DataInput` Interface

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

The methods in the `DataInput` interface are as follows:

→ **`readBoolean()`**

The `readBoolean()` method reads an input byte from a stream and returns true if the byte is not zero and false otherwise.

```
boolean readBoolean() throws IOException
```

→ **`readByte()`**

The `readByte()` method reads one byte from a stream which is a signed value in the range from -128 to 127.

```
byte readByte() throws IOException
```

→ **`readInt()`**

The `readInt()` method reads four bytes from a stream and returns the `int` value of the bytes read.

```
int readInt() throws IOException
```

→ **`readDouble()`**

The `readDouble()` method reads eight bytes from a stream and returns a `double` value of the bytes read.

```
double readDouble() throws IOException
```

→ **`readChar()`**

The `readChar()` method reads two bytes from a stream and returns a `char` value.

```
char readChar() throws IOException
```

→ **readLine()**

The `readLine()` method reads a line of text from the input stream. It reads a byte at a time and then converts the byte into a character and goes on reading until it encounters the end of line or end of file. The characters are then returned as a `String`.

```
String readLine() throws IOException
```

→ **readUTF()**

The `readUTF()` method reads a line of text in the modified UTF-8 format from a stream.

```
String readUTF() throws IOException
```

Figure 5.3 displays the methods of `DataInput` interface.

```
public abstract void close()
    throws IOException
public abstract long getFilePointer()
    throws IOException
public abstract long length()
    throws IOException
public abstract int read()
    throws IOException
public abstract int read(byte abyte0[])
    throws IOException
public abstract int read(byte abyte0[], int i, int j)
    throws IOException
public abstract boolean readBoolean()
    throws IOException
public abstract byte readByte()
    throws IOException
```

Figure 5.3: Methods of `DataInput` Interface

Code Snippet 5 displays the use of the `DataInput` interface.

Code Snippet 5:

```
try
{
    DataInputStream dis = new DataInputStream(System.in);
    double d = dis.readDouble();
    int num = dis.readInt();
}
catch (IOException e) { }
...
```

The code demonstrates the use of `readDouble()` and `readInt()` method to accept values from the user.

5.4.2 Methods in the DataOutput Interface

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

The important methods in this interface are:

→ **`writeBoolean(boolean b)`**

The `writeBoolean(boolean b)` method writes the boolean value given as parameter to an output stream. If the argument has the value as true then 1 will be written otherwise the value 0 is written.

```
public void writeBoolean(boolean b) throws IOException
```

→ **`writeByte(int value)`**

The `writeByte(int value)` method writes the byte value of the integer given as parameter to an output stream.

```
public void writeByte(int value) throws IOException
```

→ **`writeInt(int value)`**

The `writeInt(int value)` method writes four bytes that represent the integer given as parameter to an output stream.

```
public void write(int value) throws IOException
```

→ **`writeDouble(double value)`**

The `writeDouble(double value)` method writes eight bytes that represent the double value given as parameter to an output stream.

```
public void writeDouble(double value) throws IOException
```

→ **`writeChar(int value)`**

The `writeChar(int value)` method writes the char value of the integer given as parameter to a stream.

```
public void writeChar(int value) throws IOException
```

→ **`writeChars(String value)`**

The `writeChars(String value)` method writes the string given as parameter to a stream.

```
public void writeChars(String s) throws IOException
```

→ **writeUTF(String value)**

The `writeUTF(String value)` method writes a string in Java modified UTF-8 form given as parameter to a stream.

```
public void writeUTF(String str) throws IOException
```

Figure 5.4 displays the methods of `DataOutput` interface.

```
public abstract void writeChars(String s)
    throws IOException
public abstract void writeDouble(double d)
    throws IOException
public abstract void writeFloat(float f)
    throws IOException
public abstract void writeInt(int i)
    throws IOException
public abstract void writeLong(long l)
    throws IOException
public abstract void writeShort(int i)
    throws IOException
public abstract void writeUTF(String s)
    throws IOException
```

Figure 5.4: Methods of DataOutput Interface

Code Snippet 6 displays the use of `DataOutput` interface.

Code Snippet 6:

```
try
{
    outStream.writeBoolean(true);
    outStream.writeDouble(9.95);
    ...
}
catch (IOException e) {}
...
```

5.5 java.io Package

A stream represents many sources and destinations, such as disk files and memory arrays. It is a sequence of data. An I/O Stream represents an input source or an output destination.

Streams support many forms of data, such as simple bytes, primitive date type, localized characters, and

so on. Certain streams allow data to pass and certain streams transform the data in an useful way.

However, all streams provide a simple model to programs to use them. A program uses an input stream to read data from a source. It reads one item at a time.

Figure 5.5 illustrates this.

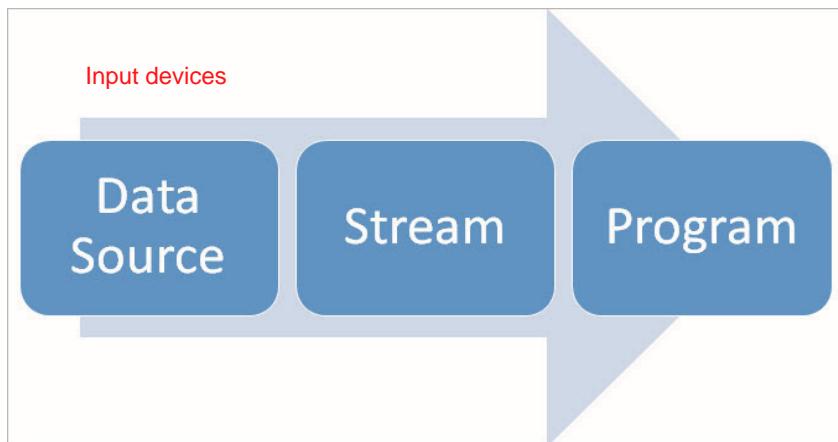


Figure 5.5: Input Stream Model

A program uses an output stream to write data to a destination. It writes one item at time.

Figure 5.6 illustrates this.

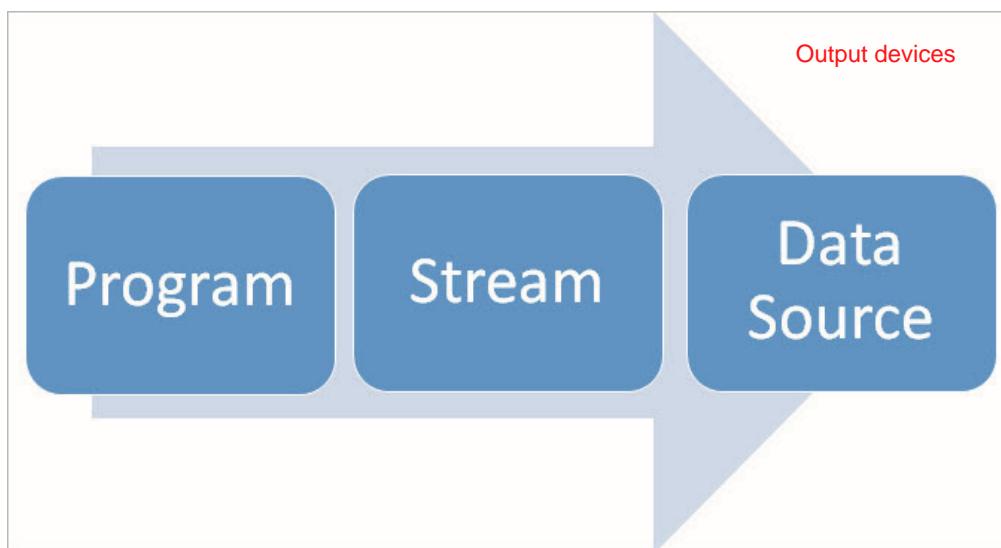


Figure 5.6: Output Stream Model

Note - Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes descend from `InputStream` and `OutputStream`.

There are many byte stream classes. They work in the same way but differ in construction.

Always put I/O procedure in the Try-catch exception

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

Code Snippet 7:

```

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

```

In Code Snippet 7, `read()` method reads a character and returns an `int` value. This allows the `read()` method to indicate that the end of the stream is reached by returning a value of `-1`.

When a stream is no longer required, it is important to `close the stream`. This helps to `avoid resource leaks`.

The Java platform uses Unicode conventions to store character values. Character stream I/O translates this format to and from the local character set.

Note - In Western locales, the local character set is typically an 8-bit superset of ASCII.

For most applications, input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams adapts to the local character set and is ready for internationalization.

All character stream classes are derived from the Reader and Writer class. There are character stream classes that specialize in file I/O operations such as `FileReader` and `FileWriter`.

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

Code Snippet 8:

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

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

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

Character I/O typically occurs in bigger units than single characters, such as a line that includes a string of characters with a line terminator at the end.

A line terminator can be any one of the following:

- Carriage-return/line-feed sequence (“\r\n”)
- A single carriage-return (“\r”)
- A single line-feed (“\n”).

`BufferedReader.readLine()` and `PrintWriter.println()` method helps input and output of one line at a time. The `readLine()` method returns a line of text with the line. The `println()` method outputs each line on a new line as it appends the line terminator for the current operating system. Note that the line terminators in the input and output file can differ.

5.6 ByteStream – `InputStream` Class and Its Subclasses

`InputStream` class is an abstract class that defines how streams receive data and is the superclass of all stream classes. This class has methods to read bytes or array of bytes, mark locations in the stream, find out number of bytes that has been read or available for reading and so on. This class is used to read data from an input stream.

Figure 5.7 shows the `InputStream` class hierarchy.

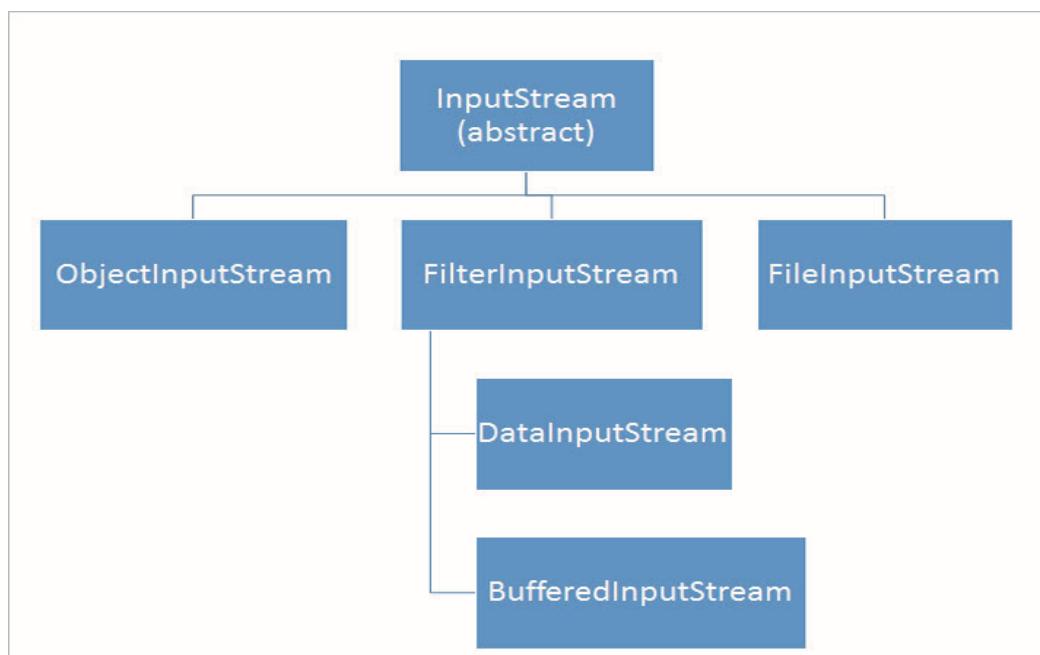


Figure 5.7: `InputStream` Class Hierarchy

5.6.1 Methods of InputStream Class

The `InputStream` class provides a number of methods that manage reading data from a stream.

Some of the methods of this class are as follows:

→ **`read()`**

The `read()` method reads the next bytes of data from the input stream and returns an `int` value in the range of 0 to 255. The method returns -1 when end of file is reached.

```
public abstract int read() throws IOException
```

→ **`available()`**

The `available()` method returns the number of bytes that can be read without blocking. In other words, it returns the number of available bytes.

```
public int available() throws IOException
```

→ **`close()`**

The `close()` method closes the input stream. It releases the system resources associated with the stream.

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

→ **`mark(int n)`**

The `mark(int n)` method marks the current position in the stream and will remain valid until the number of bytes specified in the variable, `n`, is read. A call to the `reset()` method will position the pointer to the last marked position.

```
public void mark(int readlimit)
```

→ **`skip(long n)`**

The `skip(long n)` method skips `n` bytes of data while reading from an input stream.

```
public long skip(long n) throws IOException
```

→ **`reset()`**

The `reset()` method **rests the reading pointer to the previously set mark** in the stream.

```
public void reset() throws IOException
```

5.6.2 FileInputStream Class

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

`reset()` method will generate an `IOException`.

Commonly used constructors of this class are as follows:

→ **`FileInputStream(String sObj)`**

The `FileInputStream(String sObj)` creates an `InputStream` object that can be used to read bytes from a file. The parameter `sObj` stores the full path name of a file.

```
public FileInputStream(String sObj) throws FileNotFoundException
```

→ **`FileInputStream(File fObj)`**

The `FileInputStream(File fObj)` creates an `InputStream` object that can be used to read bytes from a file where `fObj` is a `File` object.

```
public FileInputStream(File fObj) throws FileNotFoundException
```

→ **`FileInputStream(FileDescriptor fdObj)`**

The `FileInputStream(FileDescriptor fdObj)` creates a `FileInputStream` using the file descriptor object that can be used to represent an existing connection to the file which is there in the file system. The file descriptor object is `fdObj`.

```
public FileInputStream(FileDescriptor fdObj) throws FileNotFoundException
```

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

Code Snippet 9:

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

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

The code in example creates a `FileInputStream` object to which the filename is passed as an argument. The object is used to read the text characters from the specified file. The program prints out its own source code.

Code Snippet 10:

```

import java.io.FileInputStream;
import java.io.IOException;

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

```

Figure 5.8 displays the output.

```

Output - JavaOApplication (run) ✘ Inspector
public class Client {

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

```

Figure 5.8: FileInputStream Class - Output

5.6.3 ByteArrayInputStream Class

ByteArrayInputStream contains a **buffer that stores the bytes** that are read from the stream. ByteArrayInputStream class uses a byte array as the source. ByteArrayInputStream class has

an internal counter, which keeps track of the next byte to be read. This class does not support any new methods. It only overrides the methods of the `InputStream` class such as `read()`, `skip()`, `available()`, and `reset()`.

→ **`protected byte[] buf`**

This refers to an array of bytes that is provided by the creator of the stream.

→ **`protected int count`**

This refers to the index greater than the last valid character in the input stream buffer.

→ **`protected int mark`**

This refers to the currently marked position in the stream.

→ **`protected int pos`**

This refers to the index of the next character to be read from the input stream buffer.

The constructors of this class are as follows:

→ **`ByteArrayInputStream(byte[] b)`**

The `ByteArrayInputStream(byte[] b)` creates a `ByteArrayInputStream` with a byte array, `b`, as the input source. In other words, it uses `b` as its buffer array.

```
public ByteArrayInputStream(byte[] b)
```

→ **`ByteArrayInputStream(byte[] b, int start, int num)`**

The `ByteArrayInputStream(byte[] b, int start, int num)` creates a `ByteArrayInputStream` with a byte array, `b`, as the input source. It begins with the character at the index specified by `start` and is `num` bytes long.

```
public ByteArrayInputStream(byte[] b, int start, int num)
```

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

Code Snippet 11:

```
...
String content = "Hello World";
Byte [] bObj = content.getBytes();
ByteArrayInputStream inputByte = new ByteArrayInputStream(bObj);
...
```

5.7 OutputStream Class and its Subclasses

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

5.7.1 Methods in OutputStream Class

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

Some of the methods of this class are as follows:

→ `write(int b)`

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

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

→ `write(byte[] b)`

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

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

→ `write(byte[] b, int off, int len)`

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

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

→ `flush()`

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

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

→ `close()`

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

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

5.7.2 FileOutputStream Class

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

Certain platforms allow only one file-writing object to open a file for writing. Therefore, if the file is already open, the constructors in the class fail.

An `IOException` will be thrown only when a read-only file is opened.

Some of the commonly used constructors of this class are as follows:

→ **`FileOutputStream(String filename)`**

The `FileOutputStream(String filename)` method creates an output file stream object that is used to write bytes to a file. Here, `filename` is full path name of a file.

```
public FileOutputStream(String filename) throws FileNotFoundException
```

→ **`FileOutputStream(File name)`**

The `FileOutputStream(File name)` method creates an `FileOutputStream` object that can be used to write bytes to a file. Here, `name` is a `File` object that describes the file.

```
public FileOutputStream(File name) throws FileNotFoundException
```

→ **`FileOutputStream(String filename, boolean flag)`**

The `FileOutputStream(String filename, boolean flag)` method creates an `FileOutputStream` object that can be used to write bytes to a file. The `filename` is a `File` object. If `flag` is true, the file is opened in append mode. false -> replace

```
public FileOutputStream(String filename, boolean flag) throws FileNotFoundException
```

→ **`FileOutputStream(File name, boolean flag)`**

The `FileOutputStream(File name, boolean flag)` method creates an `FileOutputStream` object that can be used to write bytes to a file. Here, `name` is a `File` object. If `flag` is true, the file is opened in append mode.

```
public FileOutputStream(File name, boolean flag) throws FileNotFoundException
```

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

Code Snippet 12:

```

...
String temp = "One way to get the most out of life is to look upon it as an adventure."
byte [] bufObj = temp.getBytes();
OutputStream fileObj = new FileOutputStream("Thought.txt");
fileObj.write(bufObj);
fileObj.close();
...

```

Code Snippet 12 first stores the content of the String variable in the byte array, **bufObj**, using the `getBytes()` method. Then the entire content of the byte array is written to the file, `Thought.txt`.

5.7.3 ByteArrayOutputStream Class

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

`ByteArrayOutputStream` class defines two constructors.

The constructors can accept an integer argument or not.

The constructors of this class are as follows:

→ **ByteArrayOutputStream()**

The constructor, `ByteArrayOutputStream()`, initialises a new `ByteArrayOutputStream` object and sets the size of the buffer to a default size of 32 bytes.

```
public ByteArrayOutputStream()
```

→ **ByteArrayOutputStream(int size)**

The constructor, `ByteArrayOutputStream(int size)` initializes a new `ByteArrayOutputStream` object and sets the size of the buffer to the specified size.

```
public ByteArrayOutputStream(int size)
```

5.7.4 Methods in ByteArrayOutputStream Class

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

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

→ **`reset()`**

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

```
public void reset()
```

→ **`size()`**

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

```
public int size()
```

→ **`toByteArray()`**

The `toByteArray()` method creates a newly allocated byte array containing the bytes that have been written to this stream so far.

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

→ **`writeTo(OutputStream out)`**

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

```
public void writeTo(OutputStream out) throws IOException
```

→ **`toString()`**

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

```
public String toString()
```

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

Code Snippet 13:

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

In Code Snippet 13, a `ByteArrayOutputStream` object is created and then the content from the byte

array is written to the `ByteArrayOutputStream` object. Finally, the content from the output stream is converted to a string using the `toString()` method and displayed.

5.8 Filter Streams

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

5.8.1 FilterInputStream Class

filterstream combine with stream but cannot combine with input resources

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

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

use filter streams when don't want to convert byte into char

- **protected InputStream in:** This input stream needs to be filtered.
- **protected FilterInputStream(InputStream in):** This creates a `FilterInputStream`. The argument is assigned to the field `this.in` and can be recalled anytime.

Following are the methods of this class:

- **mark(int readlimit):**

This method identifies the current position in the input stream.

```
void mark(int readlimit)
```

- **markSupported():**

This method checks if the input stream supports the mark and reset methods.

```
boolean markSupported()
```

- **read():**

This method reads the next byte of data from the input stream.

```
int read()
```

- **available():**

This method returns an approximation of bytes that can be read or skipped from the input stream.

```
int available()
```

→ **close():**

This method closes the input stream and releases any system resources related with the stream.

```
void close()
```

→ **read(byte[] b):**

This method reads `byte.length` bytes of data from the input stream into an array of bytes.

```
int read(byte[] b)
```

→ **reset():**

This method repositions the pointer to the position in the stream when the mark method was last invoked on the input stream.

```
void reset()
```

→ **skip(long n):**

This method skips and discards n bytes of data from the input stream.

```
long skip(long n)
```

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

This method reads len bytes of data from the input stream into an array of bytes.

```
int read(byte[] b, int off, int len)
```

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

Code Snippet 14:

```
package javaioapplication;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FilterInputStream;
import java.io.IOException;
import java.io.InputStream;

public class FilterInputApplication {
    public static void main(String[] args) throws Exception {
        InputStream inputObj = null;
        FilterInputStream filterInputObj = null;
        try {
```

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

Figure 5.9 displays the output.

```
Output - JavaIOApplication (run) ✘ Inspector
run:
H
e
mark() invoked
l
l
reset() invoked
l
l
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 5.9: FilterInputStream Class - Output

5.8.2 FilterOutputStream Class

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

The `java.io.FilterOutputStream` class includes the protected `OutputStream out` field, which is the output stream to be filtered.

`FilterOutputStream(OutputStream out)` is the constructor of this class. This creates an output stream filter that exist class over the defined output stream.

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

Code Snippet 15:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;
public class FilterOutputStreamApplication {
    public static void main(String[] args) throws Exception {
        OutputStream outputStreamObj = null;
        FilterOutputStream filterOutputStreamObj = null;
```

```
FileInputStreamfilterInputStreamObj=null;
byte[] bufObj={81, 82, 83, 84, 85};

int i=0;
char c;

//encloses the creation of stream objects within try-catch block
try{

    // creates output stream objects
    OutputStreamObj=new FileOutputStream("C:/Java/test.txt");
    filterOutputStreamObj=new FilterOutputStream(OutputStreamObj);

    // writes to the output stream from bufObj
    filterOutputStreamObj.write(bufObj);

    // forces the byte contents to be written to the stream
    filterOutputStreamObj.flush();      make sure the data is completely written before it's closed

    // creates an input stream object
    filterInputStreamObj=new FileInputStream("C:/Java/test.txt");

    while((i=filterInputStreamObj.read())!=-1)

    {      // converts integer to character
        c=(char)i;

        // prints the character read
        System.out.println("Character read after conversion is: "+c);

    }

} catch(IOException e){

    // checks for any I/O errors
    System.out.print("Close() is invoked prior to write()");
}

finally{

    // releases system resources associated with the stream
    if(OutputStreamObj!=null)
        OutputStreamObj.close();
    if(filterOutputStreamObj!=null)
        filterOutputStreamObj.close();
}
```

Figure 5.10 displays the output.

The screenshot shows the 'Output' window of a Java IDE during the execution of a program named 'JavaIOApplication'. The window title is 'Output - JavaIOApplication (run)'. It contains the following text:

```

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

```

Figure 5.10: FilterOutputStream Class - Output

5.9 Buffered Streams

use for read and write in many files at the same time

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

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

Filters operate on the buffer, which is located between the program and the destination of the buffered stream. Figure 5.11 displays the Object class hierarchy.

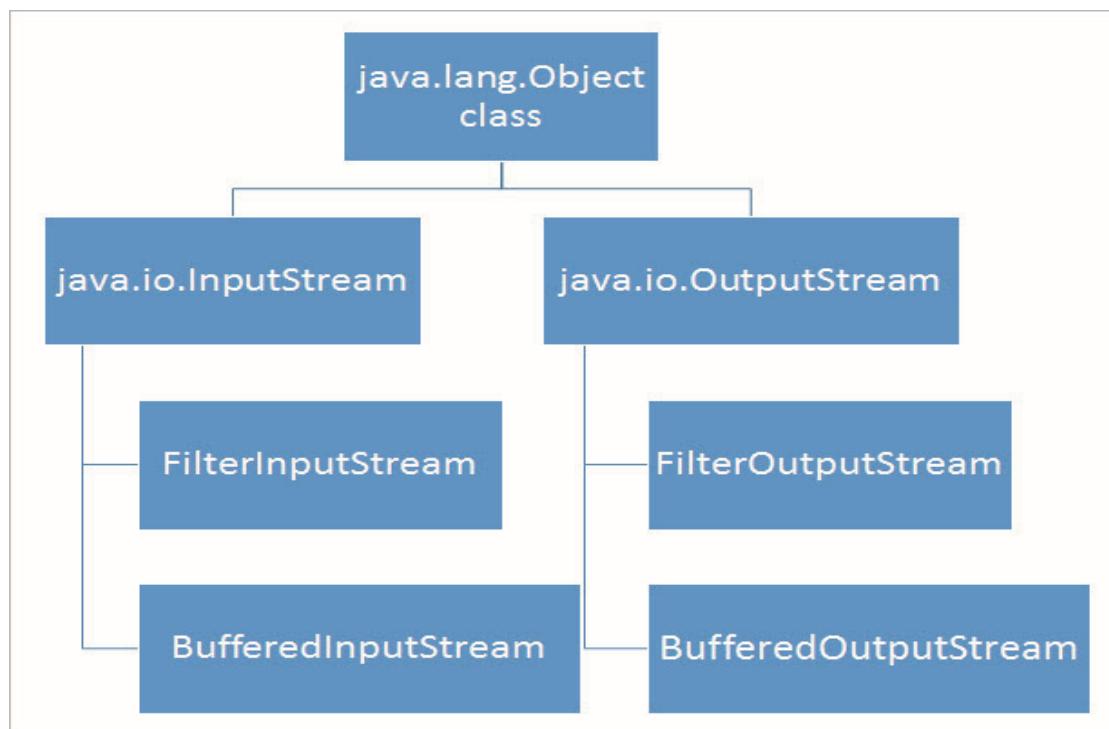


Figure 5.11: java.lang.Object Class Hierarchy

5.9.1 *BufferedInputStream Class*

`BufferedInputStream` class allows the programmer to wrap any `InputStream` class into a buffered stream. The `BufferedInputStream` act as a cache for inputs. It does so by creating the array of bytes which are utilized for future reading.

The simplest way to read data from an instance of `BufferedInputStream` class is to invoke its `read()` method. `BufferedInputStream` class also supports the `mark()` and `reset()` methods. The function `markSupported()` will return true if it is supported.

`BufferedInputStream` class defines two constructors:

→ **`BufferedInputStream(InputStream in)`**

The constructor creates a buffered input stream for the specified `InputStream` instance. The default size of the buffer is 2048 byte.

```
public BufferedInputStream(InputStream in)
```

→ **`BufferedInputStream(InputStream in, int size)`**

The constructor creates a buffered input stream of a given size for the specified `InputStream` instance.

```
public BufferedInputStream(InputStream in, int size)
```

Some of the methods of this class are listed in table 5.2.

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

Table 5.2: Methods of `BufferedInputStream` Class

Code Snippet 16 demonstrates the `BufferedInputStream` class.

Code Snippet 16:

```
...
String temp = "This is an example";
byte [] bufObj = temp.getBytes ();
ByteArrayInputStream fileObj = new ByteArrayInputStream (bufObj);
BufferedInputStream inObj = new BufferedInputStream (fileObj);
...
```

5.9.2 *BufferedOutputStream Class*

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

The constructors of this class are as follows:

→ **`BufferedOutputStream(OutputStream os)`**

The `BufferedOutputStream(OutputStream os)` creates a buffered output stream for the specified `OutputStream` instance with a default size of 512 bytes.

```
public BufferedOutputStream(OutputStream os)
```

→ **`BufferedOutputStream(OutputStream os, int size)`**

The `BufferedOutputStream(OutputStream os, int size)` creates a buffered output stream of a given size for the specified `OutputStream` instance.

```
public BufferedOutputStream(OutputStream os, int size)
```

Table 5.3 lists the method of the `BufferedOutputStream` class.

Method	Description
<code>void flush()</code>	The method flushes the buffered output stream.

Table 5.3: `BufferedOutputStream` Class

5.10 Character Streams

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

support Unicode characters and can be internationalized. Reader and Writer are abstract classes at the top of the class hierarchy that supports reading and writing of Unicode character streams. All character stream class are derived from the `Reader` and `Writer` class. Figure 5.12 displays the character stream classes.

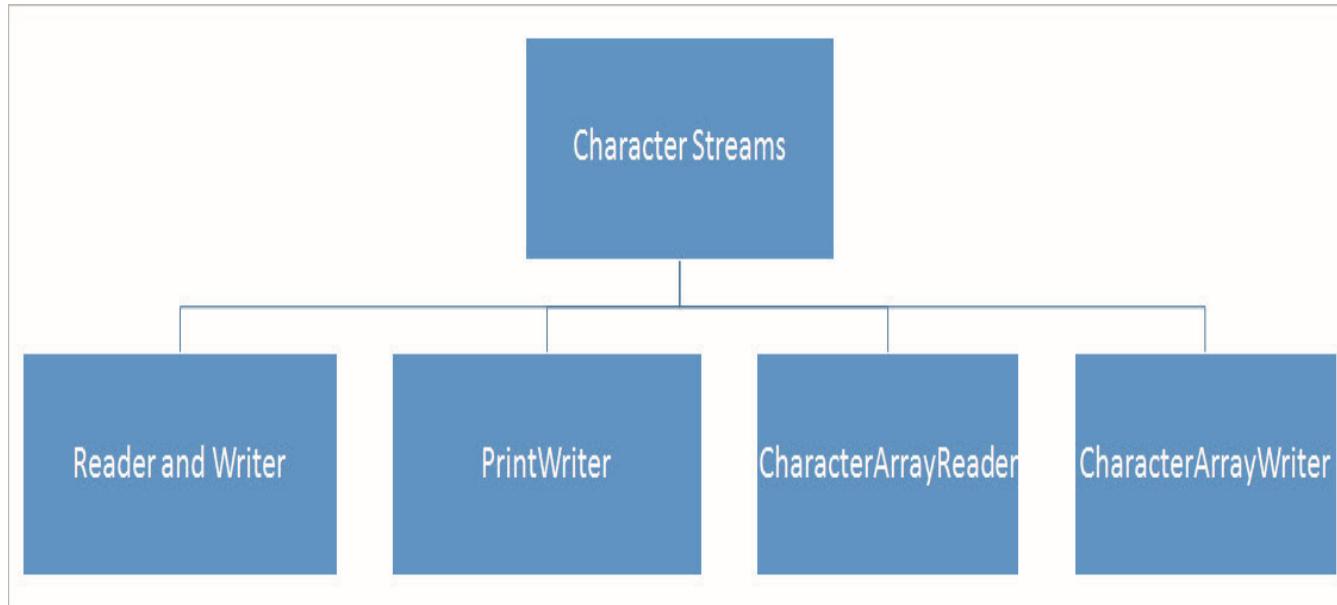


Figure 5.12: Character Streams

5.10.1 Reader Classes

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

There are two constructors for the `Reader` class. They are as follows:

→ **`Reader()`**

The constructor creates a character stream reader and synchronizes the reader and the critical section of the new character stream reader.

```
protected Reader()
```

→ **`Reader(Object lock)`**

The constructor creates a character stream reader and synchronizes new character-stream reader with the given object.

```
protected Reader(Object lock)
```

Some of the methods of the Reader class are listed in table 5.4.

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

Table 5.4: Methods of the Reader Class

5.10.2 Writer Classes

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

The constructors for the `Writer` class are as follows:

→ `Writer()`

The `Writer()` constructor create a new character-stream writer whose critical sections will synchronize on the writer itself.

```
protected Writer()
```

→ **Writer(Object lock)**

The `Writer(Object lock)` constructor creates a new character-stream writer whose critical sections will synchronize on the given object.

```
protected Writer(Object lock)
```

Some of the methods of this class are listed in table 5.5.

Method	Description
<code>void write(int c) throws IOException</code>	The method reads a single character.
<code>void write(char[] text) throws IOException</code>	The method writes a complete array of characters to an output stream.
<code>void write(char[] text, int offset, int length)</code>	The method writes a portion of an array of characters to an output stream starting from offset. The variable length specifies the number of characters to write.
<code>void write(String s) throws IOException</code>	The method writes a string to the output stream.
<code>void write(String s, int offset, int length) throws IOException</code>	The method writes a portion of a string to the output stream starting from offset. The variable length specifies the number of characters to write.
<code>void flush() throws IOException</code>	The method flushes the output stream so that buffers are cleared.
<code>void close() throws IOException</code>	The method closes the output stream.

Table 5.5: Methods of Writer Class

5.10.3 **PrintWriter Class** =printstream

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

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

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

The constructors for PrintWriter class are as follows:

→ **`PrintWriter(OutputStream out)`**

The `PrintWriter(OutputStream out)` constructor creates a new `PrintWriter` from an existing `OutputStream`. It does not support automatic line flushing.

```
public PrintWriter(OutputStream out)
```

→ **`PrintWriter(OutputStream out, boolean autoFlush)`**

The `PrintWriter(OutputStream out, boolean autoFlush)` constructor creates a new `PrintWriter` from an existing `OutputStream`. A boolean value of true will enable the `println()` or `printf()` method to flush the output buffer.

```
public PrintWriter(OutputStream out, boolean autoFlush)
```

→ **`PrintWriter(Writer out)`**

The `PrintWriter(Writer out)` constructor creates a new `PrintWriter`, and does not allow automatic line flushing.

```
public PrintWriter(Writer out)
```

→ **`PrintWriter(Writer out, boolean autoFlush)`**

The `PrintWriter(Writer out, boolean autoFlush)` constructor creates a new `PrintWriter`. A boolean value of true will enable to flush the output buffer.

```
public PrintWriter(Writer out, boolean autoFlush)
```

The main advantage of the `print()` and `println()` method is that any Java object or literal or variable can be printed by passing it as an argument. If the `autoFlush` option is set to true then automatic flushing takes place when `println()` method is invoked. The `println()` method follows its argument with a platform dependent line separator. The `print()` method does not flush the stream automatically. Otherwise, both these methods are same. Some of the methods of this class are listed in table 5.6.

Method	Description
<code>boolean checkError()</code>	The method flushes the stream if it's open and check the error state.
<code>void print(boolean b)</code>	The method prints a boolean value.
<code>void print(char c)</code>	The method is used to print a character.
<code>void print(char[] s)</code>	The method is used to print an array of characters.
<code>void print(double d)</code>	The method prints a double-precision floating-point number.
<code>void print(float f)</code>	The method prints a floating-point number.
<code>void print(int i)</code>	The method is used to print an integer.
<code>void print(long l)</code>	The method is used to print a long integer.

Method	Description
void print(Object obj)	The method prints an object and calls the <code>toString()</code> method on the argument.
void print(String s)	The method prints a string.
void println()	The method terminates the current line by using the line separator string.
void println(boolean x)	The method prints a boolean value and then terminate the line.
void flush() throws IOException	The method flushes the output stream so that buffers are cleared.
void close() throws IOException	The method closes the output stream.
void setError()	The method indicates that an error has occurred.
void write(char[] buf)	The method is used for writing an array of characters.
void write(char[] buf, int off, int len)	The method is used for writing a part of an array of characters.
void write(int c)	The method writes a single character.
void write(String s)	The method writes a string.
void write(String s, int off, int len)	The method writes a part of a string.

Table 5.6: Methods of PrintWriter Class

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

Code Snippet 17 displays the use of the `PrintWriter` class.

Code Snippet 17:

```
...
InputStreamReader reader = new InputStreamReader (System.in);
OutputStreamWriter writer = new OutputStreamWriter (System.out);
PrintWriter pwObj = new PrintWriter (writer,true);
...
try
{
while (tmp != -1)
```

```

{
    tmp = reader.read ();
    ch = (char) tmp;
    pw.println ("echo " + ch);
}
}

catch (IOException e)
{
    System.out.println ("IO error:" + e );
}
.
.
```

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

5.10.4 CharArrayReader Class =ByteArrayReader

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

The constructors of this class are as follows:

→ **CharArrayReader(char arr[])**

The `CharArrayReader(char arr[])` constructor creates a `CharArrayReader` from the specified character array, `arr`.

→ **CharArrayReader(char arr[], int start, int num)**

The `CharArrayReader(char arr[], int start, int num)` constructor creates a `CharArrayReader` from a subset of the character array, `arr`, starting from the character specified by the index, `start`, and is `num` characters long.

Some of the methods of this class are listed in table 5.7.

Method	Description
<code>long skip(long n)</code>	The method skips <code>n</code> number of characters before reading.
<code>void mark(int num)</code>	The method marks the current position in the stream.
<code>int read()</code>	The method reads a single character.
<code>int read(char[] b, int off, int length)</code>	The method reads characters into the specified character array from the given offset.

Method	Description
void reset()	The method repositions the pointer in the stream to the point where the mark method was last called or to the beginning of the stream if it has not been marked.
boolean ready()	The method is used to confirm whether this stream is ready to be read.
void close()	The method closes the input stream.

Table 5.7: Methods of CharArrayReader Class

Code Snippet 18 displays the use of the `CharArrayReader` class.

Code Snippet 18:

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

In Code Snippet 18, the content of the entire string is stored as a series of characters in the character array, `ch`, by using the `getChars()` method. Next, an instance of `CharArrayReader` is created and initialized with the first five characters from the array.

5.10.5 CharArrayWriter Class

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

The constructors of this class are as follows:

→ **CharArrayWriter()**

The `CharArrayWriter()` constructor creates a `CharArrayWriter` with a buffer having a default size of 32 characters.

→ **CharArrayWriter(int num)**

The `CharArrayWriter(int num)` constructor creates a `CharArrayWriter` with a buffer of size specified by the variable `num`.

Some of the methods of this class are listed in table 5.8.

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

Table 5.8: Methods of CharArrayWriter Class

Code Snippet 19 displays the use of the `CharArrayWriter` class.

Code Snippet 19:

```

. . .
CharArrayWriter fObj = new CharArrayWriter();

. . .

String temp = "Hello World";

int size = temp.length();
char [] ch = new char[size];
temp.getChars(0, temp.length(), ch, 0);

fObj.write(ch);
char[] buffer=fObj.toCharArray();
System.out.println(buffer);
System.out.println(fObj.toString());
. .

```

In Code Snippet 19, the content of the entire string is stored as a series of characters in the character array, ch, by using the `getChars()` method. Next, the instance of `CharArrayWriter` contains the content from the character array. The `toCharArray()` method is used to store the content of the `CharArrayWriter` in a character array. Finally, the content is printed.

Code Snippet 20 demonstrates the use of `CharArrayWriter` class.

Code Snippet 20:

```
import java.io.CharArrayWriter;
import java.io.IOException;
public class Program {
    public static void main(String[] args) throws IOException {
        // Create a CharArrayWriter object which can hold 11 characters.
        CharArrayWriter writer = new CharArrayWriter(11);
        String str ="Hello Aptech";
        writer.write("Hello Aptech", 6, str.length() - 6);
        System.out.println("The CharArrayWriter buffer contains: " + writer.
toString());
        writer.flush();
        // Print out the contents of the CharArrayWriter buffer.
        System.out.println("After flushing the CharArrayWriter buffer " +
contains: " + writer.toCharArray());
        // Now reset the buffer we just populated.
        writer.reset();
        // Print out the contents of the CharArrayWriter buffer.
        System.out.println("After reset CharArrayWriter buffer " +
contains: " + writer.
toCharArray());
```

```
// Close the CharArrayWriter and StringWriter buffers.  
writer.close();  
}  
}
```

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

5.11 Chaining I/O Systems

A program, typically, uses a series of streams to process the data. Figure 5.13 illustrates this.

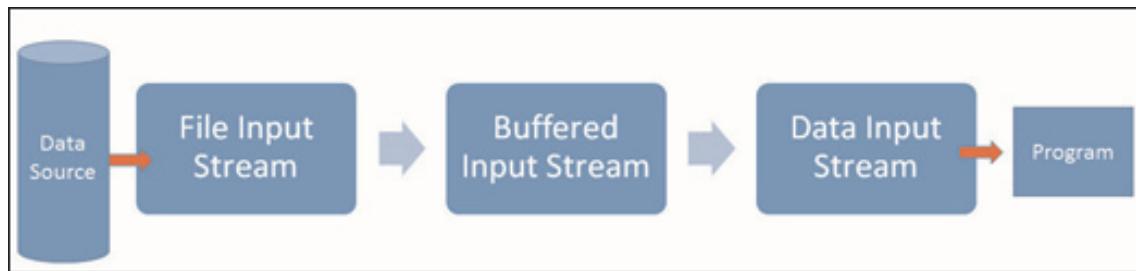


Figure 5.13: Input Stream Chain

In figure 5.13, an input stream is chained where a file stream is buffered. It is then converted into Java data items.

Note - The file stream is buffered for efficiency.

Figure 5.14 displays the chaining of an output stream. Here, it is first written to a buffer and then to a file.

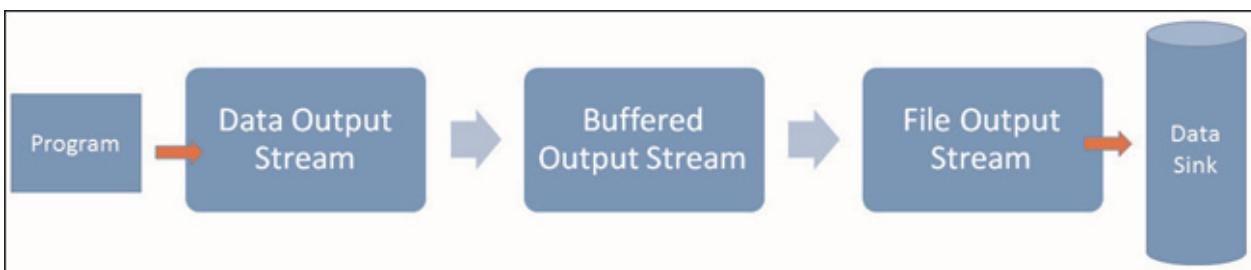


Figure 5.14: Output Stream Chain

A processing stream executes a conversion on another stream. A user can select the stream type depending on the required functionality for the final stream. Table 5.9 displays the functionalities and their respective streams.

Functionality	Character Streams	Byte Streams
Buffering (Strings)	BufferedReader	BufferedInputStream
	BufferedWriter	BufferedOutputStream
Filtering	FilterReader	FilterInputStream
	FilterWriter	FilterOutputStream
Object serialization		ObjectInputStream
		ObjectOutputStream
Data conversion		DataInputStream
		DataOutputStream
Conversion (byte to character)	InputStreamReader	OutputStreamWriter
Counting	LineNumberReader	LineNumberInputStream
Peeking ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

Table 5.9: Functionalities and Streams

5.12 Serialization

Persistence is the process of saving data to some permanent storage. A persistent object can be stored on disk or sent to some other machine for saving its data. On the other hand, a non persistent object exists as long as the JVM is running, Java is an object-oriented language and thus provides the facilities for reading and writing object. **Serialization is the process of reading and writing objects to a byte stream.**

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

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

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

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

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

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

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

5.12.1 *ObjectOutputStream Class*

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

The constructors of this class are as follows:

→ **`ObjectOutputStream()`**

The `ObjectOutputStream()` constructor prevents the subclasses that are completely reimplementing `ObjectOutputStream` from allocating private data just used by this implementation of `ObjectOutputStream`.

Its signature is as follows:

```
protected ObjectOutputStream() throws IOException, SecurityException
```

→ **`ObjectOutputStream(OutputStream out)`**

The `ObjectOutputStream(OutputStream out)` constructor creates an `ObjectOutputStream` that writes to the specified `OutputStream`. Its signature is as follows:

```
public ObjectOutputStream(OutputStream out) throws IOException
```

5.12.2 *Methods in ObjectOutputStream Class*

The methods in `ObjectOutputStream` class helps to write objects to the output stream. The methods in `ObjectOutputStream` class are as follows:

→ **`writeFloat(float f)`**

The `writeFloat(float f)` method writes a float value to the output stream. Its signature is as follows:

```
public void writeFloat(float f) throws IOException
```

→ **writeObject (Object obj)**

The `writeObject (Object obj)` method writes an object, `obj`, to the output stream. Its signature is as follows:

```
public final void writeObject(Object obj) throws IOException
```

→ **defaultWriteObject()**

The `defaultWriteObject()` method writes non-static and non-transient fields into the underlying output stream. Its signature is as follows:

```
public void defaultWriteObject() throws IOException
```

Code Snippet 21 displays the use of methods of `ObjectOutputStream` class.

Code Snippet 21:

```
.
.
.
Point pointObj = new Point(50, 75);
FileOutputStream fObj = new FileOutputStream("point");
ObjectOutputStream oos = new ObjectOutputStream(fObj);
oos.writeObject(pointObj);
oos.writeObject(new Date());
oos.close();
.
.
```

In Code Snippet 21, an object of `FileOutputStream` is chained to an object output stream. Then to the object's output stream the `writeObject ()` method is invoked with an object as its argument.

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

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

5.12.3 ObjectInputStream Class

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

The constructors of this class are as follows:

→ **ObjectInputStream()**

The `ObjectInputStream()` constructor helps subclasses to re-implement `ObjectInputStream` to avoid allocation of private data used by the implementation of `ObjectInputStream`. Its signature is as follows:

```
protected ObjectInputStream() throws IOException, SecurityException
```

→ **ObjectInputStream(InputStream in)**

The `ObjectInputStream(InputStream in)` constructor creates an `ObjectInputStream` that reads from the specified `InputStream`. Serialized objects are read from input stream `in`.

Its signature is as follows:

```
public ObjectInputStream(InputStream in) throws IOException
```

5.12.4 Methods in `ObjectInputStream` Class

The method `in` `ObjectInputStream` helps to read object from the stream. Some of the methods in `ObjectInputStream` class are as follows:

→ **readFloat()**

The `readFloat()` method reads and returns a float from the input stream. Its signature is as follows:

```
public float readFloat() throws IOException
```

→ **readBoolean()**

The `readBoolean()` method reads and returns a boolean from the input stream. Its signature is as follows:

```
public boolean readBoolean() throws IOException
```

→ **readByte()**

The `readByte()` method reads and returns a byte from the input stream. Its signature is as follows:

```
public byte readByte() throws IOException
```

→ **readChar()**

The `readChar()` method reads and returns a char from the input stream. Its signature is as follows:

```
public char readChar() throws IOException
```

→ **readObject()**

The `readObject()` method reads and returns an object from the input stream. Its signature is as follows:

```
public final Object readObject() throws IOException,  
ClassNotFoundException
```

Code Snippet 22 displays the creation of an instance of `ObjectInputStream` class.

Code Snippet 22:

```
 . . .  
 FileInputStream fObj = new FileInputStream("point");  
 ObjectInputStream ois = new ObjectInputStream(fObj);  
 Point obj = (Point) ois.readObject(); Cast ois into the type of obj  
 ois.close();
```

In Code Snippet 22, an instance of `FileInputStream` is created that refers to the file named `point`. An `ObjectInputStream` is created from that file stream. The `readObject()` method returns an object which deserialize the object. Finally, the object input stream is closed.

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

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

Code Snippet 23 demonstrates the `Serializable` interface.

Code Snippet 23:

```
import java.io.Serializable;  
  
public class Employee implements Serializable{  
    String lastName;  
    String firstName;  
    double sal;  
}  
  
public class BranchEmpProcessor {
```

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

Figure 5.15 shows the output of the code.

```
run:  
Deserialized - John Smith from NewEmployee.ser  
finally  
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 5.15: Output of BranchEmpProcessor Class

5.13 Check Your Progress

1. Which of the following are true about stream classes?

(A)	They help read input from a stream.		
(B)	Input and Output streams are used for reading and writing of structured sequence of bytes.		
(C)	They help manage disk files.		
(D)	All classes that work on streams directly works with files and the file system.		

(A)	A, C	(C)	C
(B)	B	(D)	D

2. The `File(String dirpath)` constructor _____.

(A)	Creates a <code>File</code> object with pathname of the file specified by the String variable <code>dirpath</code> into an abstract pathname.		
(B)	Creates a <code>File</code> object with pathname of the file specified by the String variables <code>parent</code> and <code>child</code> .		
(C)	Creates a new <code>File</code> instance from another <code>File</code> object specified by the variable <code>fileObj</code> .		
(D)	Converts the given <code>file: URI</code> into an abstract pathname and creates a new <code>File</code> instance.		

(A)	A	(C)	C
(B)	B	(D)	D

3. Which of the following classes create a `FileInputStream` or `FileOutputStream` to contain it?

(A)	System		
(B)	File		
(C)	FileDescriptor		
(D)	FilenameFilter		

(A)	A	(C)	C
(B)	B	(D)	D

4. The sync method _____.

(A)	Creates an (invalid) FileDescriptor object
(B)	Checks whether the file descriptor is valid
(C)	Reads bytes from a binary stream
(D)	Clears the system buffers and writes the content that they contain to the actual hardware

(A)	A	(C)	C
(B)	B	(D)	D

5. Identify the methods in the DataInput interface.

(A)	delete()
(B)	renameTo()
(C)	toPath()
(D)	readBoolean()

(A)	A	(C)	C
(B)	B	(D)	D

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

(A)	Serialization
(B)	Persistence
(C)	Deserialization
(D)	Append mode

(A)	A	(C)	C
(B)	B	(D)	D

5.13.1 Answers

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

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
        String str;
        try {
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            str = br.readLine();
            System.out.println("You entered: " + str);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Summary

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

Are you looking for online **HELP?**

We are just a *click* away



To chat with a

Login to **www.onlinevarsity.com**

Session 6

New Features in File Handling

Welcome to the Session, **New Features in File Handling**.

This session describes the `Console` class and the classes that help to compress and decompress data. The session elaborates on the `java.nio` package that helps to enhance the input/output processing tasks in the Java application development. Finally, the session elaborates on file systems, paths, and files.

In this Session, you will learn to:

- Describe the `Console` class
- Explain the `DeflaterInputStream` class
- Explain the `InflaterOutputStream` class
- Describe the `java.nio` package
- Describe the file system



6.1 Introduction

NIO in Java stands for New Input/Output operation. It is a collection of Java APIs that offers intensive I/O operations. The `java.nio.file` package provides comprehensive support for input and output operations. There are many classes present in the API, but in this session, only some of them will be discussed. The `java.nio` package mainly defines buffers which are containers for data. This API is easy to use.

6.2 Console Class

console cannot implement in the Netbeans, Eclipse

Java SE 6 has introduced the `Console` class to enhance and simplify the development of command line applications. Consider a scenario, where as a software developer you would not like to display the characters typed by the user for a console based application such as user login. Earlier, this was possible only with GUI based application. Java SE 6 came up with a solution by introducing the `Console` class. The `Console` class is a part of `java.io` package that has the ability to read text from the terminal without echoing on the screen. The `Console` object provides input and output of character streams through its `Reader` and `Writer` class.

The `Console` class provides various methods to access character-based console device. These devices should be associated with the current virtual machine. Whether a virtual machine has a console depends upon the manner the virtual machine was invoked. Automatic invocation of virtual machine will not have a console associated with it.

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

The `Console` class provides methods to perform input and output operations on character streams. The `readLine()` method reads a single line of text from console. The `readPassword()` method reads password from the console without echoing on the screen. The method returns a character array and not a `String` object to enable modification of the password. The password is removed from the memory, once it is no longer required.

Table 6.1 lists various methods available in the `Console` class.

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

Method	Description
readLine(String fmt, Object... args)	The method provides formatted output and accepts one line of text from the console

Table 6.1: Methods of Console Class

Code Snippet 1 shows the use of `Console` class methods.

Code Snippet 1:

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

The code accepts user name and password from the user through a console using the `readLine()` and `readPassword()` methods. The `System.console()` method returns a `Console` object if it is available that reads the username and password.

6.3 Classes in `java.util.zip`

Java SE 6 introduced a few changes in JAR and ZIP files. In the earlier versions of Java JAR files, the timestamp (date and time) of extracted files used to be set to the current time instead of archived file time. In Java SE 6, the JAR tool has been enhanced so that the timestamp of extracted files match the archive time. This is necessary because other decompression tools depend on the archive time not the current time for extracting files.

The upper limit of concurrently opening 2306 ZIP files in Windows now has been removed. The ZIP file names can be more than 256 characters long.

After using the ZIP files on daily basis, the important files should be archived and compressed to conserve valuable disk space. Files can be compressed and decompressed using popular utilities, such as WinRAR and WinZip.

Java in its `java.util.zip` package provides classes that can compress and decompress files.

Table 6.2 lists some of the classes in `java.util.zip` package along with their description.

Class Name	Description
CheckedInputStream	Maintains checksum of data that is being read
CheckedOutputStream	Maintains checksum of data that is to be written
Deflater	Performs data compression
DeflaterInputStream	Reads source data and then compresses it in the “deflate” compression format
DeflaterOutputStream	Reads source data, compresses it in “deflate” compression format, and then writes the compressed data to the output stream
Inflater	Performs data decompression
InflaterInputStream	Reads compressed data and then decompresses it in the “deflate” compression format
InflaterOutputStream	Reads compressed data, decompresses it in the “deflate” compression format, and then writes the decompressed data in the output stream
ZipInputStream	Implements an input stream filter to read files in the ZIP file format. It supports compressed and uncompressed entries.
ZipOutputStream	Reads the source data, compresses it in ZIP file format, and writes the data in the output stream

Table 6.2: Classes in `java.util.zip` Package

6.4 Deflater and Inflater Classes

The `Deflater` and `Inflater` class extends from the `Object` class. These classes are used for compressing and decompressing data.

6.4.1 Deflater Class

The `Deflater` class compresses the data present in an input stream. It compresses the data using the ZLIB compression library.

The constructor of the `Deflater` class is used to create instances of the `Deflater` class.

Syntax:

```
public Deflater()
```

The constructor creates an instance with the default compression level.

Methods

Table 6.3 lists various methods available in `Deflater` class along with their description.

Method	Description
<code>deflate(byte[] buffer)</code>	Fills the output buffer with compressed data and returns the actual size of compressed data in integer.
<code>deflate(byte[] buffer, int offset, int len)</code>	Fills the output buffer with compressed data and returns the actual size of compressed data in integer. Here, <code>buffer</code> is the specified buffer used to store compressed data, <code>offset</code> is the start location of the data, and <code>len</code> is the maximum number of bytes of compressed data.
<code>setInput(byte[] buffer)</code>	Sets the input data present in buffer for compression.
<code>setInput(byte[] buffer, int offset, int len)</code>	Sets the input data present in buffer for compression. Here, <code>buffer</code> is the specified buffer used to store input data bytes, <code>offset</code> is the start location of the data, and <code>len</code> is the length of input data.
<code>finish()</code>	Indicates that the compression should end with the current contents of the input buffer.
<code>end()</code>	Closes the compressor and discards the unprocessed input.

Table 6.3: Methods of `Deflater` Class

Code Snippet 2 shows the use of methods in `Deflater` class.

Code Snippet 2:

```
import java.util.zip.Deflater;

public class DeflaterApplication {

    public static void main(String[] args) throws Exception {
        // Encode a String into bytes
        String input = "The Deflater class compresses the data.";
        byte[] inputObj = input.getBytes("UTF-8");
        // Compress the bytes
    }
}
```

```

byte[] output = new byte[100];
Deflater deflater = new Deflater();
deflater.setInput(inputObj);
deflater.finish();
int compressDataLength = deflater.deflate(output);
System.out.println(compressDataLength);
}
}

```

In the code, the input string is initialized and encoded to a byte array. Next, it creates an object of the `Deflater` class, which invokes the `setInput()` method that sets the input data for compression. The `Deflater` object then invokes the `deflate()` method to compress the input string. After compression, the `deflate()` method returns the number of bytes of compressed data which is stored in an integer variable. The value is then displayed. The method helps to obtain the size of the compressed file in bytes.

Figure 6.1 displays the output.

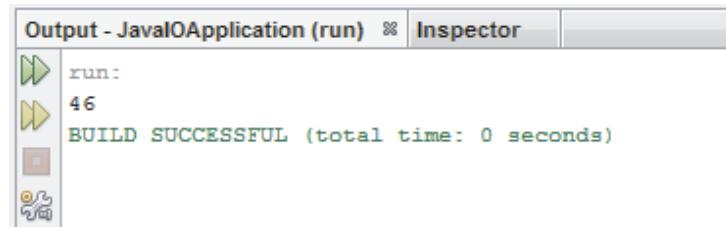


Figure 6.1: DeflaterApplication - Output

6.4.2 Inflater Class

The `Inflater` class decompresses the compressed data. This class supports decompression using the ZLIB compression library.

Syntax:

```
public Inflater()
```

The constructor creates an instance with default compression level.

Table 6.4 lists various methods available in `Inflater` class along with their description.

Method	Description
<code>inflate(byte[] buffer)</code>	Fills the output buffer with decompressed data and returns its actual size in integer.

Method	Description
inflate(byte[] buffer, int offset, int len)	Fills the output buffer with decompressed data and returns its actual size in integer. Here, buffer is the specified buffer used to store the decompressed data, offset is the start location of the data, and len is the maximum number of bytes of decompressed data.
setInput(byte[] buffer)	Sets the input data present in the buffer for decompression.
setInput(byte[] buffer, int offset, int len)	Sets the input data present in the buffer for decompression. Here, buffer is the specified buffer used to store compressed data bytes, offset is the start location of the data, and len is the length of compressed data.
end()	Closes the decompressor.

Table 6.4: Methods of Inflater Class

Code Snippet 3 shows the use of methods in `Inflater` class.

Code Snippet 3:

```
public class DeflaterApplication {

    public static void main(String[] args) throws Exception {
        // Encode a String into bytes
        String input = "The Deflater class compresses the data.";
        byte[] inputObj = input.getBytes("UTF-8");
        // Compress the bytes
        byte[] output = new byte[100];
        Deflater deflaterObj = new Deflater();
        deflaterObj.setInput(inputObj);
        deflaterObj.finish();
        int compressDataLength = deflaterObj.deflate(output);
        System.out.println(compressDataLength);

        // Decompress the bytes
        Inflater inflaterObj = new Inflater();
        inflaterObj.setInput(output, 0, output.length);
    }
}
```

```

byte[] resultObj = new byte[100];
int resultLength = inflaterObj.inflate(resultObj);
inflaterObj.end();
// Decode the bytes into a String
String strOutput = new String(resultObj, 0, resultLength, "UTF-8");
System.out.println("Recovered string is: " + strOutput);
}
}

```

In the code, the input data for decompression is set using the `Inflater` object. Next, the `Inflater` object invokes the `inflate()` method to decompress the compressed data. After decompression, the result is stored in the byte array, `result`. Finally, the `result` is stored in a `String` object.

Figure 6.2 displays the output.

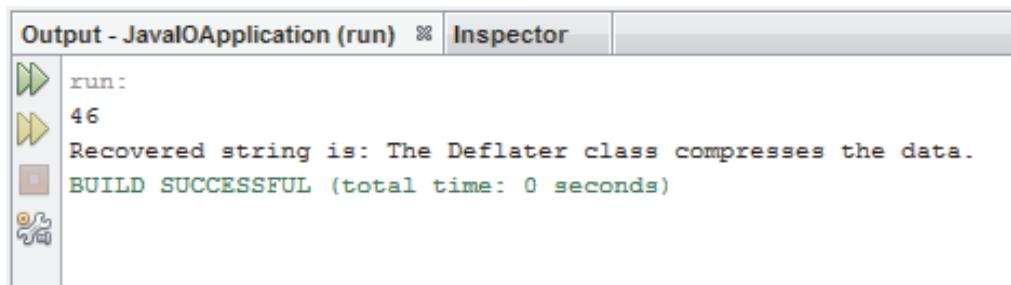


Figure 6.2: Inflater Output

6.5 Compression Classes

The `DeflaterInputStream` and `DeflaterOutputStream` classes are inherited from the `FilterInputStream` and `FilterOutputStream` classes respectively.

6.5.1 *DeflaterInputStream* Class

The `DeflaterInputStream` class reads the source data from an input stream and then compresses it in the 'deflate' compression format. This class provides its own constructors and methods. The syntax for some of the constructors are explained as follows:

Syntax:

- `public DeflaterInputStream(InputStream in)`

The constructor creates an input stream of bytes to read the source data with a default compressor and buffer size.

- `DeflaterInputStream(InputStream in, Deflater defl)`

The constructor creates a new input stream with a default buffer size and the specified compressor.

- `DeflaterInputStream(InputStream in, Deflater defl, int bufLen)`

The constructor creates a new input stream with the buffer size and the specified compressor.

Table 6.5 lists various methods available in `DeflaterInputStream` class along with their description.

Method	Description
<code>read()</code>	Returns one byte of compressed data read from an input stream.
<code>read(byte[] buffer, int offset, int bufferSize) throws IOException</code>	Returns the number of bytes of compressed data read into a byte array starting from the location specified by offset and of bufferSize long.
<code>close()</code>	Closes the input stream after reading the remaining source data.
<code>boolean markSupported()</code>	Returns false. This is because the input stream does not support the <code>mark()</code> and <code>reset()</code> methods.
<code>int available()</code>	Returns 0 after EOF is reached. It returns 1 otherwise.
<code>long skip(long n)</code>	Skips and discards data from the input stream.

Table 6.5: Methods of `DeflaterInputStream` Class

Code Snippet 4 shows the use of methods in `DeflaterInputStream` class.

Code Snippet 4:

```
package java.ioapplication;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.zip.DeflaterInputStream;
```

```
public class DeflaterInputStreamApplication {  
  
    public static byte[] increaseArray(byte[] arrtemp) {  
        byte[] temp = arrtemp;  
        arrtemp = new byte[arrtemp.length + 1];  
        // backs up the data  
        for (int itr = 0; itr < temp.length; itr++) {  
            arrtemp[itr] = temp[itr];  
        }  
        return arrtemp;  
    }  
  
    public static void main(String args[]) throws IOException {  
        FileOutputStream fos = null;  
        try {  
            File file = new File("C:/Java/Hello.txt");  
            FileInputStream fis = new FileInputStream(file);  
            DeflaterInputStream dis = new DeflaterInputStream(fis);  
            // Creating byte array for deflating the data  
            byte input[] = new byte[0];  
            int iindex = -1;  
            // reads data from file  
            int iread = 0;  
            while ((iread = dis.read()) != -1) {  
                input = increaseArray(input);  
                input[++iindex] = (byte) iread;  
            }  
            fos = new FileOutputStream("C:/Java/DeflatedMain.dfl");  
            fos.write(input, 0, input.length);  
            fos.close();  
            System.out.println("File size after compression " +  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            if (fos != null) {  
                try {  
                    fos.close();  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

```
input.length);

} catch (FileNotFoundException ex) {

    Logger.getLogger(DeflaterInputApplication.class.getName()).log(Level.SEVERE, null, ex);
}

} finally {

    try {

        fos.close();
    } catch (IOException ex) {

        Logger.getLogger(DeflaterInputApplication.class.getName()).log(Level.SEVERE, null, ex);
    }
}
}
```

The `increaseArray()` method in the code creates a dynamic array to store the size of the compressed file. The path of the source file is specified as an argument to the instance of the `File` class. The `File` object is then passed as a parameter to the `FileInputStream` object. Next, the `FileInputStream` object is passed as a parameter to the `DeflaterInputStream` instance. A byte array is created for storing the deflated data. The `DeflaterInputStream` object invokes the `read()` method to read data from the source file. The compressed data is stored in the buffer, `input`. The `FileOutputStream` class creates a file named `DeflatedMain.dfl` and then writes the compressed data into it.

Figure 6.3 displays the output.

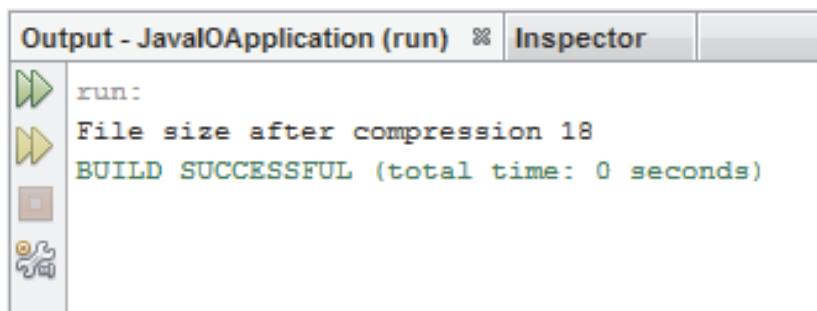


Figure 6.3: DeflaterInputApplication - Output

6.5.2 DeflaterOutputStream Class

The `DeflaterOutputStream` class reads the source data, compresses it in the ‘deflate’ compression format, and then writes the compressed data to a predefined output stream. It also acts as the base for other types of compression filters, such as `GZIPOutputStream`.

Syntax:

```
public DeflaterOutputStream(OutputStream in)
```

The constructor creates an output stream of bytes to write the compressed data to with default compressor and buffer size.

Table 6.6 lists various methods available in `DeflaterOutputStream` class along with their description.

Method	Description
<code>write(int buffer)</code>	Writes one byte of compressed data to the output stream.
<code>write(byte[] buffer, int offset, int bufferSize)</code>	Writes an array of bytes of compressed data to the output stream. Here, <code>buffer</code> is the input data to be written in bytes, <code>offset</code> is the start location of input data, and <code>bufferSize</code> is the size of buffer.
<code>deflate()</code>	Compresses the source data and then writes the next block of compressed data to the output stream.
<code>close()</code>	Closes the output stream after writing the remaining compressed data.
<code>finish()</code>	Completes the process of writing compressed data to the output stream without closing it.

Table 6.6: Methods of `DeflaterOutputStream` Class

Code Snippet 5 shows the use of methods in `DeflaterOutputStream` class.

Code Snippet 5:

```
package java.ioapplication;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.zip.DeflaterOutputStream;

public class DeflaterOutputApplication {
    public static void main(String args[]) {
        try {
            File filein = new File("C:/Java/Hello.txt");
            FileInputStream finRead = new FileInputStream(filein);

```

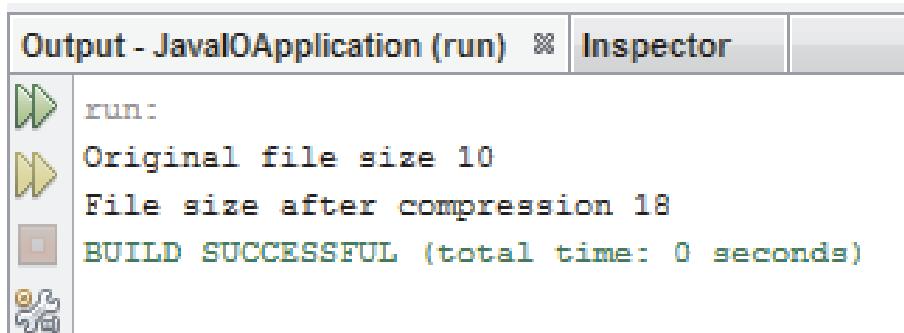
```

File fileout = new File("C:/Java/DeflatedMain.dfl");
FileOutputStream foutWrite = new FileOutputStream(fileout);
DeflaterOutputStream deflWrite = new DeflaterOutputStream(foutWrite);
System.out.println("Original file size " + filein.length());
// Reading and writing the compressed data
int bread = 0;
while ((bread = finRead.read()) != -1) {
    deflWrite.write(bread);
}
// Closing objects
deflWrite.close();
finRead.close();
System.out.println("File size after compression " + fileout.length());
} catch (Exception e) {
    e.printStackTrace();
}
}
}
}

```

In the code, two `File` objects are created, `filein` and `fileout` where, `filein` holds the location of source file and `fileout` holds the location of compressed file. The object, `filein` is passed as a reference to the `FileInputStream` and the object, `fileout` is passed as a reference to the `FileOutputStream`. The `DeflaterOutputStream` object reads the `Hello.txt` file and compresses it. Finally, the `FileInputStream` object named `finRead` invokes the `write()` method to write the compressed data to the output file named `DeflatedMain.dfl`.

Figure 6.4 displays the output.



The screenshot shows the Java Output window with the title "Output - JavaIOApplication (run)". The window contains the following text:

```

run:
Original file size 10
File size after compression 18
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 6.4: DeflaterOutputApplication - Output

6.6 InflaterInputStream and InflaterOutputStream Classes

The `InflaterInputStream` and `InflaterOutputStream` classes are inherited from the `FilterInputStream` and `FilterOutputStream` classes respectively.

6.6.1 InflaterInputStream Class

The `InflaterInputStream` class reads the compressed data and decompresses it in the “deflate” compression format.

Syntax:

```
public InflaterInputStream(InputStream in)
```

The constructor creates an input stream of bytes to read the compressed data with a default decompressor and buffer size.

Table 6.7 lists various methods available in `InflaterInputStream` class along with their description.

Method	Description
<code>read()</code>	Returns one byte of decompressed data read from the input stream
<code>read(byte[] buffer, int offset, int bufferSize)</code>	Returns the number of bytes of decompressed data read into a byte array from the start location specified by <code>offset</code> and of <code>bufferSize</code> long

Table 6.7: Methods of `InflaterInputStream` Class

Code Snippet 6 shows the use of methods in `InflaterInputStream` class.

Code Snippet 6:

```
...
public static void main(String args[]) {
    try {
        File finf = new File("C:\\\\DeflatedMain.dfl");
        FileOutputStream foutWrite = new FileOutputStream(finf);
        File fout = new File("C:\\\\InflatedMain.java");
        FileInputStream finRead = new FileInputStream(fout);
        InflaterInputStream defRead = new InflaterInputStream(finRead);
        System.out.println("File size before Inflation " + fout.length());
        // Inflating the file to original size
        int bread = 0;
```

```

while ((bread = defRead.read()) != -1) {
    fOut.write(bread);
}

fOut.close();
System.out.println("File size after Inflation " + finf.length());
} catch (IOException ioe) {
    ioe.printStackTrace();
}
}

...

```

The code creates two `File` objects, `fOut` and `finf` where, `fOut` holds the location of compressed file and `finf` holds the location of decompressed file. The object, `fOut` is passed as a reference to the `FileInputStream` and the object, `finf` is passed as a reference to the `FileOutputStream`. The `InflaterInputStream` object reads the data in the `FileInputStream` object, decompresses the compressed data, and then invokes the `write()` method to write the decompressed data to the output file named `InflatedMain.java`.

6.6.2 InflaterOutputStream Class

The `InflaterOutputStream` class reads the compressed data, decompresses the data stored in the deflate compression format, and then writes the decompressed data to an output stream. This class also serves as the base class for the decompression class named `GZIPInputStream`.

The `InflaterOutputStream` class provides methods to decompress deflated files.

Syntax:

```
public InflaterOutputStream(OutputStream out)
```

The constructor creates an output stream of bytes to write decompressed data with a default decompressor and buffer size.

Table 6.8 lists various methods available in `InflaterOutputStream` class along with their description.

Methods	Description
<code>write(int buffer)</code>	Write one byte of decompressed data to the output stream
<code>write(byte[] buffer, int offset, int bufferSize)</code>	Writes an array of bytes of decompressed data to the output stream. Here, <code>buffer</code> is the input data in bytes, <code>offset</code> is the start location of input data, and <code>bufferSize</code> is the size of buffer.

Methods	Description
close()	Closes the output stream after writing the remaining uncompressed data
finish()	Completes writing decompressed data to the output stream without closing the underlying output stream

Table 6.8: Methods of InflaterOutputStream Class

Code Snippet 7 shows decompression of data using the methods of `InflaterOutputStream` class.

Code Snippet 7:

```
public class InflaterOutputStreamApplication {

    public static void main(String args[]) {
        try {
            // Writing decompressed data
            File fin = new File("C:/Java/DeflatedMain.dfl");
            FileInputStream finWrite = new FileInputStream(fin);
            File fout = new File("C:/Java/InflatedMain.java");
            FileOutputStream foutWrite = new FileOutputStream(fout);
            InflaterOutputStream infWrite = new InflaterOutputStream(foutWrite);
            System.out.println("Original file size " + fin.length());
            // Reading and writing the decompressed data
            int bread = 0;
            while ((bread = finWrite.read()) != -1) {
                infWrite.write(bread);
            }
            // Inflating the file to original size
            infWrite.close();
            System.out.println("File size after Inflation " + fout.length());
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

The `InflaterOutputStream` object reads the `FileOutputStream` object, compresses the input data, and then invokes the `write()` method to write the decompressed data to the output file named `InflatedMain.java`.

6.7 `java.nio` Package

Java New Input/Output (NIO) API package was introduced in 2002 with J2SE 1.4 to enhance the input/output processing tasks in the Java application development. Although, it was not used to its best capacity, Java SE 7 introduced more New Input/Output APIs (NIO.2).

The primary goal for both NIO and NIO.2 remains same, that is, to enhance the I/O processing tasks in the Java application development. Its use can cut down the time required for certain common I/O operations.

Note - Both NIO and NIO.2 are very **complex** to work with

Both NIO and NIO.2 APIs expose lower-level-system operating-system (OS) entry points. They also provide greater control over I/O. Another aspect of NIO is its attention to application expressivity.

NIO is platform dependent. Its ability to enhance application performance depends on the following:

- OS
- Specific JVM
- Mass storage characteristics
- Data
- Host virtualization context

Following are the central features of the NIO APIs:

- **Charsets and their Associated Decoders and Encoders:** These translate the data between bytes and **Unicode characters**. The charset API is defined in the `java.nio.charset` package. The `charset` package is more flexible than `getBytes()` method, easier to implement, and yields superior performance.
- **Buffers:** These are containers for data. The buffer classes are defined in the `java.nio` package and are used by all NIO APIs.
- **Channels of Various Types:** These represent connections to entities that can perform I/O operations. Channels are abstract files and sockets and supports asynchronous I/O.

- **Selectors and Selection Keys:** These along with selectable channels define a multiplexed, non-blocking I/O facility. Non blocking I/O is event based which means that a selector is defined for an I/O channel and then processing happens. When an event takes place such as the arrival of an input, on the selector, the selector wakes up and executes. This can be performed using a single thread. The channel and selector APIs are defined in the `java.nio.channels` package.

Note - Each subpackage has its own service-provider (SPI) subpackage that helps to extend the platform's default implementations. It also helps to build alternative implementations.

The `java.nio.file` package and `java.nio.file.attribute` package support file I/O. They also help to access the default file system.

6.7.1 File Systems, Paths, and Files

A file system stores and organizes files on media, typically hard drives. Such files can be easily retrieved. Every file has a path through the file system.

→ **File Systems**

Typically, files are stored in a hierarchical structure, where there is a root node. Under the root node, there exist files and directories. Each directory can contain files and subdirectories, which can contain files and subdirectories and so on. There is no limit to the hierarchical structure. File systems can have one or more root directories. File systems have different characteristics for path separators.

In NIO.2, instances of `java.nio.file.Path` objects represent the relative or absolute location of a file or directory.

Note - Before JDK 7, the `java.io.File` class represented files.

→ **Path**

Every file is identified through its path. In other words, the path specifies a unique location in the file system. It starts from the root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as D:\. The Solaris OS supports a single root node. It is represented by the slash character, /.

File system include different characteristics for path separators. The character that separates the directory names is called the delimiter. This is specific to the file system. For example, Microsoft Windows uses the backslash slash (\) and the Solaris OS uses the forward slash (/).

Figure 6.5 illustrates this.

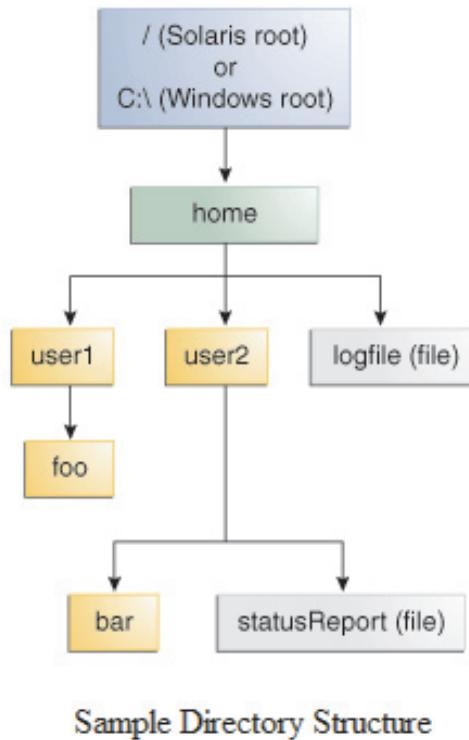


Figure 6.5: Path Separators

Based on figure 6.5, consider the following:

- In the Solaris OS, the path for the `dailySalesReport` file will be represented as `/home/user/dailySalesReport`.
- In Microsoft Windows, the path for the `dailySalesReport` will be represented as `C:\home\user\dailySalesReport`.

A path can be relative or absolute. An absolute path includes the root element and the complete directory list to locate the file. For example, `/home/user/dailySalesReport` is an absolute path. In this all the information required to locate the file is included in the path string.

A relative path does not include the complete directory list. For example, `user/dailySalesReport`. It needs to be used with another path to access a file. Without more information, a program will not be able to locate the file.

→ Files

Before JDK 7, the `java.io.File` class was used for performing all file and directory operations. NIO.2 includes the following new package and classes:

- `java.nio.file.Path`: This uses a system dependent path to locate a file or a directory.

- `Java.nio.file.Files`: This uses a `Path` object to perform operations on files and directories.
- `java.nio.file.FileSystem`: This provides an interface to a file system. This also helps to create a `Path` object and other objects to access a file system.

Note - All methods that access the file system throws `IOException` or a subclass.

The difference between NIO.2 and `java.io.File` is the way that the file system is accessed. In `java.io.File` class the methods to manipulate the path information was in the same class which also contained the methods to read and write files and directories.

With NIO.2 the two process have been separated. In NIO.2, it is the `Path` interface that helps to create and control paths. It is the `Files` class that executes operations on files and directories. The `Files` class operates only on `Path` objects. The `Files` class methods that operate directly on the file system throws an `IOException`.

6.7.2 Symbolic Links

Besides directories or files, certain file systems support symbolic links. These links are also called **symlinks** or a **soft links**.

A symbolic link is a reference to another file and is transparent to applications and users. Operations on symbolic links are automatically redirected to the target of the link. Here, the target is the file or directory that is pointed to. Figure 6.6 displays the symbolic link.

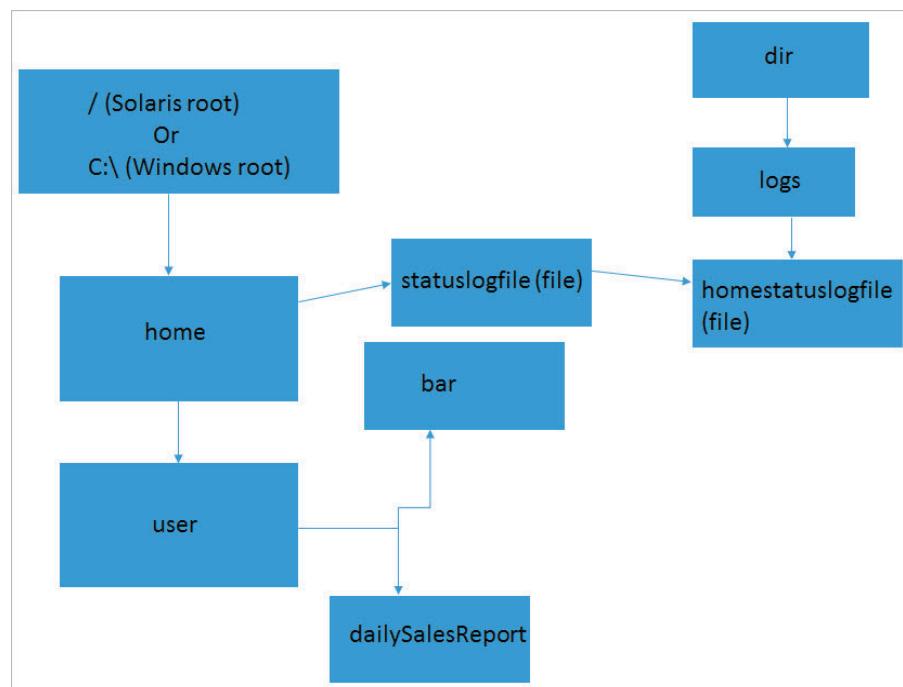


Figure 6.6: Symbolic Link

Note - When a symbolic link is deleted or renamed, the link is deleted or renamed. This does not affect its target.

In the figure 6.6, note the following:

- `statusLogFile` is a symbolic link to `dir/logs/homestatuslogfile`.
- `homestatuslogfile` is the target of the symbolic link.

Reading or writing to a symbolic link is equivalent to reading or writing to any other file or directory.

The act of substituting the actual location in the file system for the symbolic link is called resolving a link. For example, resolving `statusLogFile` will give `dir/logs/homestatulogfile`.

If a symbolic link is carelessly created, there are chances that the target of a link points to the original link. This is called a circular reference. For example, directory `a` points to directory `b`. This in turn points to directory `c`. This directory `c` contains a subdirectory that points to directory `a`.

Such an event can cause issues specially when a program is recursively navigating a directory structure.

Note - Circular reference will not cause the program to loop infinitely.

6.7.3 Path Interface

The `java.nio.file.Path` interface object can help to [locate a file in a file system](#). Typically, the interface represents a system dependent file path. In other words, it provides the entry point for the file and directory manipulation.

A `Path` is hierarchical. It includes a sequence of directory and file name elements. These are separated by a delimiter. Following are the features of a `Path`:

- There could be a root component. This represents a file system hierarchy.
- The name of a file or directory is the name element that is extreme far from the root of the directory hierarchy.
- The other name elements include directory names.

A `Path` can represent the following:

- A root
- A root and a sequence of names
- One or more name elements

A `Path` is empty if it includes only one empty name element.

Note - Accessing a file using an empty path is similar to accessing the default directory of the file system.

The `java.nio.file` package also provides a helper class named `Paths`. This class is static and final and has the `getDefault()` method.

Following are some of the methods of the `Path` interface that can be grouped based on their common functionalities:

- To access the path components or a subsequence of its name elements, the `getFileName()`, `getParent()`, `getRoot()`, and `subpath()` methods can be used.
- To combine paths, the `Path` interface defines `resolve()` and `resolveSibling()` methods can be used.
- To construct a relative path between two paths, the `relativize()` method can be used.
- To compare and test paths, the `startsWith()` and `endsWith()` methods can be used.

Note - A directory located by a path can be `WatchService` and entries in the directory can be watched.

To obtain a `Path` object, obtain an instance of the default file system. Next, invoke the `getPath()` method. Code Snippet 8 illustrates this.

Code Snippet 8:

```
FileSystem fs = FileSystem.getDefault();
Path pathObj = fsObj.getPath("C:/Java/Hello.txt");
```

It is the default provider that creates an object to implement the class. The same object handles all the operations to be performed on a file or a directory in a file system.

`Path` objects once created cannot be changed. In other words, they are immutable.

For `Path` operations, if the default file system is used then the `Paths` utility should be used. The default file system refers to the file system that the JVM is running on. It is the shorter method. For other file systems (not the default one), obtain an instance of a file system and build the `Path` objects for performing `Path` operations.

Note - `Path` objects are similar to `String` objects.

Code Snippet 9 displays the use of the `Path` interface.

Code Snippet 9:

```
package javaioapplication;

import java.nio.file.Path;
import java.nio.file.Paths;

public class PathApplication {
```

```

public static void main(String[] args) {
    Path pathObj = Paths.get("C:/Java/Hello.txt");
    System.out.printf("FileName is: %s%n", pathObj.getFileName());
    System.out.printf("Parent is: %s%n", pathObj.getParent());
    System.out.printf("Name count is: %d%n", pathObj.getNameCount());
    System.out.printf("Root directory is: %s%n", pathObj.getRoot());
    System.out.printf("Is Absolute: %s%n", pathObj.isAbsolute());
}
}

```

Figure 6.7 shows the output.

The screenshot shows the Java Output window with the title "Output - JavaIOApplication (run)". The window displays the following text:

```

FileName is: Hello.txt
Parent is: C:\Java
Name count is: 2
Root directory is: C:\
Is Absolute: true
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 6.7: PathApplication - Output

→ **subpath()**

A portion of a path can be obtained using the `subpath()` method.

Syntax:

```
Path subpath(int startIndex, int endIndex);
```

where,

`startIndex` -represents the beginning value

`endIndex` - represent the `endIndex` value - 1

Code Snippet 10 demonstrates the use of `subpath()` method.

Code Snippet 10:

```

Path pathObj = Paths.get("C:/Java/Hello.txt");

Path pathObj1 = pathObj.subpath(0, 2);

```

In the code, the element name closest to the root has the index value of 0. The element farthest from the root has the index value of index count – 1. Hence, the output of the code will be **Java/Hello.txt**.

→ **resolve()**

The `resolve()` method is used to combine two paths. It accepts a partial path and append the partial path to the original path. Code Snippet 11 uses the `resolve()` method to combine two paths.

Code Snippet 11:

```
Path pathObj = Paths.get("/Java/test");
pathObj.resolve("bar");
```

The output of this will be **/Java/test/bar**.

Passing an absolute path to the `resolve()` method will return the passed-in path. For example, Code Snippet 12 returns **/Java/test**:

Code Snippet 12:

```
Paths.get("bar").resolve("/Java/test");
```

→ **relativize()**

The `relativize()` method helps to construct a path from one location in the file system to another location. Code Snippet 13 illustrates this.

Code Snippet 13:

```
Path pObj = Paths.get("user");
Path p1Obj = Paths.get("sales");
Path pTop = pObj.relativize(p1Obj);
```

The output will be **/sales**.

Note - The new path is relative to the original path. The path originates from the original path and ends at the location defined by the passed-in path.

6.7.4 Working with Links

`Path` interface is link aware and every method either detects what to do or provides an option to configure the behavior when a symbolic link is encountered.

While certain file systems support symbolic link, certain support hard links. Hard links differ from symbolic

links. The following defines a hard link:

- It is not allowed on directories and not allowed to cross partitions or volumes.
- It is hard to find as it behaves like a regular file.
- It should include the target of the link.

The attributes of a hard link is the same as the original file, such as the file permissions.

Note - The `Path` methods work seamlessly with hard links.

Every `Path` method knows how to deal with a symbolic link.

6.7.5 Tracking File System and Directory Changes

Static methods in the `java.nio.file.Files` class perform primary functions for the `Path` objects. The methods in the class can identify and automatically manage symbolic links.

Following are the various `File` operations:

→ Copying a file or directory

To do so, the `copy(Path, Path, CopyOption...)` method can be used. Consider the following when copying:

- If the target file exists, the `REPLACE _ EXISTING` option should be specified. Otherwise, the copy fails.
- When copying a directory, files inside the directory will not be copied.
- When copying a symbolic link, the target of the link is copied. To copy only the link, use the `NOFOLLOW _ LINKS` or `REPLACE _ EXISTING` option.

This method supports the following options:

- `COPY _ ATTRIBUTES`: This copies the file attributes of the file to the target file. File attributes include file system. They are also platform dependent. As an exception, `last-modified-time` is supported across platforms and is copied to the target file.
- `NOFOLLOW _ LINKS`: This is used when symbolic links are not followed. Only the link is copied if the file to be copied is a symbolic link. The target of the link is not copied.
- `REPLACE _ EXISTING`: This copies the file even if the target file exists. If the target is a symbolic link, the link is copied. The target of the link is not copied. If the target is a non-empty directory, the copy fails. The `FileAlreadyExistsException` exception is thrown.

Code Snippet 14 shows how to use the `copy()` method.

Code Snippet 14:

```
import static java.nio.file.StandardCopyOption.*;  
...  
Files.copy(source, target, REPLACE_EXISTING);
```

There are also methods in the `Files` class to copy between a file and a stream. Following are such methods:

- `copy(InputStream in, Path p, CopyOptions... options)`: This copies all bytes from an input stream to a file.

where,

`in` – represents the input stream to read from

`p` – represents the path to the file

`options` – specifies how the copy should be performed

The method throws `IOException`, `FileAlreadyExistsException`, `DirectoryNotEmptyException`, `UnsupportedOperationException`, and `SecurityException`.

- `copy(Path source, OutputStream out)`: This copies all bytes from a file to an output stream.

where,

`source` – represents the path to the file

`out` – represents the output stream to write to

The method throws `IOException` and `SecurityException`.

Code Snippet 15 copies a stream to a path.

Code Snippet 15:

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URI;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
public class CopyStreamApplication {

    public static void main(String[] args) {
        Path pathObj = Paths.get("C:/Java/test.txt");
        URI uriObj = URI.create("http://www.aptech-worldwide.com/");
        try (InputStream inObj = uriObj.toURL().openStream()) {
            Files.copy(inObj, pathObj, StandardCopyOption.REPLACE_EXISTING);
        } catch (final MalformedURLException e) {
            System.out.println("Exception" + e);
        } catch (IOException e) {
            System.out.println("Exception" + e);
        }
    }
}
```

In the code, the content of the Web page (<http://www.aptech-worldwide.com/>) will be copied to a text file created in the specified directory.

Figure 6.8 displays the output.

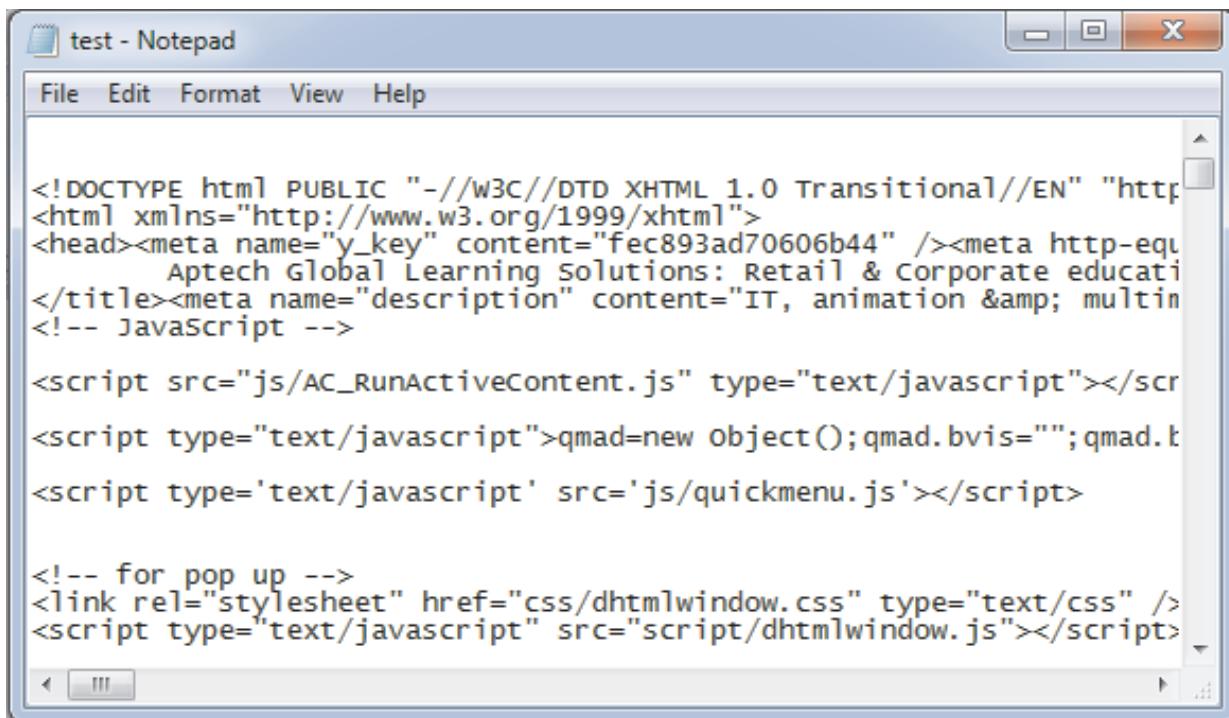


Figure 6.8: CopyStreamApplication - Output

→ Moving a file or directory

To do so, use the `move(Path, Path, CopyOption...)` method. If the target file exists, the `REPLACE_EXISTING` option should be used. Otherwise, the move fails. The method takes a varargs argument.

Empty directories can be moved. If the directory is not empty, it can be moved without moving the contents.

The `move(Path source, Path target, CopyOption... options)` method supports the following `StandardCopyOption` enums:

- REPLACE _ EXISTING: This replaces the target file even when the target file exists for a non-empty directory. If the target is a symbolic link, only the symbolic link is replaced. The target of the link is not replaced.
 - ATOMIC _ MOVE: This moves the directories and files as an atomic file system operation. Use this option to move a file into a directory. Any process then observing the directory accesses a complete file.

Note - An exception is thrown if the file system does not support an atomic move.

The following syntax shows the `move()` method.

Syntax:

```
import static java.nio.file.StandardCopyOption.*;  
...  
Files.move(source, target, REPLACE_EXISTING);
```

where,

source – represents the path to the file to move

target – represents the path to the target file

options- specifies how the move should be performed

The following are the guidelines for moving:

- If the target path is a directory and the directory is empty, the move will succeed if REPLACE _ EXISTING is set
- If the target directory does not exists then the move will succeed
- If the target directory is existing but not empty, then DirectoryNotEmptyException is thrown
- If the source is a file, target directory exists, and REPLACE _ EXISTING is set then the move will rename the file to the intended directory name

Note - On UNIX systems, when a directory is moved within the same partition, typically, it is renamed. In this case, the `move(Path, Path, CopyOption...)` method works even when the directory contains files.

→ Checking a file or directory

To do so, the file system should be accessed using the `Files` methods to determine if a particular `Path` exists. The methods in the `Path` class operate on the `Path` instance.

Following are the `Files` methods for checking the existence of `Path` instance:

- `exists(Path, LinkOption...opt)`: These check if the file exists. By default, it uses symbolic links.
- `notExists(Path, LinkOption...)`: These check if the file does not exist.
`Files.exists(path)` is not equivalent to `Files.notExists(path)`. When testing a file's existence, one of the following is the possible outcome:
 - ◆ The file is verified to not exist.
 - ◆ The file is verified to exist.
 - ◆ The existence of the file cannot be verified. This occurs if both `exists` and `notExists` return false.

- The file's status is unknown. This occurs when the program does not have access to the file.

To check if the program can access a file, the `isReadable(Path)`, `isWritable(Path)`, and `isExecutable(Path)` methods can be used. Code Snippet 16 displays this.

Code Snippet 16:

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
    Files.isReadable(file) & Files.isExecutable(file);
```

Note - The results of these tests are not reliable.

A file system can use two different paths to locate the same file. This occurs when the file system uses symbolic links. The `isSameFile(Path, Path)` method can help compare two paths to check if they locate the same file on the file system. Code Snippet 17 illustrates this.

Code Snippet 17:

```
Path p1 = ...;
Path p2 = ...;
if (Files.isSameFile(p1, p2)) {
    // Logic when the paths locate the same file
```

→ Deleting a file or directory

The `delete(Path)` method can be used to delete a file, directories, or links. The deletion will fail if the directory is not empty. The method throws an exception if the deletion fails. If the file does not exist, a `NoSuchFileException` is thrown.

To determine the reason for the failure, the following exception can be caught as shown in Code Snippet 18.

Code Snippet 18:

```
try {
    Files.delete(path);
} catch (NoSuchElementException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

The `deleteIfExists(Path)` method deletes the file. In addition, if the file does not exist, no exception is thrown. This is useful when there are multiple threads to delete files.

→ Listing a Directory's Content

To do so, use the `DirectoryStream` class that iterates over all the files and directories from any `Path` directory. Consider the following:

- `DirectoryIteratorException` is thrown if there is an I/O error while iterating over the entries in the specified directory.
- `PatternSyntaxException` is thrown when the pattern is invalid.

Code Snippet 19 displays the use of the `DirectoryStream` class.

Code Snippet 19:

```
import java.io.IOException;
import java.nio.file.DirectoryIteratorException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Iterator;
public class ListDirApplication {
```

```

public static void main(String[] args) {
    Path pathObj = Paths.get("D:/resources");
    try (DirectoryStream<Path> dirStreamObj = Files.newDirectoryStream(pathObj, "*.java")) {
        for (Iterator<Path> itrObj = dirStreamObj.iterator(); itrObj.hasNext();) {
            Path fileObj = itrObj.next();
            System.out.println(fileObj.getFileName());
        }
    } catch (IOException | DirectoryIteratorException ex) {
        // IOException can never be thrown by the iteration.
        // In this snippet, itObj can only be thrown by newDirectoryStream.
        System.err.println(ex.getMessage());
    }
}
}

```

Figure 6.9 displays the output.

Output - JavaIApplication (run) ×

- ▶ run:
- ▶ Client.java
- ▶ GenericAcceptReturn.java
- ▶ GenericApplication.java
- ▶ GenericArrayListExample.java
- ▶ HierTest.java
- ▶ MyTest.java
- ▶ MyTestQueue.java
- ▶ NumberList.java
- ▶ StudPair.java
- ▶ TestQueue.java

BUILD SUCCESSFUL (total time: 0 seconds)

Figure 6.9: ListDirApplication - Output

→ Creating and Reading Directories

The `createDirectory(Path dir)` method is used to **create a new directory**.

The `createDirectories()` method can be used to create directories from top to bottom. Code Snippet 20 illustrates this.

Code Snippet 20:

```
Files.createDirectories(Paths.get("C:/Java/test/example"));
```

when executed will create the directories, Java, test, and example, in the given order. The directory test will be created inside Java and example will be created inside the directory test.

To create a file, use the `createFile` method. Consider the following syntax:

Syntax:

```
Files.createFile(Path dir)
```

→ Reading and Writing from Files

To read from files, use the `readAllBytes` or `ReadAllLines` methods that will read the entire content of the file.

Note - The results of these tests are not reliable.

Code Snippet 21 shows the use of the `ReadAllLines` method.

Code Snippet 21:

```
Path pathObj = ....;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

To write bytes or lines to a file, the following methods can be used:

- `write(Path p, byte[] b, OpenOption...)`: The method writes bytes to a file.

where,

p – specifies the path to the file

b – specifies the byte array with the bytes to write

options – specifies the mode of opening the file

Some of the following standard `OpenOptions` enums supported are as follows:

- ◆ WRITE – Opens the file for write access
- ◆ APPEND – Adds the data to the end of the file and is used with WRITE or CREATE options

- ◆ TRUNCATE _ EXISTING – Truncates the file
- ◆ CREATE _ NEW – Creates a new file
- ◆ CREATE – Creates a new file if the file does not exist or opens the file
- ◆ DELETE _ ON _ CLOSE – Deletes the file when the stream closes

The method throws IOException, UnsupportedOperationException, and SecurityException.

- write (Path p, Iterable<extends CharSequence> lines, Charset ch, OpenOption... options): This method writes lines of text to a file.

where,

p – specifies the path to the file

lines – specifies an object to iterate over the char sequences

ch – specifies the charset to be used for encoding

options – specifies the mode of opening the file

The following syntax includes the method to write bytes to a file.

Syntax:

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);
```

The java.nio.file package supports channel I/O.

Note - Channel I/O moves data in buffers. It bypasses certain layers that hinder stream I/O.

→ Reading a File by Using Buffered Stream I/O

The newBufferedReader(Path, Charset) method opens a file for reading. This returns a BufferedReader object that helps to read text from a file efficiently. Code Snippet 22 demonstrates the use of newBufferedReader() method.

Code Snippet 22:

```

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 BufferedIOApplication {

    public static void main(String[] args) {
        Path pathObj = Paths.get("C:/Java/Hello.txt");
        Charset charset = Charset.forName("US-ASCII");
        try (BufferedReader buffReadObj = Files.newBufferedReader(pathObj,
                charset)) {
            String lineObj = null;
            while ((lineObj = buffReadObj.readLine()) != null) {
                System.out.println(lineObj);
            }
        } catch (IOException e) {
            System.err.format("IOException: %s%n", e);
        }
    }
}

```

Figure 6.10 shows the output for Code Snippet 22.

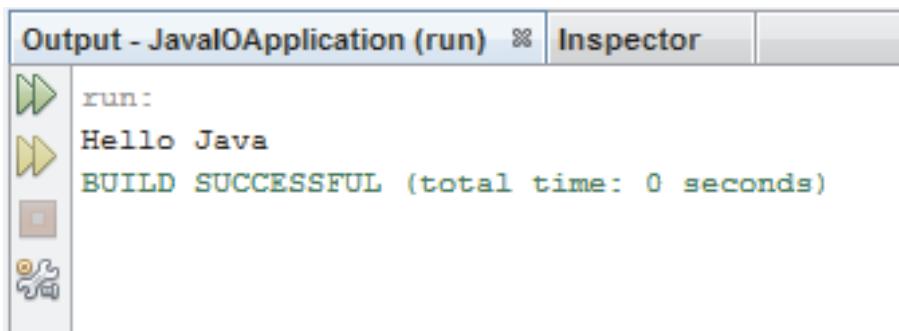


Figure 6.10: BufferedIOApplication - Output

→ Writing a File by Using Buffered Stream I/O

The `newBufferedWriter(Path, Charset, OpenOption...)` method can be used to write to a file using a `BufferedWriter`.

Code Snippet 23 uses the `newBufferedWriter(Path, Charset, OpenOption...)` method to create a file encoded in “US-ASCII”.

Code Snippet 23:

```
public class BufferedIOApplication {  
  
    public static void main(String[] args) {  
        Charset charset = Charset.forName("US-ASCII");  
        String strObj = "Learn Java Programming";  
        Path pathObj1 = Paths.get("C:/Java/JavaPg.txt");  
        try (BufferedWriter writer = Files.newBufferedWriter(pathObj1,  
                charset)) {  
            writer.write(strObj, 0, strObj.length());  
            truyen data cua strObj tu index 0 cho toi het  
        } catch (IOException ex) {  
            System.err.format("IOException: %s%n", ex);  
        }  
    }  
}
```

Figure 6.11 displays the output.

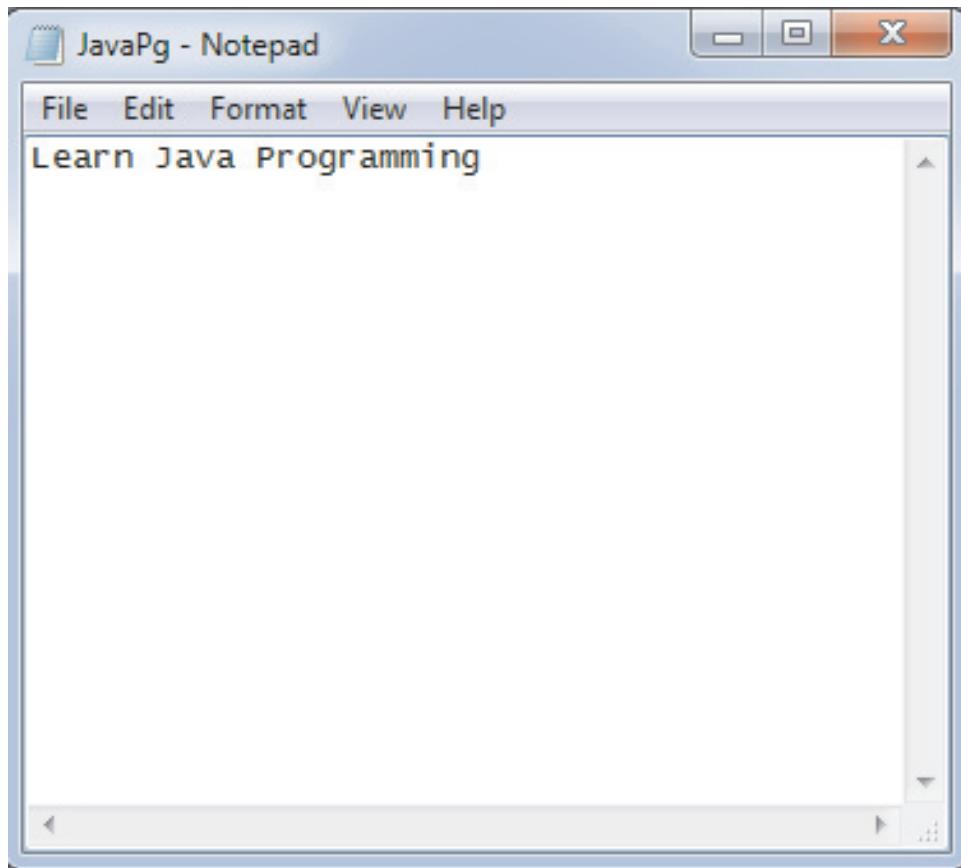


Figure 6.11: Buffered I/O Application - Output

→ Accessing a File Randomly

Files can be nonsequentially or randomly accessed using the `SeekableByteChannel` interface. To randomly access a file, perform the following:

- Open the file.
- Find the particular location.
- Read from the file or write to the file.

The `SeekableByteChannel` interface extends channel I/O and includes various methods to set the position. The data can then be read from the location, or written to it.

The interface includes the following methods:

- `read(ByteBuffer)`: This method reads bytes into the buffer from the channel.
- `write(ByteBuffer)`: This method writes bytes from the buffer to the channel.
- `truncate(long)`: This method truncates the file connected to the channel. The method can be used to truncate any other entity also.
- `position`: This method returns the channel's current position.
- `position(long)`: This method sets the channel's position.

When files are read and written with Channel I/O, the `Path.newByteChannel` methods return an instance of `SeekableByteChannel`. To access advanced features, such as locking a region of the file, the channel can be casted to a `FileChannel`.

Code Snippet 24 displays the use the methods of random access file.

Code Snippet 24:

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardOpenOption.*;
public class RandomAccessApplication {

    public static void main(String[] args) throws FileNotFoundException {
        String strObj = "I love Java programming!\n";
        byte byteDataObj[] = strObj.getBytes();
        ByteBuffer bufObj = ByteBuffer.wrap(byteDataObj);

        ByteBuffer bufCopyObj = ByteBuffer.allocate(20);
        Path pathObj = Paths.get("C:/Java/NewHello.txt");
        //creates a new file if it is not existing
        try (FileChannel fcObj = (FileChannel.open(pathObj, CREATE, READ, WRITE))) {
            //reads the first 20 bytes of the file
            int nreadChar;
            do {
                bufCopyObj.position(0);
                bufCopyObj.limit(20);
                fcObj.read(bufCopyObj);
                byteDataObj = bufCopyObj.array();
                for (int i = 0; i < 20; i++) {
                    System.out.print((char) byteDataObj[i]);
                }
            } while (nreadChar != 0);
        }
    }
}
```

```

nreadChar = fcObj.read(bufCopyObj);
System.out.println(nreadChar);
} while (nreadChar != -1 && bufCopyObj.hasRemaining());

// writes the string at the beginning of the file.
fcObj.position(0);
while (bufObj.hasRemaining()) {
    fcObj.write(bufObj);
}
bufObj.rewind();

// Moves to the end of the file and copies the first 20 bytes to
// the end of the file.
long length = fcObj.size();
fcObj.position(length - 1);
//flips the buffer and sets the limit to the current position
bufCopyObj.flip();
while (bufCopyObj.hasRemaining()) {
    fcObj.write(bufCopyObj);
}
while (bufObj.hasRemaining()) {
    fcObj.write(bufObj);
}
} catch (IOException ex) {
    System.out.println("I/O Exception: " + ex.getMessage());
}
}
}
}

```

In the code, note the following:

- The `SeekableByteChannel` that is returned is cast to a `FileChannel`.

- 20 bytes are read from the beginning of the file, and the string “I love Java programming!” is written at that location.
- The current position in the file is moved to the end, and the 20 bytes from the beginning are appended.
- Next, the string, “I love Java programming!” is appended.
- Finally, the channel on the file is closed.

Figure 6.12 displays the output in a text file.

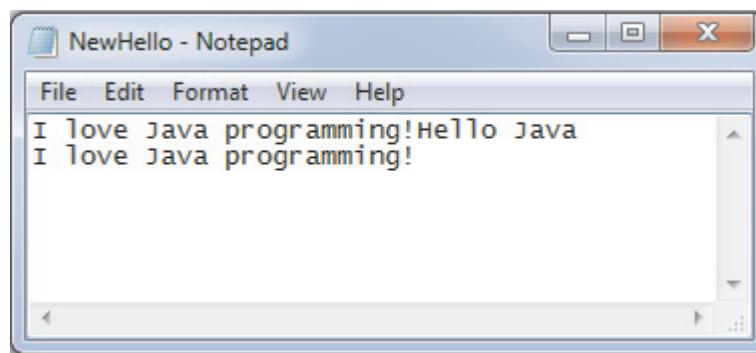


Figure 6.12: RandomAccessApplication - Output

6.7.6 Tracking File System using WatchService and PathMatcher

The `FileSystem` class provides an interface to a file system and is the factory for objects to access files and other objects in the file system. It includes several types of objects. Following are some of the objects:

- `getUserPrincipalLookupService()`: This method returns the `UserPrincipalLookupService` to lookup users or groups by name.
- `newWatchService()`: This method creates a `WatchService` that watches objects for changes.
- `getPath()`: This method converts a system dependent path string and returns a `Path` object to locate and access a file.
- `getPathMatcher()`: This method creates a `PathMatcher` that performs match operations on paths.
- `getFileStores()`: This method returns an iterator over the underlying file-stores.

A file system can have a single hierarchy of files or several distinct file hierarchies. A single hierarchy of

files includes one top-level root directory.

In several distinct file hierarchies, each can have its own top-level root directory. The `getRootDirectories` method can be used to iterate over the root directories in the file system. The `file-stores` store the files in a file system.

A file system can include one or more `file-stores`. These file stores support different features and the file attributes.

Invoke the `close` method to close a file system. Any attempt to access objects in the closed file system throws `ClosedFileSystemException`.

Note - File systems created by the default provider cannot be closed.

A `FileSystem` can provide read-only or read-write access to the file system. Any attempt to write to file stores using an object with a read-only file system throws `ReadOnlyFileSystemException`.

6.7.7 WatchService

A watch service watches registered objects for changes and events. This can be customized to a low level API or a high-level API. Multiple concurrent consumers can use a watch service.

A file manager can use a watch service to check a directory for changes such as when files are created or deleted. This way the file manager can update its list of files when files are created or deleted.

The `register()` method is invoked to register a `Watchable` object with a watch service. This method returns a `WatchKey` to represent the registration. When a change or event for an object occurs, the key is signaled or queued to the watch service. This helps consumers to retrieve the key and process the events when `poll()` or `take()` methods are invoked. After the events are processed, the consumer invokes the key's `reset()` method. This resets the key. The key is then signaled and re-queued with further events.

To cancel a registration with a watch service, the key's `cancel()` method is invoked. However, it remains in the queue till it is retrieved. For example, if a file system is no longer accessible and a key is cancelled in this manner, the return value from the `reset` method indicates if the key is valid.

The key's `reset()` method should be invoked only after its events are processed. This ensures that only one consumer processes the events for a particular object at any time. The `close()` method can be invoked to close the service. If a thread is waiting to retrieve the key, then `ClosedWatchServiceException` is thrown. In other words, it closes the service that is causing any thread waiting to retrieve keys.

If an event indicates that a file in a watched directory is modified, then this does not ensure that the program that has modified the file has completed. Note that there could be other programs that might be updating the file. It should be ensured that there is proper coordination with these programs in terms of access. It is also possible to lock regions of a file against access by other programs. The `FileChannel` class defines various methods that can help do so.

Table 6.9 describes certain methods of `WatchService` interface.

FileChannelClass Methods	Description
<code>WatchKey poll(long timeout, TimeUnit unit)</code>	This method retrieves and removes the next watch key. This can wait till the specified wait time.
<code>WatchKey take()</code>	This method retrieves and removes next watch key. This waits if none are yet present.
<code>Void close()</code>	This closes the watch service.

Table 6.9: WatchService Interface Methods

6.7.8 PathMatcher Interface

To locate a file, one would search directory. One could use a search tool or the `PathMatcher` interface which has a `match` method that determines whether a `Path` object matches a specified string. The `PathMatcher` interface is implemented by objects to match operations on paths. The following is the syntax for the `syntaxAndPattern` string:

Syntax:

Syntax:pattern

where,

syntax: can be glob or regex

Table 6.10 describes certain patterns.

Pattern	Description
<code>.*.{java,class}</code>	This represents file ending with extension either .java or .class.
<code>Client.?</code>	This represents file name starting with Client and a single character extension.
<code>C:*</code>	This represents c:\\test or c:\\Java on Windows platform.
<code>*.java</code>	This represents a file name ending with an extension of .java.
<code>*.*</code>	This represents a file containing a dot.

Table 6.10: Patterns

Table 6.11 describes the rules which are used for interpreting the glob patterns.

Glob Pattern	Description
<code>*</code>	This represents a single or zero character.
<code>**</code>	This represents zero or more characters.
<code>?</code>	This represents exactly one character.
<code>\</code>	This represents escape characters that can be used otherwise as a special character.

Glob Pattern	Description
[]	This represents a single character of a name within the square brackets. For example, [abc] should match a or b or c.
-	is used to specify a range.

Table 6.11: Glob Patterns

*, ?, and \ characters within a bracket match themselves.

Note - Leading period in a file name is considered a regular character while performing a match operation.

{ } characters are a group of subpatterns where the group matches if any subpattern in the group matches. Comma is used for separating the subpattern and groups cannot be nested.

To find a file, the user will search a directory recursively. The `java.nio.PathMatcher` interface has a `match` method to determine whether a `Path` object matches the specified search string. The `FileSystem` factory methods can be used to retrieve the `PathMatcher` instance.

To walk a file tree, the `FileVisitor` interface needs to be implemented. This interface specifies the behavior in traversal process. The key points in the traversal process includes when a file is visited, before accessing a directory, after accessing a directory, or when a failure happens. The methods corresponding to these situations are as follows:

- `preVisitDirectory()` – invoked before visiting a directory
- `postVisitDirectory()` – invoked after all entries in a directory is visited
- `visitFile()` – invoked when a file is visited
- `visitFileFailed()` – invoked when a file cannot be accessed

The `SimpleFileVisitor` class implements the `FileVisitor` interface and overrides all the methods of this class. This class visits all the files and when it encounters an error, it throws an `IOException`. This class can be extended and only the required methods is required to be overridden.

Code Snippet 25 displays the use of `PatternMatcher` and the `SimpleFileVisitor` class to search for a file based on a pattern.

Code Snippet 25:

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
```

```
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import static java.nio.file.FileVisitResult.*;
import java.nio.file.FileVisitResult;
import java.nio.file.attribute.BasicFileAttributes;
class Finder extends SimpleFileVisitor<Path> {

    private Path file;
    private PathMatcher matcher;
    private int num;

    public Finder(Path path, PathMatcher matcher) {
        file = path;
        this.matcher = matcher;
    }

    private void find(Path file) {
        Path name = file.getFileName();
        if (name != null && matcher.matches(name)) {
            num++;
            System.out.println(file);
        }
    }

    void done() {
        System.out.println("Matched: " + num);
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attr) {
        find(file);
    }
}
```

```
// Invoke the pattern matching
// method on each directory.

@Override
public FileVisitResult preVisitDirectory(Path dir,
    BasicFileAttributes attrs) {
    find(dir);
    return CONTINUE;
}

@Override
public FileVisitResult visitFileFailed(Path file,
    IOException exc) {
    System.err.println(exc);
    return CONTINUE;
}

public class PathMatcherApplication {

    public static void main(String[] args) throws IOException {
        Path pathObj;
        pathObj = Paths.get("D:/resources");
        PathMatcher matcherObj = FileSystems.getDefault().getPathMatcher("glob:" +
            "*.java");
        Finder finder = new Finder(pathObj, matcherObj);
        try {
            Files.walkFileTree(pathObj, finder);
        } catch (IOException ex) {
            System.out.println(ex);
        }
    }
}
```

```
}
```

```
}
```

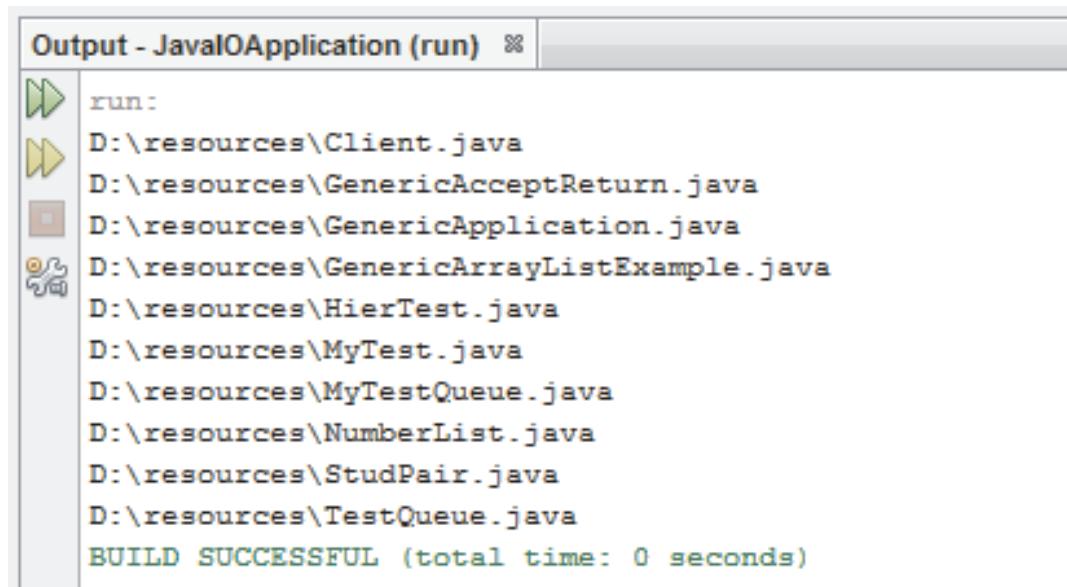
In the code, the `Finder` class is used to walk the tree and search for matches between file and the file specified by the `visitFile()` method. The class extends the `SimpleFileVisitor` class so that it can be passed to a `walkFileTree()` method. This class will call the `match` method on each of the files visited in the tree.

The `FileVisitor` interface has methods that are invoked as each node in the file tree is visited. There is a `SimpleFileVisitor` class that is easier to implement as it implements all the methods of the `FileVisitor` interface. In the `FileVisitor` interface, the `preVisitDirectory()` method is invoked on the class passed to the `walkFileTree()` method. If the `preVisitDirectory()` method returns `FileVisitResult.CONTINUE`, the next node is explored. File tree traversal is complete when all the files have been visited or when the visit method returns `TERMINATE`. This result in traversal being terminated and the error being thrown to the caller of this method.

When the `walkFileTree()` method encounters a file, it attempts to read the `BasicFileAttributes` and the `visitFile()` method is invoked with the File attributes. The `visitFileFailed()` method will be invoked on failing to read the file attributes due to an I/O exception. The `postVisitDirectory()` method is invoked after all the children in the nodes are reached.

In the `PathMatcherApplication`, the `Path` is obtained for the specified directory. Next, the `PathMatcher` instance is created with a regular expression using the `FileSystems` factory method.

Figure 6.13 displays the output.



The screenshot shows the 'Output - JavaIOApplication (run)' window. It displays the command 'run:' followed by a list of files from the 'D:\resources' directory. The files listed are: Client.java, GenericAcceptReturn.java, GenericApplication.java, GenericArrayListExample.java, HierTest.java, MyTest.java, MyTestQueue.java, NumberList.java, StudPair.java, and TestQueue.java. At the bottom of the list, the message 'BUILD SUCCESSFUL (total time: 0 seconds)' is displayed.

```
run:
D:\resources\Client.java
D:\resources\GenericAcceptReturn.java
D:\resources\GenericApplication.java
D:\resources\GenericArrayListExample.java
D:\resources\HierTest.java
D:\resources\MyTest.java
D:\resources\MyTestQueue.java
D:\resources\NumberList.java
D:\resources\StudPair.java
D:\resources\TestQueue.java
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 6.13: PathMatcherApplication - Output

6.8 File Class and File Attributes

As discussed, in a file system, the files and directories include data. It is the metadata that tracks information about each object in the file system. Information can be such as the file creation date, the last file modified date, file owner, and so on.

Note - Metadata means data about other data.

Metadata refers to file attributes of a file system. These file attributes can be achieved by various methods of the Files class. Table 6.12 describes these methods.

Methods	Comment
isRegularFile(Path, LinkOption...)	This method returns true if the specified Path locates a regular file.
isSymbolicLink(Path)	This method returns true if the specified Path locates a file that is symbolic link.
size(Path)	This method returns the size of the specified file in bytes.
isDirectory(Path, LinkOption)	This method returns true if the specified Path locates a file that is directory.
isHidden(Path)	This method returns true if the specified Path locates a hidden file.
getPosixFilePermissions(Path, LinkOption...)	These method return or set a file's POSIX file permissions.
setPosixFilePermissions(Path, Set<PosixFilePermission>)	
getAttribute(Path, String, LinkOption...)	These method return or set the attribute value of a file.
setAttribute(Path, String, Object, LinkOption...)	These method return or set the specified file's last modified time.
getLastModifiedTime(Path, LinkOption...)	
setLastModifiedTime(Path, FileTime)	
getOwner(Path, LinkOption...)	These method return or set the file owner.
setOwner(Path, UserPrincipal)	

Table 6.12: Methods of the Files Class for File Attributes of a File System

To obtain a set of attributes, the Files class includes the following two `readAttributes()` methods to get a file's attributes in one bulk operation:

- **`readAttributes(Path p, Class<A> type, LinkOption... option)`**: This reads a file's attributes as a bulk operation.

where,

`p` – specifies the path to the file

`type` – specifies the type of file attributes that are required to be read

`option` – specifies the way that the symbolic link will be handled

The method returns an object of the type supported. In other word, it returns the file attributes. The method throws `IOException`, `UnsupportedOperationException`, and `SecurityException`.

- **`readAttributes(Path p, String str, LinkOption... option)`**: This reads a set of file's attributes as a bulk operation. Here, the String parameter, `str`, identifies the attributes to be read.

where,

`p` – specifies the path to the file

`str` – specifies the type of file attributes that are required to be read

`option` – specifies the way that the symbolic link will be handled

The method returns a map of attributes. The map's key stores the attribute names and the value stores the attribute values. The method throws `IOException`, `UnsupportedOperationException`, `IllegalArgumentException`, and `SecurityException`.

Note - Repeatedly accessing the file system for a single attribute adversely affects the performance. In addition, if a program requires multiple file attributes simultaneously, methods can be inefficient to retrieve a single attribute.

Related file attributes are grouped together into views because different file systems track different attributes. A view maps to a common functionality, such as file ownership, or to a specific file system implementation, such as DOS.

Table 6.13 describes the supported views.

View	Description
BasicFileAttributeView	This provides a view of basic attributes supported by all file system implementations.
PosixFileAttributeView	This view extends the basic attribute view with attributes, such as file owner and group owner, supported on the POSIX file systems such as UNIX.
DosFileAttributeView	This view extends the basic attribute view with the standard four bits supported on the file systems supporting the DOS attributes.
FileOwnerAttributeView	This view is supported by any file system that supports the concept of a file owner.

View	Description
UserDefinedFileAttributeView	This view supports user-defined metadata. This view maps to extension mechanisms that a file system supports.
AclFileAttributeView	This view supports reading or updating a file's Access Control Lists (ACL). Note: The NFSv4 ACL model is supported.

Table 6.13:Different Attribute Views

A specific file system implementation may support any of the following:

- Only the basic file attribute view
- Several file attribute views
- Other attribute views not included in the API

Note - The `FileAttributeView` can be accessed using the `getFileAttributeView(Path, Class<V>, LinkOption...)` method.

The `readAttributes` methods use generics. These can be used to read the attributes for any of the file attributes views.

Instead of accessing the file system separately to read each individual attribute, it is recommended to use one of the `Files.readAttributes` methods to read the basic attributes of a file. These methods read all the basic attributes in one bulk operation and also prove to be very efficient.

Note - The varargs argument supports the `LinkOption` enum, `NOFOLLOW_LINKS`. Use the option when symbolic links are not to be followed.

The basic attributes of a file system includes the following time stamps:

- `creationTime`
- `lastModifiedTime`
- `lastAccessTime`

A particular implementation when supports these time stamps, returns an `FileTime` object. Otherwise, it returns an implementation-specific value.

6.8.1 DOS File Attributes

File systems other than DOS such as Samba also support DOS file attributes.

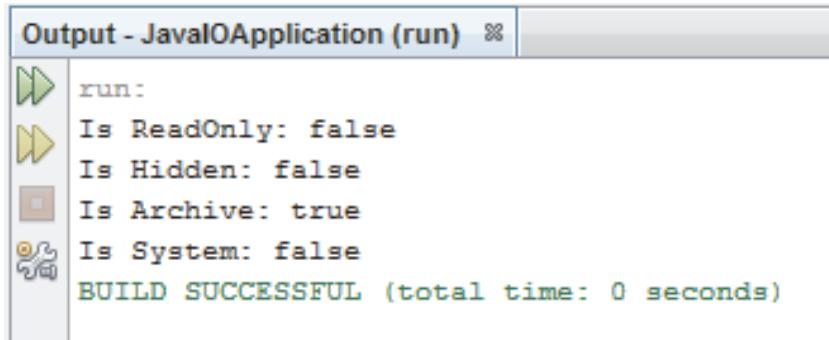
Code Snippet 26 shows the use of the methods of the DosFileAttributes class.

Code Snippet 26:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.DosFileAttributes;
public class DOSFileAttriApplication {

    /**
     *
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        Path pathObj;
        pathObj = Paths.get("C:/Java/Hello.txt");
        try {
            DosFileAttributes attrObj =
                Files.readAttributes(pathObj, DosFileAttributes.class);
            System.out.println("Is ReadOnly: " + attrObj.isReadOnly());
            System.out.println("Is Hidden: " + attrObj.isHidden());
            System.out.println("Is Archive: " + attrObj.isArchive());
            System.out.println("Is System: " + attrObj.isSystem());
        } catch (UnsupportedOperationException ex) {
            System.err.println("DOS file" + " attributes not supported:" + ex);
        }
    }
}
```

Figure 6.14 displays the output.



The screenshot shows the 'Output' window from a Java application named 'JavaIOApplication'. The window title is 'Output - JavaIOApplication (run)'. The output pane displays the following text:
run:
Is ReadOnly: false
Is Hidden: false
Is Archive: true
Is System: false
BUILD SUCCESSFUL (total time: 0 seconds)

Figure 6.14: DOSFileAttributeApplication - Output

A DOS attribute can be set on a file using the `setAttribute(Path, String, Object, LinkOption...)` method. Code Snippet 27 demonstrates the setting of the DOS attribute.

Code Snippet 27:

```
Path file = ...;  
Files.setAttribute(file, "dos:hidden", true);
```

6.9 Check Your Progress

1. Which of the following object provides input and output character streams?

(A)	Console	(C)	InflaterOutputStream
(B)	DeflaterInputStream	(D)	CheckedInputStream

2. The overloaded _____ method reads the compressed data into a byte array starting from the specified location, and a specified buffer size by buffLength.

(A)	read(int[] buffer, int offset, byte[] bufferSize)	(C)	read(byte[] buffer, int offset, int bufferSize)
(B)	read(byte buffer, int offset, int bufferSize)	(D)	read(int buffer, int offset, byte[] bufferSize)

3. The _____ method writes an array of bytes of compressed data starting from a specified location, and specified buffer size.

(A)	write(int[] buffer, int offset, int bufferSize)	(C)	write(byte buffer, int offset, int bufferSize)
(B)	write(int buffer, int offset, int bufferSize)	(D)	write(byte[] buffer, int offset, int bufferSize)

4. Which of the following option represents the character that separates the directory names in a file system?

(A)	Root node	(C)	Delimiter
(B)	Root element	(D)	Absolute

5. Which of the following option define a symbolic link?

(A)	A reference to another file	(C)	Circular reference
(B)	A file system that stores root directories	(D)	Path interface

6.9.1 Answers

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

Summary

- Java SE 6 has introduced the Console class to enhance and simplify command line applications.
- The Console class provides various methods to access character-based console device.
- Java in its java.util.zip package provides classes that can compress and decompress files.
- The Deflater and Inflater classes extend from the Object class.
- The DeflaterInputStream and DeflaterOutputStream classes are inherited from the FilterInputStream and FilterOutputStream classes respectively.
- The InflaterInputStream and InflaterOutputStream classes are inherited from the FilterInputStream and FilterOutputStream classes respectively.
- NIO is platform dependent.
- A file system stores and organizes files on media, typically hard drives.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION



Session 7

Introduction to Threads

Welcome to the Session, **Introduction to Threads**.

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

In this Session, you will learn to:

- ➔ Introduction to Thread
- ➔ Creating Threads
- ➔ Thread States
- ➔ Methods of Thread class
- ➔ Managing Threads
- ➔ Daemon Threads



thead la don vi thi hanh cac task trong mot process

7.1 Introduction

A process is a program that is executing. Each process has its own run-time resources, such as their own data, variables and memory space. These run-time resources constitute an execution environment for the processes inside a program. So every process has a self contained execution environment to run independently. Each process executes several tasks at a time and each task is carried out by separate thread. A thread is nothing but the basic unit to which the operating system allocates processor time. A thread is the entity within a process that can be scheduled for execution. A process is started with a single thread, often called the primary, default or main thread. So a process in turn contains one or more threads.

A thread has the following characteristics:

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

7.1.1 Comparing Processes and Threads

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

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

→ Similarities

- Threads share a central processing unit and only one thread is active (running) at a time.
- Threads within processes execute sequentially.
- A thread can create child threads or sub threads.
- If one thread is blocked, another thread can run.

→ **Differences**

- Unlike processes, threads are **not independent of one another.**
- Unlike processes, all threads can **access every address** in the task.
- Unlike processes, threads are **designed to assist one other.**

7.1.2 Application and Uses of Threads

In Java, a Thread facilitates parallel processing of multiple tasks.

Some of the applications of threads are:

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

7.2 Creating Threads

An easy way of creating a new thread is to derive a class from `java.lang.Thread` class. This class consists of methods and constructors, which helps in realising concurrent programming concept in Java. This is used to create applications which can execute multiple tasks at a given point of time. This approach fails when a class that wants to implement thread is being derived from another class.

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

→ **Step 1: Creating a Subclass**

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

Code Snippet 1 shows the implementation of the `Thread` class as a subclass.

Code Snippet 1:

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

→ **Step 2: Overriding the `run()` Method**

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

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

Code Snippet 2:

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

→ Step 3: Starting the Thread

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

Code Snippet 3 displays the implementation of the `start()` method.

Code Snippet 3:

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

7.2.1 Constructors and Methods of Thread Class

Constructors of the `Thread` class are listed in table 7.1. The `ThreadGroup` class represents a group of threads and is often used in constructors of the `Thread` class.

Directory	Description
<code>Thread()</code>	Default constructor
<code>Thread(Runnable objRun)</code>	Creates a new <code>Thread</code> object, where <code>objRun</code> is the object whose <code>run()</code> method is called

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

Table 7.1: Constructors of Thread Class

The Thread class provides a number of methods to work with threaded programs. Some methods of the Thread class are listed in table 7.2.

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

Table 7.2: Methods of the Thread Class

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

Code Snippet 4:

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

```

}

public static void main(String args[]) {
    NamedThread objNamedThread = new NamedThread();
    objNamedThread.setName("Thread1");
    //Display the status of the thread, whether alive or not
    System.out.println(Thread.currentThread().isAlive());
    System.out.println(objNamedThread.isAlive());
    /*invokes the start method which in turn will call
     * run and begin thread execution
    */
    objNamedThread.start();
    System.out.println(Thread.currentThread().isAlive());
    System.out.println(objNamedThread.isAlive());
}
}

```

In this example, **NamedThread** is declared as a derived class of **Thread**. In the **main()** method of this class, a thread object **objNamedThread** is created by instantiating **NamedThread** and its name is set to **Thread1**. The code then checks to see if the current thread is alive by invoking the **isAlive()** method and displays the return value of the method. This will result in **true** being printed because the main (default) thread has begun execution and is currently alive. The code also checks if **objNamedThread** is alive by invoking the same method on it. However, at this point of time, **objNamedThread** has not yet begun execution so the output will be **false**. Next, the **start()** method is invoked on **objNamedThread** which will cause the thread to invoke the **run()** method which has been overridden. The **run()** method prints the total number of threads running, which are by now, 2. The method then checks the name of the thread running and prints **Marimba** if the currently running thread's name is **Thread1**. The method performs this checking three times. Thus, the final output of the code will be:

```

true
false
true
true
2
Thread1
Marimba
2
Thread1

```

Marimba

2

Thread1

Marimba

2

Thread1

Marimba

Note -

Concurrent Programming

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

7.2.2 Runnable Interface

The Runnable interface is designed to provide a common set of rules for objects that wish to execute a code while a thread is active. Another way of creating a new thread is by **implementing the Runnable interface**. This approach can be used because Java does not allow multiple class inheritance. Therefore, depending upon the need and requirement, either of the approaches can be used to create Java threads.

The step by step procedure for creating and running a new Thread by implementing the Runnable interface has been discussed here.

→ **Step 1: Implementing the Runnable Interface**

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

Code Snippet 5:

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

→ **Step 2: Implementing the `run()` Method**

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

Code Snippet 6:

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

→ **Step 3: Starting the Thread**

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

Code Snippet 7 implements the `start()` method.

Code Snippet 7:

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

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

Code Snippet 8:

```
/*
*Creating threads using Thread
*class and using methods of the class
*/
package test;
/**
```

```

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

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

```

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

Thread-0
Thread-0
Thread-0

7.3 Thread States

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

In this state, it is an empty `Thread` object with no system resources allocated. So the thread is left as a “new thread” state till the `start()` method is invoked on it. When a thread is in this state, you can only start the thread or stop it. Calling any method before starting a thread raises an `IllegalThreadStateException`. Figure 7.1 displays the new state of a thread.

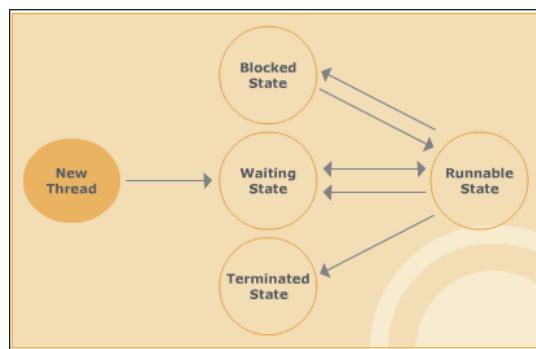


Figure 7.1: New State

Code Snippet 9 displays the creation of a thread object.

Code Snippet 9:

```

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

An instance of `Thread` class is created.

7.3.1 Runnable State

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

Threads are prioritized because in a single processor system all runnable threads cannot be executed at a time. In the runnable state, a thread is eligible to run, but may not be running as it depends on the priority of the thread. The runnable thread, when it becomes eligible for running, executes the instructions in its `run()` method. Figure 7.2 displays the runnable state.

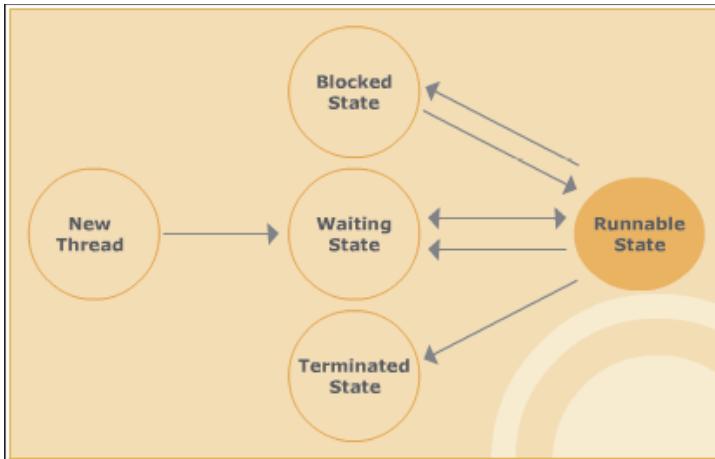


Figure 7.2: Runnable State

Code Snippet 10 demonstrates the runnable state of a thread.

Code Snippet 10:

```

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

```

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

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

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

7.3.2 Blocked State

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

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

A thread in the blocked state waits to operate on the resource or object which at the same time is being processed by another thread. A running thread goes to blocked state when `sleep()`, `wait()`, or `suspend()` method is invoked on it. Figure 7.3 displays the blocked state.

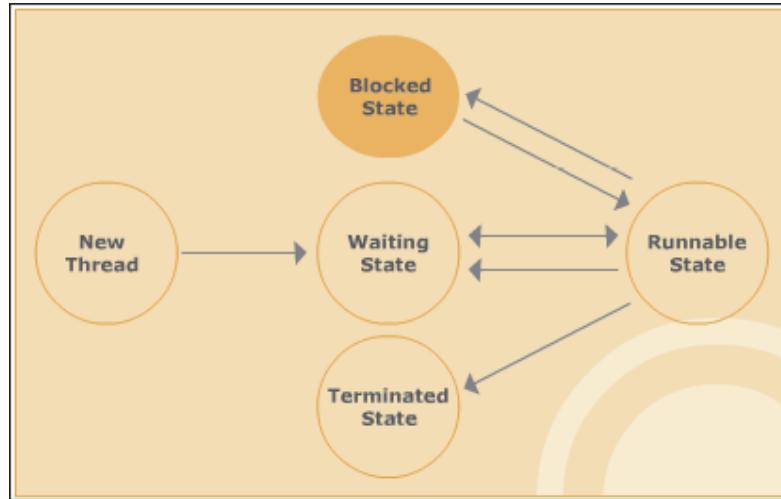


Figure 7.3: Blocked State

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

7.3.3 Waiting State

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

A call to the `wait()` method puts a thread in this state. Invoking the `notify()` or `notifyAll()` method brings the thread from the waiting state to the runnable state. Figure 7.4 displays the waiting state of a thread.

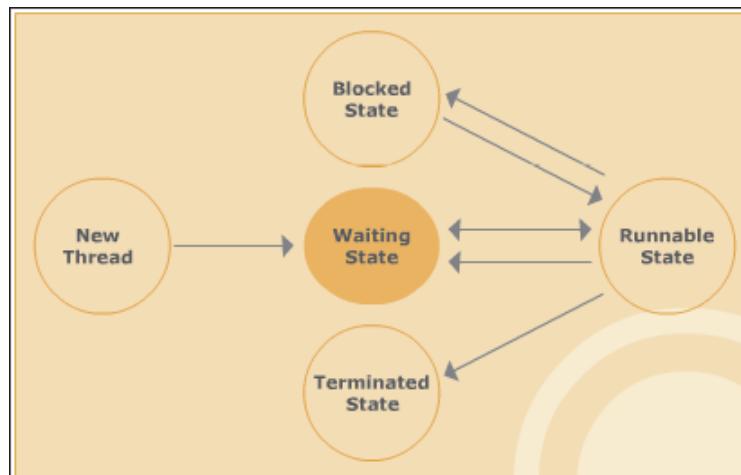


Figure 7.4: Waiting State

7.3.4 Terminated State

A thread, after executing its `run()` method dies and is said to be in a terminated state. This is the way a thread can be stopped naturally. Once a thread is terminated, it cannot be brought back to runnable state. Methods such as `stop()` and `destroy()` can force a thread to be terminated, but in JDK 1.5, these methods are deprecated. Figure 7.5 displays the terminated state of a thread.

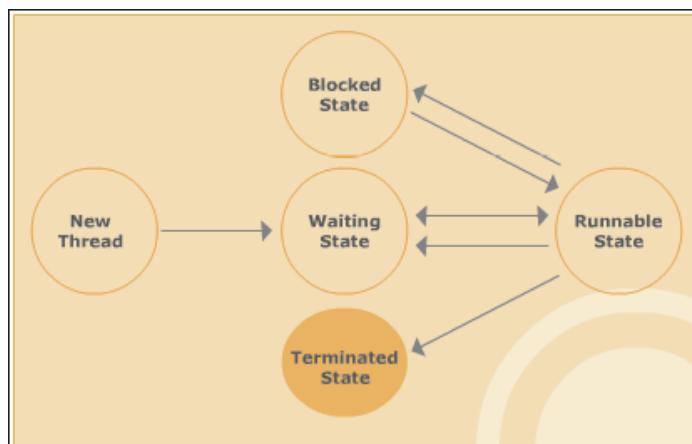


Figure 7.5: Terminated State

Note - If `start()` method is invoked on a terminated state it will throw a run-time exception.

7.4 Methods of Thread Class

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

7.4.1 `getName()` Method

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

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

Code Snippet 11:

```

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

The code snippet demonstrates the use of `getName()` method to obtain the name of the thread object.

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

```
public final void setName(String name)
```

where,

`name` is a String argument passed to the `setName()` method.

7.4.2 `start()` Method

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

At the time of calling the `start()` method:

- A new thread execution starts
- The thread moves from the new thread to runnable state

Figure 7.6 displays the use of the `start()` method in the code.

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

Figure 7.6: `start()` Method

Syntax:

```
void start()
```

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

Code Snippet 12:

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

A thread object is in a runnable state.

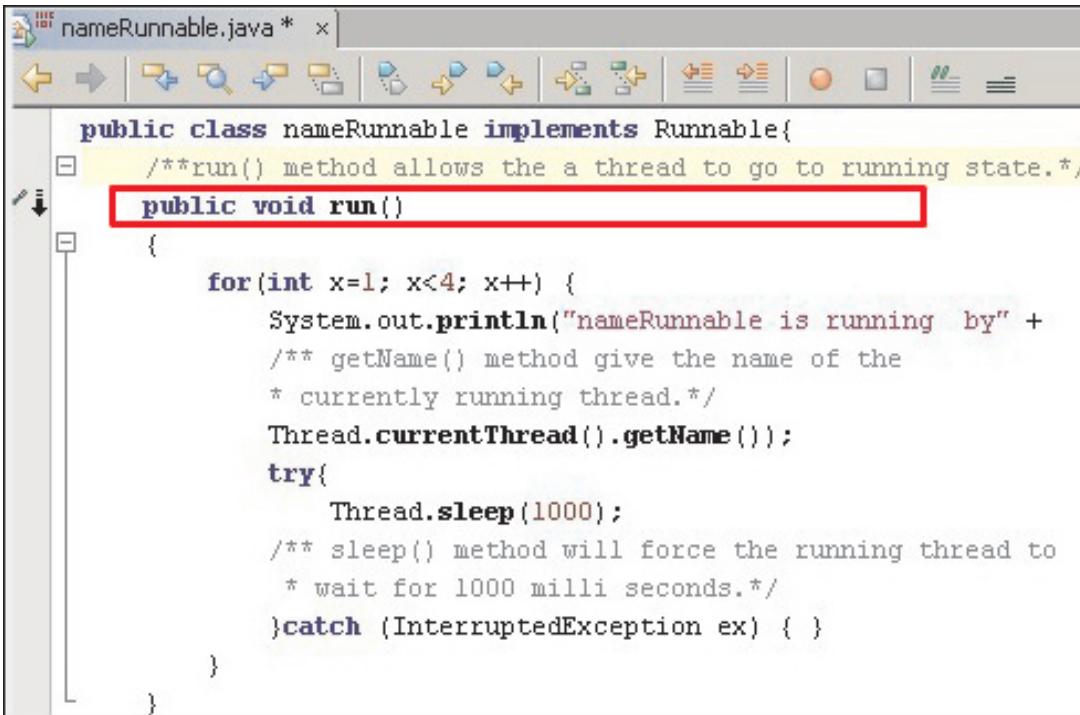
Note - Do not start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

7.4.3 run() Method

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

- It is public
- Accepts no argument
- Does not return any value
- Does not throw any exceptions

The `run()` method contains instructions, which are executed once the `start()` method is invoked. Figure 7.7 displays the use of `run()` method in the code.



The screenshot shows a Java code editor with a file named "nameRunnable.java". The code implements the Runnable interface and defines a run() method. The run() method contains a loop that prints the name of the currently running thread every second. A red box highlights the run() method definition.

```

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

```

Figure 7.7: run() Method

Syntax:

```
public void run()
```

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

Code Snippet 13:

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

The code demonstrates the use of `run()` method.

7.4.4 `sleep()` Method

The `sleep()` method has the following characteristics:

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

Figure 7.8 displays the use of the `sleep()` method in the code.

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

Figure 7.8: `sleep()` Method

Syntax:

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

The following code snippet 14 demonstrates the use of the `sleep()` method.

Code Snippet 14:

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

The code snippet demonstrates the use of `sleep()` method. The thread has been put to sleep for 10000 milliseconds.

7.4.5 interrupt () Method

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

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

Syntax:

```
public void interrupt()
```

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

7.5 Managing Threads

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

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

Threads are self-executing entities inside a program. In a single program, several threads can execute independent of each other. But at times, it may happen that a particular run-time resource has to be shared by many threads, running simultaneously. This forces other running threads to enter the blocked state. So, in such situations some internal control or management of the threads is required so that the threads are executed simultaneously. Figure 7.9 displays multiple threads.

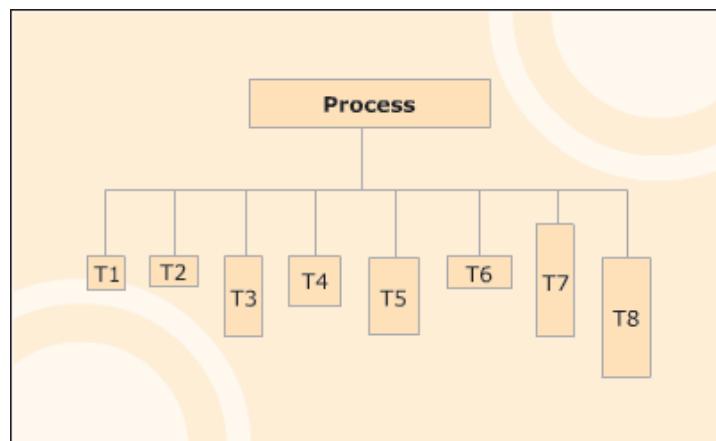


Figure 7.9: Managing Threads

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

7.5.1 Need for Thread Priority

While creating multi-threaded applications, situations may come up where a thread is already running and you need to run another thread of greater importance. This is where thread priorities play an important role. Priorities are used to express the importance of different threads. Priorities play an important part when there is a heavy contention among threads trying to get a chance to execute. This prioritizing process is managed by the scheduler which assigns priority to the respective threads.

Note - Thread priority is similar to your daily life where you have to prioritise your schedules or decisions based on some factors so that every work is carried out according to their priority.

7.5.2 Types of Thread Priority

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

The higher the integers, the higher are the priorities. Higher priority thread gets more CPU time than a

low priority thread. Thread priorities in Java are constants defined in the Thread class. They are:

→ **Thread.MAX_PRIORITY**

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

→ **Thread.NORM_PRIORITY**

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

→ **Thread.MIN_PRIORITY**

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

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

7.5.3 *setPriority () Method*

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

Syntax:

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

Code Snippet 15 demonstrates how to set the priority for a thread.

Code Snippet 15:

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

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

7.5.4 *getPriority () Method*

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

Syntax:

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

7.6 Daemon Threads

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

The characteristics of the daemon threads are:

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

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

- **setDaemon(boolean value)**

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

Syntax:

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

- **isDaemon()**

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

Syntax:

```
boolean isDaemon () "
```

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

7.6.1 Need for Daemon Threads

The task performed by the Daemon threads are:

- Daemon threads are service providers for other threads running in the same process.

- Daemon threads are designed as low-level background threads that perform some tasks such as mouse events for Java program.

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

7.7 Check Your Progress

1. Which of these statements related to the characteristics and uses of processes and threads are true?

(A)	A process has a self-contained execution environment.
(B)	A thread is the smallest unit of executable code in an application that performs a particular task.
(C)	A thread has no definite starting point, execution steps, and terminating point.
(D)	A process contains one or more threads inside it.
(E)	Threads can be used for playing sound and displaying images simultaneously.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

2. Which of the following code snippets creates a thread object and makes it runnable?

(A)	<pre>class MyRunnable implements Runnable { public void run() { . . . //Implementation } } class TestThread { public static void main(String args[]) { MyRunnable mrObj = new MyRunnable(); Thread thObj = new Thread(mrObj); thObj.start(); } }</pre>
-----	---

	<pre>class MyRunnable implements Runnable { public void run() { . . . //Implementation } }</pre>
(B)	<pre>public static void main(String args[]) { MyRunnable mrObj = new MyRunnable(); MyThread thObj=new MyThread(mrObj); } }</pre>
(C)	<pre>class MyThread extends Thread { public void run() { //Implementation } } class TestThread { public static void main(String args[]) { MyThread myThread = new MyThread(); myThread.start(); } }</pre>
(D)	<pre>class MyThread extends Thread { //class declaration public void run() { //Implementation} }</pre>

3. Which of these statements regarding different states of a thread are true?

(A)	A thread in the waiting state is alive but not running.
(B)	A thread is considered terminated when its <code>run()</code> method starts.
(C)	A thread is in a blocked state when it is waiting for the lock of another object.
(D)	The thread comes out of the runnable state when the <code>start()</code> method is invoked on it.
(E)	A new thread is an empty thread object with no system resources allocated.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

4. Which of the following method declarations of the `Thread` class are true?

(A)	public void interrupt()
(B)	static void Sleep(long millis)
(C)	public void run()
(D)	thread.start()
(E)	public String getName()

(A)	A, C, D	(C)	A, C, E
(B)	B, C, D	(D)	A, B, D

5. Which of these statements regarding the priority of threads are true?

(A)	The <code>getPriority()</code> method retrieves the current priority of any thread.
(B)	The <code>setPriority()</code> method changes the current priority of any thread.
(C)	Thread priorities are numbers and range from <code>Thread.MIN_PRIORITY</code> to <code>Thread.MAX_PRIORITY</code> .
(D)	A newly created thread inherits the priority of its parent thread.
(E)	The higher the integers, the lower are the priorities.

(A)	A, C, E	(C)	A, C, D, E
(B)	A, B, C, D	(D)	A, B, D

6. Which of these statements about daemon threads are true?

(A)	Daemon threads are service providers for other threads running in the different process.
(B)	Daemon threads are designed as low-level background threads that perform some task such as mouse events.
(C)	The threads that run system code are user threads.
(D)	The <code>setDaemon()</code> method with the argument as true will convert the user thread to a daemon thread.
(E)	The <code>isDaemon()</code> method returns true if the thread is a user thread.

(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

7.7.1 Answers

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

```
g packa
String pac
String pr
ids string
y catch
```

Summary

- A process consists of many semi-processes known as threads. A thread is the smallest executable code in a Java program, which contributes in making it possible to run multiple tasks at a time.
- There are two ways of creating a Thread object. A thread object can be created by extending the Thread class that defines the run() method. It can also be created by declaring a class that implements the Runnable interface and passing the object to the Thread constructor.
- A newly created thread can be in following states: new, runnable, waiting, blocked, and terminated. A thread is said to be new Thread after it has been instantiated and the start() method has not been invoked. When the start() method is invoked on the thread object, the thread is said to be in runnable state. The thread is in waiting state when the wait() method has been invoked on it.
- A thread is said to be in blocked state while waiting for the lock of an object. When the run() method has been completely executed a thread terminates.
- The getName() method returns the name of the thread while start() allows a thread to be in runnable state. The run() method allows a thread to start executing whereas sleep() forces a running thread to wait for another thread. Finally, the interrupt() method redirects a thread to perform another task keeping aside what it was doing before.
- The setPriority() and getPriority() methods are used to assign and retrieve the priority of any thread respectively. While working with multiple threads, it is inevitable to prioritise the threads so that they will be executed in a sequence without interfering each other.
- The daemon thread runs independently of the default user thread in a program, providing background service like the mouse event, garbage collection and so on. The isDaemon() method helps to find whether a thread is a daemon or not, whereas setDaemon() method changes a user thread to a daemon thread.

Amplification **Metaphrase**
Decoding **Abbreviations**
Terms **Acronyms**
GLOSSARY **Explanation**
look-up guide Terminology
Wordbook Wordlist

Session 8

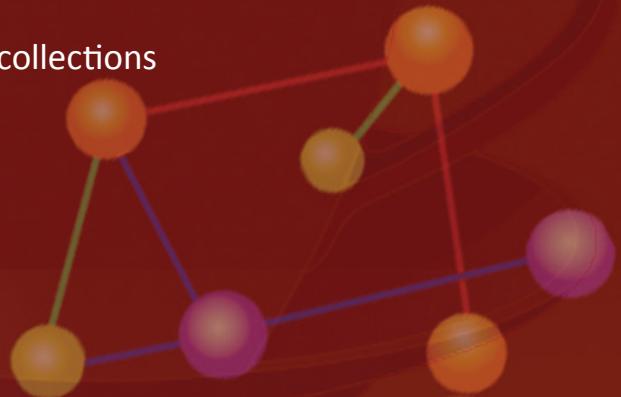
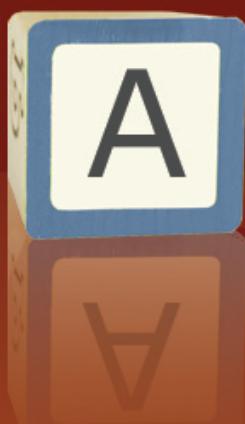
Multithreading and Concurrency

Welcome to the Session, **Multithreading and Concurrency**.

This session describes multithreading, which allows two parts of the program to run simultaneously. The session also explains the concept of synchronization. The mechanism for inter-process communication will also be discussed. The session describes how to deal with deadlocks which cause two processes to be waiting for a resource and also describes java.util.concurrent collections.

In this Session, you will learn to:

- Define multithreading
- Differentiate between multithreading and multitasking
- Explain the use of isAlive() and join() method
- Explain race conditions and ways to overcome them
- Describe intrinsic lock and synchronization
- Describe atomic access
- Describe the use of wait() and notify() methods
- Define deadlock and describe the ways to overcome deadlock
- Explain java.util.concurrent collections



8.1 Introduction

A thread performs a certain task and is the smallest unit of executable code in a program. Multitasking can be process-based or thread-based. In process-based multitasking two or more programs run concurrently. In thread-based multitasking a program performs two or more tasks at the same time. Thus, multithreading can be defined as the concurrent running of the two or more parts of the same program. There are basically two types of multitasking in use among operating systems. These are as follows:

→ **Preemptive**

In this case, the operating system controls multitasking by assigning the CPU time to each running program. This approach is used in Windows 95 and 98.

→ **Cooperative**

In this approach, related applications voluntarily surrender their time to one another. This was used in Windows 3.x.

Multithreading is a technique similar to multitasking and involves the creation of one or more threads within a program to enable number of tasks to run concurrently or in parallel.

Multithreading also supports the following features:

- Managing many tasks concurrently.
- Distinguishing between tasks of varying priority.
- Allowing the user interface to remain responsive, while allocating time to background tasks.

Consider a Web site that has both images and text as its content. When the Web site is loaded on a user's computer, both the content must be displayed at the same time. Displaying the image or text is a separate task but must happen simultaneously. This is where multithreading comes into picture.

8.1.1 Multithreading vs. Multitasking

Multithreading is a specialized form of multitasking. The differences between multithreading and multitasking have been provided in table 8.1.

Multithreading	Multitasking
In a multithreaded program, two or more threads can run concurrently.	In a multitasking environment, two or more processes run concurrently.
Multithreading requires less overhead. In case of multithreading, threads are lightweight processes. Threads can share same address space and inter-thread communication is less expensive than inter-process communication.	Multitasking requires more overhead. Processes are heavyweight tasks that require their own address space. Inter-process communication is very expensive and the context switching from one process to another is costly.

Table 8.1: Differences between Multithreading and Multitasking

8.1.2 Need for Multithreading

Multithreading is needed for the following reasons:

- Multithreading increases performance of single-processor systems, as it reduces the CPU idle time.
- Multithreading encourages faster execution of a program when compared to an application with multiple processes, as threads share the same data whereas processes have their own sets of data.
- Multithreading introduces the concept of parallel processing of multiple threads in an application which services a huge number of users.

The Java programming language provides ample support for multithreading by means of the Thread class and Runnable interface.

Code Snippet 1 creates multiple threads, displays the count of the threads, and displays the name of each running child thread within the run() method.

Code Snippet 1:

```
/**  
 * Creating multiple threads using a class derived from Thread class  
 */  
  
package test;  
/**  
 * MultipleThreads is created as a subclass of the class Thread  
 */  
public class MultipleThreads extends Thread {  
  
    // Variable to store the name of the thread  
    String name;
```

```

/**
 * This method of Thread class is overridden to specify the action
 * that will be done when the thread begins execution.
 */

public void run() {
    while(true) {
        name = Thread.currentThread().getName();
        System.out.println(name);
        try {
            Thread.sleep(500);
        } catch(InterruptedException e) {
            break;
        }
    } // End of while loop
}

/**
 * This is the entry point for the MultipleThreads class.
 */

public static void main(String args[]) {
    MultipleThreads t1 = new MultipleThreads();
    MultipleThreads t2 = new MultipleThreads();
    t1.setName("Thread2");
    t2.setName("Thread3");
    t1.start();
    t2.start();
    System.out.println("Number of threads running: " + Thread.
activeCount());
}
}

```

In the code, the `main()` method creates two child threads by instantiating the `MultipleThreads` class which has been derived from the `Thread` class. The names of the child threads are set to **Thread2** and **Thread3** respectively. When the `start()` method is invoked on the child thread objects, the control is transferred to the `run()` method which will begin thread execution.

Just as the child threads begin to execute, the number of active threads is printed in the `main()` method.

Figure 8.1 shows the output of the code.

```
run:
Thread2
Number of threads running: 3
Thread3
Thread2
Thread3
Thread2
Thread3
Thread3
Thread2
Thread3
Thread2
Thread2
Thread2
```

Figure 8.1: Output of Code Snippet 1

The code executes until the user presses `Ctrl + C` to stop execution of the program.

Thread execution stops once it finishes the execution of `run()` method. Once stopped, thread execution cannot restart by using the `start()` method. Also, the `start()` method cannot be invoked on an already running thread. This will also throw an exception of type `IllegalThreadStateException`.

Threads eat up a lot memory (RAM) therefore it is always advisable to set the references to null when a thread has finished executing. If a `Thread` object is created but fails to call the `start()` method, it will not be eligible for garbage collection even if the underlying application has removed all references to the thread.

8.2 `isAlive()` Method

The thread that is used to **start the application should be the last thread to terminate**. This indicates that the application has terminated. This can be ensured by stopping the execution of the main thread for a longer duration within the main thread itself. Also it should be ensured that all the child threads terminate before the main thread. However, how does one ensure that the main thread is aware of the status of the other threads? There are ways to find out if a thread has terminated. First, by using the `isAlive()` method.

A thread is considered to be alive when it is running. The `Thread` class includes a method named `isAlive()`. This method is used to find out whether a specific thread is running or not. If the thread is alive, then the boolean value `true` is returned. If the `isAlive()` method returns `false`, it is understood that the thread is either in new state or in terminated state.

Syntax:

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

Code Snippet 2 displays the use of `isAlive()` method.

Code Snippet 2:

```
...
public static void main(String [] args)
{
    ThreadDemo Obj = new ThreadDemo ();
    Thread t = new Thread(Obj);
    System.out.println("The thread is alive :" + t.isAlive());
}
...
```

The code demonstrates the use of `isAlive()` method. The method returns a boolean value of true or false depending whether the thread is running or terminated. The code will return false since the thread is in new state and is not running.

8.2.1 *join () Method*

join tang do uu tien cho cac thread su dung no

The `join()` method causes the current thread to wait until the thread on which it is called terminates. The `join()` method performs the following operations:

- ➔ This method allows specifying the maximum amount of time that the program should wait for the particular thread to terminate.
- ➔ It throws `InterruptedException` if another thread interrupts it.
- ➔ The calling thread waits until the specified thread terminates.

Syntax:

```
public final void join()
```

Code Snippet 3 displays the use of `join()` method.

Code Snippet 3:

```
try {
    System.out.println("I am in the main and waiting for the thread to finish");
    objTh.join(); // objTh is a Thread object
}
```

```

catch(InterruptedException e)
{
    System.out.println("Main thread is interrupted");
}
...

```

The code illustrates the use of the `join()` method. In this snippet, the current thread waits until the thread, `objTh` terminates.

The `join()` method of the `Thread` class has two other overloaded versions:

→ **`void join(long timeout)`**

In this type of `join()` method, an argument of type `long` is passed. The amount of timeout is in milliseconds. This forces the thread to wait for the completion of the specified thread until the given number of milliseconds elapses.

→ **`void join(long timeout, int nanoseconds)`**

In this type of `join()` method arguments of type `long` and `integer` are passed. The amount of timeout is given in milliseconds in addition to a specified amount of nanoseconds. This forces the thread to wait for the completion of the specified thread until the given timeout elapses.

Code Snippet 4 displays the use of the different methods of the `Thread` class.

Code Snippet 4:

```

/*
 * Using the isAlive and join methods
 */

package test;

/** ThreadDemo inherits from Runnable interface */
class ThreadDemo implements Runnable {

    String name;
    Thread objTh;

    /* Constructor of the class */
    ThreadDemo(String str) {
        name = str;
        objTh = new Thread(this, name);
        System.out.println("New Threads are starting : " + objTh);
    }
}

```

```
objTh.start();  
}  
  
public void run() {  
    try {  
        for (int count = 0; count < 2; count++) {  
            System.out.println(name + ":" + count);  
            objTh.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println(name + " interrupted");  
    }  
    System.out.println(name + " exiting");  
}  
  
public static void main(String [] args) {  
    ThreadDemo objNew1 = new ThreadDemo ("one");  
    ThreadDemo objNew2 = new ThreadDemo ("two");  
    ThreadDemo objNew3 = new ThreadDemo ("three");  
    System.out.println("First thread is alive :" + \  
    objNew1.objTh.isAlive());  
    System.out.println("Second thread is alive :" + objNew2.objTh.isAlive());  
    System.out.println("Third thread is alive :" + objNew3.objTh.isAlive());  
    try {  
        System.out.println("In the main method, waiting for the threads to finish");  
        objNew1.objTh.join();  
        objNew2.objTh.join();  
        objNew3.objTh.join();  
    } catch (InterruptedException e) {  
        System.out.println("Main thread is interrupted");  
        System.out.println("First thread is alive :" + objNew1.objTh.isAlive());  
    }
```

```

System.out.println("Second thread is alive :" + objNew2.objTh.isAlive());
System.out.println("Third thread is alive :" + objNew3.objTh.isAlive());
System.out.println("Main thread is over and exiting");
}
}

```

In the code, three thread objects are created in the `main()` method. The `isAlive()` method is invoked by the three thread objects to test whether they are alive or dead. Then the `join()` method is invoked by each of the thread objects. The `join()` method ensures that the main thread is the last one to terminate. Finally, the `isAlive()` method is invoked again to check whether the threads are still alive or dead. These statements are enclosed inside a try-catch block.

Figure 8.2 shows the output of the code.

```

run:
New Threads are starting : Thread[one,5,main]
New Threads are starting : Thread[two,5,main]
New Threads are starting : Thread[three,5,main]
First thread is alive :true
Second thread is alive :true
Third thread is alive :true
In the main method, waiting for the threads to finish
two : 0
one : 0
three : 0
one : 1
three : 1
two : 1
three exiting
two exiting
one exiting
BUILD SUCCESSFUL (total time: 3 seconds)

```

Figure 8.2: Output of Code Snippet 4

8.3 Thread Synchronization

In multithreaded programs, several threads may simultaneously try to update the same resource, such as a file. This leaves the resource in an undefined or inconsistent state. This is called race condition.

8.3.1 Race Conditions

In general, race conditions in a program occur when:

- Two or more threads share the same data between them.
- Two or more threads try to read and write the shared data simultaneously.

The race conditions can be avoided by using synchronized blocks. This is a block of code qualified by the `synchronized` keyword.

8.3.2 Synchronized Blocks and Methods

Consider a situation where people are standing in a queue outside a telephone booth. They wish to make a call and are waiting for their turn. This is similar to a synchronized way of accessing data where each thread wanting to access data waits its turn. However, going back to the analogy described earlier, if there was no queue and people were permitted to go in randomly, two or more persons would try to enter the booth at the same time, resulting in confusion and chaos. This is similar to a race condition that can take place with threads. When two threads attempt to access and manipulate the same object and leave the object in an undefined state, a race condition occurs. Java provides the `synchronized` keyword to help avoid such situations. The core concept in synchronization with Java threads is something called a monitor. A monitor is a piece of code that is guarded by a mutual-exclusion program called a mutex. A real-life analogy for a monitor can be the telephone booth described earlier, but this time with a lock. Only one person can be inside the telephone booth at a time and while the person is inside, the booth will be locked, thus preventing the others from entering. The telephone inside the booth here is the real-life equivalent of an object, the booth is the monitor and the lock is the mutex. A Java object has only one monitor and mutex associated with it.

Thus, synchronized blocks are used to prevent the race conditions in Java applications. The synchronized block contains code qualified by the `synchronized` keyword. A lock is assigned to the object qualified by `synchronized` keyword. When a thread encounters the `synchronized` keyword, it locks all the doors on that object, preventing other threads from accessing it. A lock allows only one thread at a time to access the code. When a thread starts to execute a synchronized block, it grabs the lock on it. Any other thread will not be able to execute the code until the first thread has finished and released the lock. The lock is based on the object and not on the method.

The syntax to create the synchronized block and is as follows:

Syntax:

```
// Synchronized block
synchronized(object)
{
    // statements to be synchronized
}
```

where,

Object is the reference of the object being synchronized.

Code Snippet 5 demonstrates the synchronized block.

Code Snippet 5:

```
...
public class Account {
    double balance = 200.0;
    public void deposit(double amount) {
        balance = balance + amount;
    }
    public void displayBalance() {
        System.out.println("Balance is:" + balance);
    }
}
public class Transaction implements Runnable {
    double amount;
    Account account;
    Thread t;
    public Transaction(Account acc, double amt) {
        account = acc;
        amount = amt;
        t = new Thread(this);
        t.start();
    }
    // Synchronized block calls deposit method
    public void run() {
        synchronized (account) { // Synchronized block
            account.deposit(amount);
            account.displayBalance();
        }
    }
}
```

```
public class DepositAmount {
    public static void main(String[] args) {
        Account accObj = new Account();
        Transaction t1 = new Transaction(accObj, 500.00);
        Transaction t2 = new Transaction(accObj, 200.00);
    }
}
```

The code creates an **Account** class with methods namely, **deposit()** and **displayBalance()** which will add the amount to the existing balance and display it.

The **Transaction** class creates a Thread object and calls the **run()** method on it. The **run()** method contains a synchronized block that will mark the block as synchronized. The synchronized block takes the account object which will act as a monitor object. This allows only one thread to be executed inside the synchronized block on the same monitor object.

Figure 8.3 displays the output of the code.

```
run:
Balance is:700.0
Balance is:900.0
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 8.3: Output - Synchronized Block

8.3.3 Synchronized Methods

The synchronized method obtains a lock on the class object. This means that at a single point of time only one thread obtains a lock on the method, while all other threads need to wait to invoke the synchronized method.

Consider a scenario where two threads need to perform the read and write operation on a single file stored on the system. If both threads attempt to manipulate the file data for read or write operations simultaneously, it may leave the file in inconsistent state. To prevent this, a synchronized method can be defined. Each thread will acquire a lock on the method to perform the respective operation. Thus, both the thread cannot invoke the method at the same time, as it will be locked by the other thread.

The syntax to declare the synchronized method is as follows:

Syntax:

```
synchronized method(....)
{
    // body of method
}
```

Note - Constructors cannot be synchronized.

In this snippet, the `deposit()` method of `Account` class has been synchronized by using a `synchronized` keyword. This method can be accessed by a single thread at a time from the several threads in a program. The `synchronized` method allows threads to access it sequentially.

Code Snippet 6 shows how to use a synchronized method.

Code Snippet 6:

```
/**  
 * Demonstrating synchronized methods.  
 */  
  
package test;  
  
class One {  
  
    // This method is synchronized to use the thread safely  
    synchronized void display(int num) {  
  
        num++;  
  
        System.out.print(num);  
  
        try {  
            Thread.sleep(1000);  
        } catch(InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
  
        System.out.println(" done");  
    }  
}  
  
class Two extends Thread {  
  
    int number;  
    One objOne;  
  
    public Two(One one_num, int num) {  
        objOne = one_num;  
        number = num;  
    }  
}
```

```

public void run() {
    // Invoke the synchronized method
    objOne.display(number);
}

class SynchMethod {
    public static void main(String args[]) {
        One objOne = new One();
        int digit = 10;
        // Create three thread objects
        Two objSynch1 = new Two(objOne);
        Two objSynch2 = new Two(objOne);
        Two objSynch3 = new Two(objOne);
        objSynch1.start();
        objSynch2.start();
        objSynch3.start();
    }
}

```

Here, the class **One** has a method **display()** that takes an **int** parameter. This number is displayed with a suffix “done”. The **Thread.sleep(1000)** method pauses the current thread after the method **display()** is called.

The constructor of the class **Two** takes a reference to an object **t** of the class **One** and an integer variable. Here, a new thread is also created. This thread calls the method **run()** of the object **t**. The main class **SynchDemo** instantiates the class **One** as a object **objOne** and creates three objects of the class **Two**. The same object **objOne** is passed to each **Two** object. The method **join()** makes the caller thread wait till the calling thread terminates.

The output of the Code Snippet 6 would be:

```

10 done
11 done
12 done

```

It is not always possible to achieve synchronization by creating synchronized methods within classes. The reason for this is as follows.

Consider a case where the programmer wants to synchronize access to objects of a class, which does not use synchronized methods. Also assume that the source code is unavailable because either a third party created it or the class was imported from the built-in library.

In such a case, the keyword synchronized cannot be added to the appropriate methods within the class. Therefore, the problem here would be how to make the access to an object of this class synchronized. This could be achieved by putting all calls to the methods defined by this class inside a synchronized block.

8.3.4 Intrinsic Locks and Synchronization

Synchronization is built around the concept of intrinsic lock or monitor lock. This is an internal entity. Intrinsic locks help impose exclusive access to an object's state. They also help to establish important visible relationships.

Every object is connected to an intrinsic lock. Typically, a thread acquires the object's intrinsic lock before accessing its fields, and then on completion of the required task releases the intrinsic lock. In this span of acquiring and releasing the intrinsic lock, the thread owns the intrinsic lock. No other thread can acquire the same lock. The other thread will not be allowed to acquire the lock when it attempts to obtain it.

On releasing an intrinsic lock, the thread establishes a happens-before relationship. This relationship is established between that action and any subsequent acquisition of the same lock.

When a thread invokes a synchronized method, the following occurs:

- It automatically acquires the intrinsic lock for that method's object.
- It releases it when the method returns.

The lock is released even if the return is caused by an uncaught exception.

If a static method is associated with a class, the thread gets the intrinsic lock for the `Class` object associated with the class. Therefore, the lock that controls the access to the class's static fields is different from the lock for any instance of the class.

Synchronized code can also be created with synchronized statements. These statements should specify the object that provides the intrinsic lock.

Synchronized statements also help improve concurrency as they prevent unnecessary blocking.

8.3.5 Reentrant Synchronization

Reentrant synchronization occurs when a thread acquires a lock that it already owns. In other words it means allowing a thread to acquire the same lock more than once. In this event, the synchronized code invokes a method directly or indirectly that also contains synchronized code. Both sets of code use the same lock.

With reentrant synchronization, it is easy for synchronized code to avoid a thread cause itself to block.

8.3.6 Atomic Access

In programming, an atomic action occurs all at once. Following are the features of an atomic action:

- It occurs completely, or it doesn't occur at all.
- Effects of an atomic action are visible only after the action is complete.

Following are the actions that can be specified as atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`).
- Reads and writes are atomic for all variables declared `volatile`.

Atomic actions cannot be interleaved. There will be no thread interference when atomic actions are used.

However, atomic actions should be synchronized to avoid memory consistency errors. Such errors can be reduced by using `volatile` variables. This is because any write to a `volatile` variable creates a happens-before relationship with successive reads of that variable. Therefore, changes to a `volatile` variable are always visible to other threads. When a thread reads a `volatile` variable, it sees the latest change to the `volatile` variable and the effects of the code that brings the change.

Note - It is recommended to use simple atomic variable access. This is because no effort is required to avoid memory consistency errors.

8.4 wait-notify Mechanism

Java also provides a wait and notify mechanism to work in conjunction with synchronization. The wait-notify mechanism acts as the traffic signal system in the program. It allows the specific thread to wait for some time for other running thread and wakes it up when it is required to do so. For these operations, it uses the `wait()`, `notify()` and `notifyAll()` methods.

In other words, the wait-notify mechanism is a process used to manipulate the `wait()` and `notify()` methods. This mechanism ensures that there is a smooth transition of a particular resource between two competitive threads. It also oversees the condition in a program where one thread is:

- Allowed to wait for the lock of a synchronized block of resource currently used by another thread.
- Notified to end its waiting state and get the lock of that synchronized block of resource.

8.4.1 `wait()` Method

The `wait()` method causes a thread to wait for some other thread to release a resource. It forces the currently running thread to release the lock or monitor, which it is holding on an object. Once the resource is released, another thread can get the lock and start running. The `wait()` method can only be invoked only from within the synchronized code.

The following points should be remembered while using the `wait()` method:

- The calling thread gives up the CPU and lock.
- The calling thread goes into the waiting state of monitor.

Syntax:

```
public final void wait()
```

Code Snippet 7 display the use of `wait()` method.

Code Snippet 7:

```
...
public synchronized void takeup()
{
    while (!available)
    {
        try
        {
            System.out.println("Philosopher is waiting for the other chopstick");
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    available=false;
}
...
```

This code snippet demonstrates the use of `wait()` method. The `wait()` method forces the currently running thread to pause so that another thread can get hold of the resource.

8.4.2 `notify()` Method

The `notify()` method alerts the thread that is waiting for a monitor of an object. This method can be invoked only within a synchronized block. If several threads are waiting for a specific object, one of them is selected to get the object. The scheduler decides this based on the need of the program.

The `notify()` method functions in the following way:

- The waiting thread moves out of the waiting space of the monitor and into the ready state.

- The thread that was notified is now eligible to get back the monitor's lock before it can continue.

Syntax:

```
public final void notify()
```

Code Snippet 8 displays the use of `notify()` method.

Code Snippet 8:

```
...
public synchronized void putdown ()
{
    available=true;
    notify();
}
...
```

This code illustrates the use of the `notify()` method. This method will notify the waiting thread to resume its execution.

8.5 Deadlocks

Deadlock describes a situation where two or more threads are blocked forever, waiting for the others to release a resource. At times, it happens that two threads are locked to their respective resources, waiting for the corresponding locks to interchange the resources between them. In that situation, the waiting state continues forever as both are in a state of confusion as to which one will leave the lock and which one will get into it. This is the deadlock situation in a thread based Java program. The deadlock situation brings the execution of the program to a halt.

Figure 8.4 displays a deadlock condition.

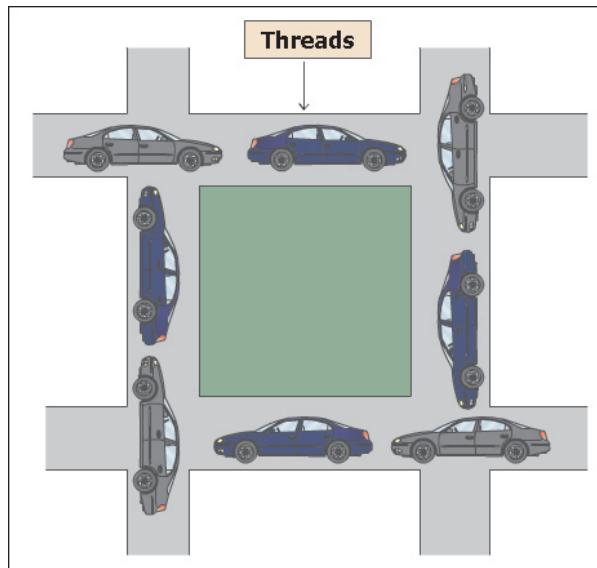


Figure 8.4: Deadlock

Note - Thread A can lock data structure X, and thread B can lock data structure Y. Then, if A tries to lock Y and B tries to lock X, both A and B will wait forever - B will wait for A to unlock X and A will wait for B to unlock Y.

It is difficult to debug a deadlock since it occurs very rarely.

Code Snippet 9 demonstrates a deadlock condition.

Code Snippet 9:

```
/**
 * Demonstrating Deadlock.
 */
package test;

/**
 * DeadlockDemo implements the Runnable interface.
 */
public class DeadlockDemo implements Runnable {
    public static void main(String args[]) {
        DeadlockDemo objDead1 = new DeadlockDemo();
        DeadlockDemo objDead2 = new DeadlockDemo();
        Thread objTh1 = new Thread(objDead1);
        Thread objTh2 = new Thread(objDead2);
    }
}
```

```
objDead1.grabIt=objDead2;
objDead2.grabIt=objDead1;
objTh1.start();
objTh2.start();
System.out.println("Started");
try {
    objTh1.join();
    objTh2.join();
} catch(InterruptedException e) {
    System.out.println("error occurred");
}
System.exit(0);
}
DeadlockDemo grabIt;
public synchronized void run() {
try {
    Thread.sleep(500);
} catch(InterruptedException e) {
    System.out.println("error occurred");
}
grabIt.syncIt();
}
public synchronized void syncIt() {
try {
    Thread.sleep(500);
    System.out.println("Sync");
} catch(InterruptedException e) {
    System.out.println("error occurred");
}
}
```

```
System.out.println("In the syncIt() method");
}
} // end class
```

The program creates two child threads. Each thread calls the synchronized `run()` method. When thread `objTh1` wakes up, it calls the method `syncIt()` of the `DeadlockDemo` object `objDead1`. Since the thread `objTh2` owns the monitor of `objDead2`, thread `objTh1` begins waiting for the monitor. When thread `objTh2` wakes up, it tries to call the method `syncIt()` of the `DeadlockDemo` object `objDead2`. At this point, `objTh2` also is forced to wait since `objTh1` owns the monitor of `objDead1`. Since both threads are waiting for each other, neither will wake up. This is a deadlock condition. The program is blocked and does not proceed further.

Figure 8.5 shows the output of the code.



Figure 8.5: Output – Deadlock Condition

8.5.1 Overcoming the Deadlock

You can plan for the prevention of deadlock while writing the codes. You can ensure any of the following things in a program to avoid deadlock situations in it:

- Avoid acquiring more than one lock at a time.
- Ensure that in a Java program, you acquire multiple locks in a consistent and defined order.

Note -

1. If a thread holds a lock and goes in sleeping state, it does not loose the lock. However, when a thread goes in the blocked state, it releases the lock. This eliminates potential deadlock situations.
2. Java does not provide any mechanism for detection or control of potential deadlock situations. The programmer is responsible for avoiding them.

8.6 Concurrency Utilities

Java 5 has introduced the `java.util.concurrent` package. This package contains classes that are useful for concurrent programming. Some of the features are as follows:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools

8.6.1 *java.util.concurrent Collections*

These are the various additions to the Java Collections Framework.

Note - All of these collections help avoid memory consistency errors by defining a happens-before relationship between an operation that adds an object to the collection with successive operations that access or remove that object.

Following are some of these collections that are categorized by the collection interfaces:

→ **BlockingQueue**: This defines a FIFO data structure that blocks or times out when data is added to a full queue or retrieved from an empty queue.

→ **ConcurrentMap**: This is a subinterface of `java.util.Map` that defines useful atomic operations. These operations add a key-value pair only if the key is absent. They can also remove or replace a key-value pair only if the key is present. Such operations can be made atomic to avoid synchronization.

Note - `ConcurrentHashMap` is the standard general-purpose implementation of `ConcurrentMap`. `ConcurrentHashMap` is a concurrent analog of `HashMap`.

→ **ConcurrentNavigableMap**: This is a subinterface of `ConcurrentMap` that supports approximate matches.

Note - `ConcurrentSkipListMap` is the standard general-purpose implementation of `ConcurrentNavigableMap`. `ConcurrentSkipListMap` is a concurrent analog of `TreeMap`.

8.6.2 *Atomic Variables*

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables. All classes include get and set methods that functions similar to reads and writes on volatile variables. Therefore, a set has a happens-before relationship with any successive get on the same variable.

Note - The `atomic compareAndSet` method includes memory consistency features similar to the simple atomic arithmetic methods that apply to integer atomic variables.

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

Code Snippet 10:

```
public class AtomicVariableApplication {
    private final AtomicInteger value = new AtomicInteger(0);
    public int getValue() {
        return value.get();
    }
}
```

```

public int getNextValue() {
    return value.incrementAndGet();
}

public int getPreviousValue() {
    return value.decrementAndGet();
}

public static void main(String[] args) {
    AtomicVariableApplication obj = new AtomicVariableApplication();
    System.out.println(obj.getValue());
    System.out.println(obj.getNextValue());
    System.out.println(obj.getPreviousValue());
}
}

```

Figure 8.6 displays the output of the code.

```

run:
0
1
0
BUILD SUCCESSFUL (total time: 3 seconds)

```

Figure 8.6: AtomicVariableApplication Class - Output

8.6.3 Lock Objects

The `java.util.concurrent.locks` package provides a framework for locking and waiting for conditions. Though they are similar to implicit locks used by synchronized locks but this type of lock is different from built-in synchronization and monitors.

The advantage of Lock objects over implicit locks is their ability to back out when it is trying to acquire a lock. The `tryLock()` method will back out if the lock is unavailable either immediately or before a timeout expires (if specified). The `lockInterruptibly()` method will back out when another thread sends an interrupt before the lock is acquired.

8.6.4 Executors and Executor Interface

Objects that separate thread management and creates them from the rest of the application are called executors.

The `java.util.concurrent` package defines the following three executor interfaces:

- **Executor:** This helps launch new tasks. The Executor interface includes a single method, execute. It is designed to be a drop-in replacement for a common thread-creation idiom. If `r` is a Runnable object and `e` is an Executor object, `(new Thread(r)).start();` can be replaced with `e.execute(r);`

The executor implementations in `java.util.concurrent` uses advanced `ExecutorService` and `ScheduledExecutorService` interfaces.

The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute uses an existing worker thread to run `r`. It can also place `r` object in a queue to wait for a worker thread to become available.

- **ExecutorService:** This is a subinterface of Executor and helps manage the development of the executor tasks and individual tasks. The interface provides execute with a resourceful submit method that accepts Runnable objects and Callable objects. The latter allow the task to return a value. The submit method returns a Future object, which retrieves the Callable return value. The object also manages the status of Callable and Runnable tasks.

`ExecutorService` provides methods to submit large collections of Callable objects. It also includes various methods to manage the shutdown of the executor.

Note - Tasks should manage interrupts well to support immediate shutdown.

- **ScheduledExecutorService:** This is a subinterface of `ExecutorService` and helps periodic execution of tasks.

Note - Variables referring to executor objects are declared as one of the executor interface types.

The `Scheduled` interface provides schedule for the methods of its parent `ExecutorService`. Schedule executes a Runnable or Callable task after a specified delay. The interface also defines `scheduleAtFixedRate` and `scheduleWithFixedDelay`. At defined intervals, these execute specified tasks repeatedly.

8.6.5 ThreadPools

Thread pools have worker threads that help create threads and thus **minimize the overhead**. Certain executor implementations in `java.util.concurrent` use thread pools. Thread pools are often used to **execute multiple tasks**.

Allocating and deallocating multiple thread objects creates a considerable memory management overhead in a large-scale application.

Fixed thread pool is a common type of thread pool that includes the following features:

- There are a specified number of threads running.

- When in use if a thread is terminated, it is automatically replaced with a new thread.
- Applications using fixed thread pool services HTTP requests as quickly as the system sustains.

Note - Internal queue holds extra tasks. Tasks are submitted to the pool through an internal queue.

Invoking the `newFixedThreadPool` factory method in `java.util.concurrent.Executors` creates an executor that uses a fixed thread pool. This class provides the following factory methods:

- `newCachedThreadPool` method: This creates an executor with an expandable thread pool. This is suitable for applications that launch many short-lived tasks.
- `newSingleThreadExecutor` method: This creates an executor that executes a one task at a time.

Note - Other factory methods include various `ScheduledExecutorService` versions of the executors created by `newCachedThreadPool` and `newSingleThreadExecutor` methods.

8.6.6 Fork/Join Framework

This is an implementation of the `ExecutorService` interface. The framework helps work with several processors to boost the performance of an application. It uses a work-stealing algorithm and is used when work is broken into smaller pieces recursively.

The Fork/Join framework allocates tasks to worker threads in a thread pool.

There is the `ForkJoinPool` class in the fork/join framework. The class is an extension of the `AbstractExecutorService` class. The `ForkJoinPool` class implements the main work-stealing algorithm and executes `ForkJoinTask` processes.

The following describes the basic steps to use the fork/join framework:

1. Write the code that performs a segment of the work. The code should resemble the following pseudocode:

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

2. Wrap the code in a `ForkJoinTask` subclass, typically using `RecursiveTask` that returns a result or `RecursiveAction`.
3. Create the object for all the tasks to be done.
4. Pass the object to the `invoke()` method of a `ForkJoinPool` instance.

Code Snippet 11 displays the use of Fork/Join functionality.

Code Snippet 11:

```
package threadapplication;
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class ForkJoinApplication extends RecursiveTask<Integer> {
    private static final int SEQUENTIAL_THRESHOLD = 5;
    private final int[] data;
    private final int startData;
    private final int endData;
    public ForkJoinApplication(int[] data, int startValue, int endValue) {
        this.data = data;
        this.startData = startValue;
        this.endData = endValue;
    }
    public ForkJoinApplication(int[] data) {
        this(data, 0, data.length);
    }
    //recursive method which forks all small work units and then joins them
    @Override
    protected Integer compute() {
        final int length = endData - startData;
        if (length < SEQUENTIAL_THRESHOLD) {
            return computeDirectly();
        }
        final int midValue = length / 2;
        final ForkJoinApplication leftValues = new ForkJoinApplication(data,
                startData, startData + midValue);
        //forks all the small work units
        leftValues.fork();
```

```
final ForkJoinApplication rightValues = new ForkJoinApplication(data,
    startData + midValue, endData);

//joins them all again using the join method

return Math.max(rightValues.compute(), leftValues.join());
}

private Integer computeDirectly() {
    System.out.println(Thread.currentThread() + " computing: " + startData
        + " to " + endData);

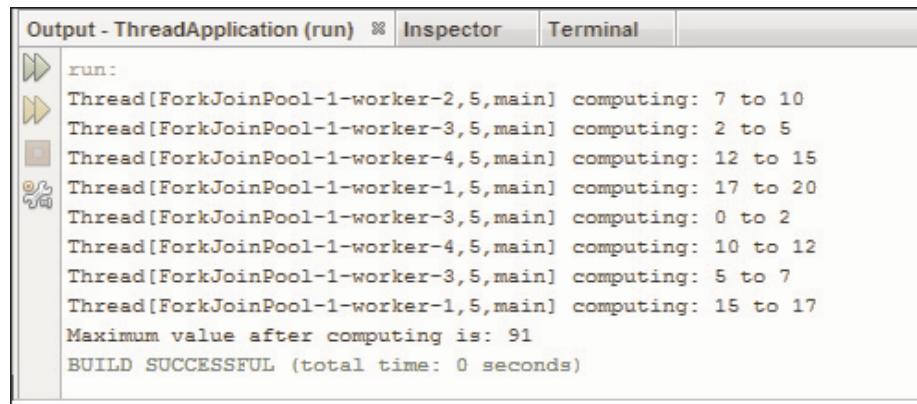
    int max = Integer.MIN_VALUE;
    for (int i = startData; i < endData; i++) {
        if (data[i] > max) {
            max = data[i];
        }
    }

    return max;
}

public static void main(String[] args) {
    // create a random object value set
    final int[] value = new int[20];
    final Random randObj = new Random();
    for (int i = 0; i < value.length; i++) {
        value[i] = randObj.nextInt(100);
    }

    // submit the task to the pool
    final ForkJoinPool pool = new ForkJoinPool(4);
    final ForkJoinApplication maxFindObj = new ForkJoinApplication(value);
    // invokes the compute method
    System.out.println(pool.invoke(maxFindObj));
}
}
```

Figure 8.7 displays the output.



The screenshot shows a terminal window titled "Output - ThreadApplication (run)". The window has tabs for "Inspector" and "Terminal". The terminal pane displays the following text:

```
run:  
Thread[ForkJoinPool-1-worker-2,5,main] computing: 7 to 10  
Thread[ForkJoinPool-1-worker-3,5,main] computing: 2 to 5  
Thread[ForkJoinPool-1-worker-4,5,main] computing: 12 to 15  
Thread[ForkJoinPool-1-worker-1,5,main] computing: 17 to 20  
Thread[ForkJoinPool-1-worker-3,5,main] computing: 0 to 2  
Thread[ForkJoinPool-1-worker-4,5,main] computing: 10 to 12  
Thread[ForkJoinPool-1-worker-3,5,main] computing: 5 to 7  
Thread[ForkJoinPool-1-worker-1,5,main] computing: 15 to 17  
Maximum value after computing is: 91  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 8.7: Fork/Join Functionality - Output

8.7 Check Your Progress

1. Which one of the following can cause race condition?

A.	When two or more threads share the same data between them
B.	The use of synchronized blocks
C.	Releasing the intrinsic lock
D.	Interleaving the atomic actions

(A)	A	(C)	C
(B)	B	(D)	D

2. Which of the following statements are true for multithreading?

A.	Multithreading requires less overhead
B.	In a multithreaded environment two or more processes run concurrently
C.	Multithreading is a specialized form of multitasking
D.	In a multithreaded program, two or more threads can run concurrently

(A)	A, C, and D	(C)	C and D
(B)	B, C, and D	(D)	B and D

3. Which one of the following acts as the traffic signal system in a program?

A.	Locks
B.	Race condition
C.	The wait-notify mechanism
D.	Atomic variables

4. Which one of the following brings the execution of the program to a halt?

A.	Threadpools
B.	Executors
C.	Race condition
D.	Deadlock

5. Which of the following options do the executor implementations in `java.util.concurrent` use?

A.	ExecutorService interface
B.	Runnable interface
C.	Collections interface
D.	ScheduledExecutorService interface

6. The _____ is a piece of code that is guarded by a mutual-exclusion program.

A.	Mutex
B.	Monitor
C.	synchronized keyword
D.	Lock

8.7.1 Answers

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

Summary

- Multithreading is nothing but running of several threads in a single application.
- The isAlive() method tests whether the thread is in runnable, running, or terminated state.
- The join() method forces a running thread to wait until another thread completes its task.
- Race condition can be avoided by using synchronized block.
- The wait() method sends the running thread out of the lock or monitor to wait.
- The notify() method instructs a waiting thread to get in to the lock of the object for which it has been waiting.
- Deadlock describes a situation where two or more threads are blocked forever, waiting for each to release a resource.

Session 9

JDBC API

Welcome to the Session, **JDBC API**.

This session introduces you to a software API for database connectivity, explores JDBC and its architecture. It also discusses how to use JDBC and its drivers to develop a database application. The session also describes how to connect to and retrieve data from an SQL Server database. Finally, the module explains about database meta data information.

In this Session, you will learn to:

- Explain basics of JDBC
- Explain JDBC architecture
- Explain different types of processing models
- Explain JDBC Driver Types
- Understand the steps involved in JDBC application development
- Explain Database Meta Information
- Explain parameterized queries in JDBC API



9.1 Introduction

A database contains data that is in an organized form.

Client/server applications make extensive use of database programming. Typical activities involved in a database application involve opening a connection, communicating with a database, executing SQL statements, and retrieving query results.

Java has established itself as one of the prime backbones of enterprise computing.

The core of Java enterprise applications depends on Database Management Systems (DBMS), which acts as the repository for an enterprise's data. Hence, to build such applications, these databases in the repository need to be accessed. To connect the Java applications with the databases, the Application Programming Interfaces (APIs) software for database connectivity, known as JDBC, is used. This software API is a collection of application libraries and database drivers, whose implementation is independent of programming languages, database systems, and operating systems.

Open DataBase Connectivity (ODBC) and JDBC are two widely used APIs for such activities.

Figure 9.1 displays database connectivity.

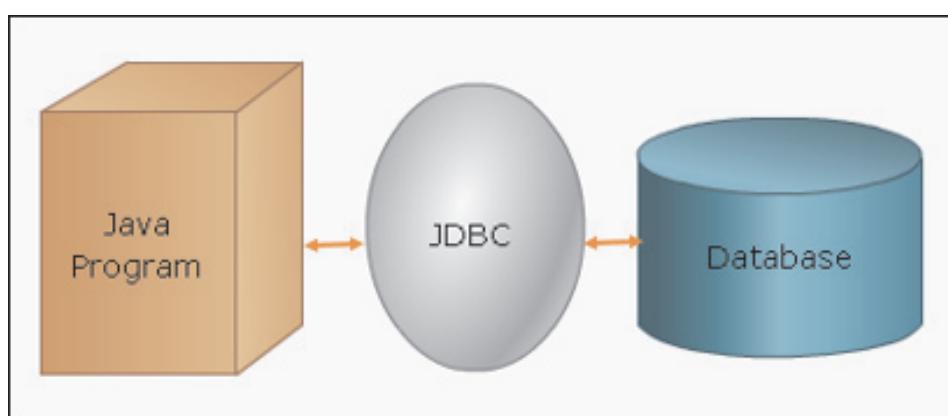


Figure 9.1: Database Connectivity

9.2 ODBC

ODBC is an API provided by Microsoft for accessing the database. It uses Structured Query Language (SQL) as its database language. It provides functions to insert, modify, and delete data and obtain information from the database. Figure 9.2 displays the ODBC connection.

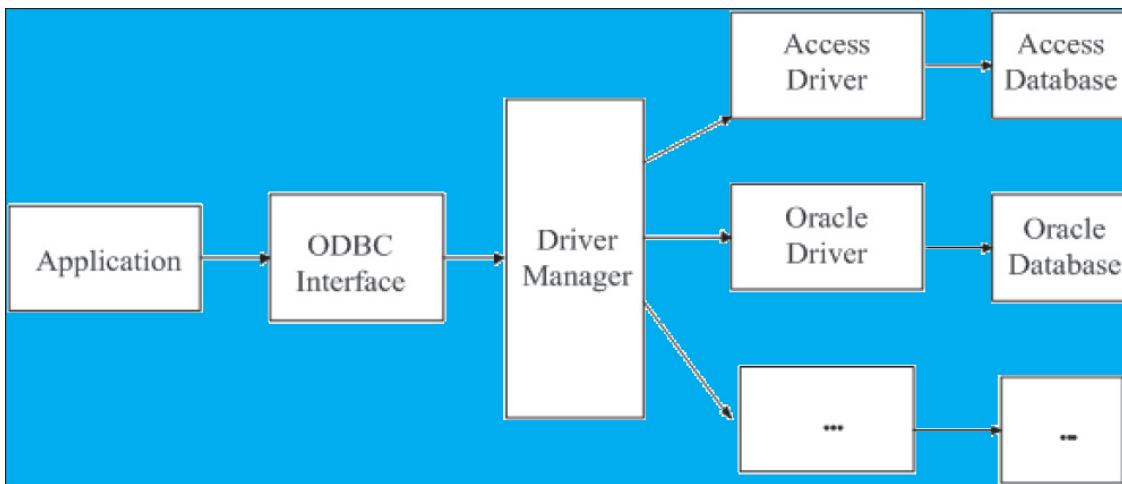


Figure 9.2: ODBC Connection

The application can be a GUI program written in Java, VC++, or any other software. The application makes use of ODBC to connect with the databases. The driver manager is part of Microsoft ODBC API and is used to manage various drivers in the system including loading. In cases where the application uses multiple databases, it is the function of the driver manager to make sure that the function calls get diverted to the correct Database Management System (DBMS). The driver is the actual software component that knows about the database. Drivers are software specific such as Microsoft Access Driver, Oracle Driver and so on.

9.2.1 Definitions of JDBC

JDBC is the Java-based API, which provides a set of classes and interfaces written in Java to access and manipulate different kinds of databases. Java Database Connectivity (JDBC) API is a Java API provided by Sun. The JDBC API has a set of classes and interfaces used for accessing tabular data. These classes and interfaces are written in Java programming language and provides a standard API for database developers. To access data quickly and efficiently from databases Java applications uses JDBC. The advantage of using JDBC API is that an application can access any database and run on any platform having Java Virtual Machine. In other words, a Java application can write a program using JDBC API, and the SQL statement can access any database.

The combination of JDBC API and Java platform offers the advantage of accessing any type of data source and flexibility of running on any platform which supports Java virtual machine (JVM). For a developer, it is not necessary to write separate programs to access different databases like SQL Server, Oracle or IBM DB2. Instead, a single program with the JDBC implementation can send Structured Query Language (SQL) or other statements to the suitable data source or database. The three tasks that can be performed by using JDBC drivers are: establish a connection with the data source, send queries, and update statements to the data source and finally process the results.

Note - JDBC is not an acronym, though it is often mistaken to be “Java Database Connectivity”. It is actually the brand name of the product.

9.2.2 Need for JDBC

JDBC helps to establish connection with the database. The following paragraph explains the need for JDBC.

- **ODBC uses a C interface.** Calls from Java to native C code have a number of drawbacks in the areas of security, implementation, robustness, and automatic portability of applications. This hampers the use of ODBC driver in a Java application.
- A literal translation of the ODBC C interface into a Java API will not be desirable. For example, Java has no pointers, and ODBC makes use of these, including the error-prone generic pointer “void *”. In simple terms, JDBC is ODBC translated into an object-oriented interface that is natural for Java programmers.
- ODBC mixes simple and advanced features together, and it has complex options even for simple queries. JDBC has an upper hand as it was designed to keep things simple while allowing more advanced capabilities wherever required.
- A Java API like JDBC is needed in order to enable a “pure Java” solution. When ODBC is used, the ODBC driver manager and drivers need to be installed manually on every client machine. However, since the JDBC driver is written completely in Java, the JDBC code is installable, portable and secure on all Java platforms from network computers to mainframes.
- JDBC is a standard interface for Java programs to access relational databases.

Thus, it can be said that the JDBC API is a natural Java interface to the basic SQL concepts. JDBC have been built on ODBC and thus retains the basic design features of ODBC. In fact, both interfaces are based on the X/Open SQL Call Level Interface (CLI).

→ Product Components of JDBC

The four product components of JDBC are:

- **JDBC API**

JDBC API is a part of Java platform. JDBC 4.0 APIs are included in two packages which are included in Java Standard Edition and Java Enterprise Edition. The two packages are `java.sql` and `javax.sql`. JDBC API allows access to the relational database from the Java programming language. Thus, JDBC API allows applications to execute SQL statements, retrieve results, and make changes to the underlying database.

- **JDBC Driver Manager**

A Java application is connected to a JDBC driver using an object of the `DriverManager` class. The `DriverManager` class is the backbone of the JDBC architecture. The task performed by the `DriverManager` class are first to **locate the driver for a specific database** and then **process the initialization calls for JDBC**.

- **JDBC Test Suite**

JDBC driver test suite helps to determine that JDBC drivers will run the program.

- **JDBC-ODBC Bridge**

JDBC access is provided using ODBC drivers.

9.3 JDBC Architecture

JDBC is one of the core part of Java platform and is included in the distribution of JDK. The main task of JDBC API is to allow the developers to execute the SQL statements in an independent manner irrespective of the underlying database. The classes and interfaces of JDBC API are used to represent objects of database connections, SQL statements, Result sets, Database metadata, Prepared statements, Callable statements, and so on.

To provide connectivity to heterogeneous databases, the driver manager and database specific drivers are used by JDBC API. The driver manager ensures that correct drivers are used to access the database. Multiple drivers connected to access different data source is supported by the driver manager. Figure 9.3 shows the location of the driver manager with respect to JDBC drivers and applications.

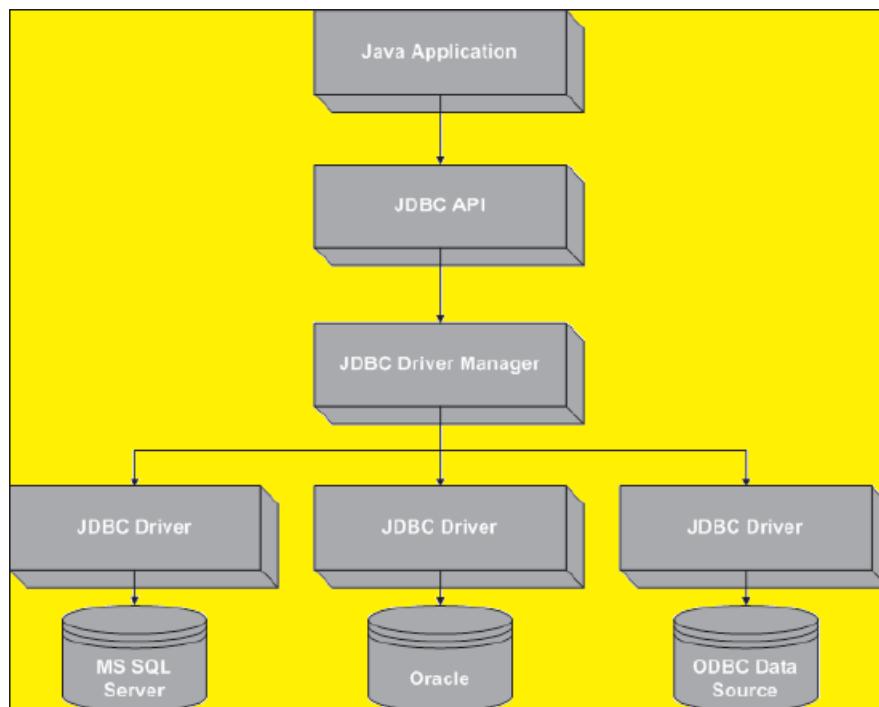


Figure 9.3: JDBC Architecture

9.3.1 Advantages of JDBC

Some of the advantages of JDBC are:

→ **Continued usage of existing data**

JDBC enables enterprise applications to continue using existing data even if the data is stored on different database management systems.

→ **Vendor independent**

The combination of the Java API and the JDBC API makes the databases transferable from one vendor to another without modifications in the application code.

→ **Platform independent**

JDBC is usually used to connect a user application to a “behind the scenes” database, no matter of what database management software is used to control the database. In this fashion, JDBC is **cross-platform or platform independent**.

→ **Ease of use**

With JDBC, the complexity of connecting a user program to a “behind the scenes” database is hidden, and makes it easy to deploy and economical to maintain.

9.3.2 Two-Tier Data Processing Model

The JDBC API supports **two-tier as well as three-tier data processing models for accessing database**.

In a **two-tier** client/server system, the **client communicates directly to the database server without the help of any middle-ware technologies** or another server. In a two-tier JDBC environment, the **Java application** is the **client** and **DBMS** is the **database server**.

The typical implementation of a two-tier model involves the use of JDBC API to translate and send the client request to the database. The database may be located on the same or another machine in the network. The results are delivered back to the client again through the JDBC API. Figure 9.4 displays the two-tier data processing model.

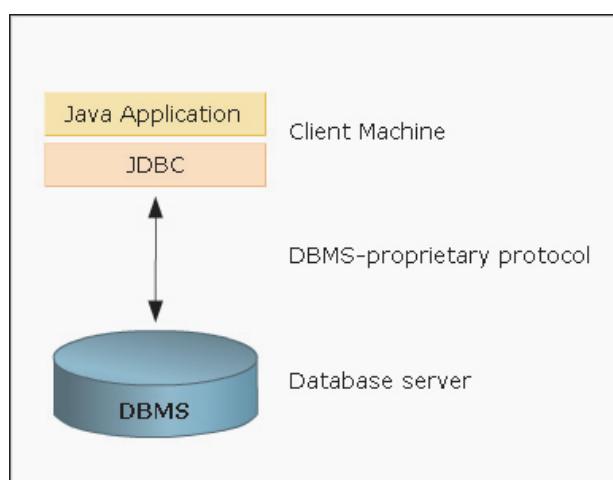


Figure 9.4: Two-Tier Data Processing Model

9.3.3 Three-Tier Data Processing Model

In a three-tier model, a “middle tier” of services, a third server is employed to send the client request to the database server. This middle-tier helps in separating the database server from the Web server. The involvement of this third server or proxy server **enhances the security by passing all the requests to the database server through the proxy server.** The database server processes the requests and sends back the results to the middle tier (proxy server), which again sends it to the client. Figure 9.5 displays the three-tier data processing model.

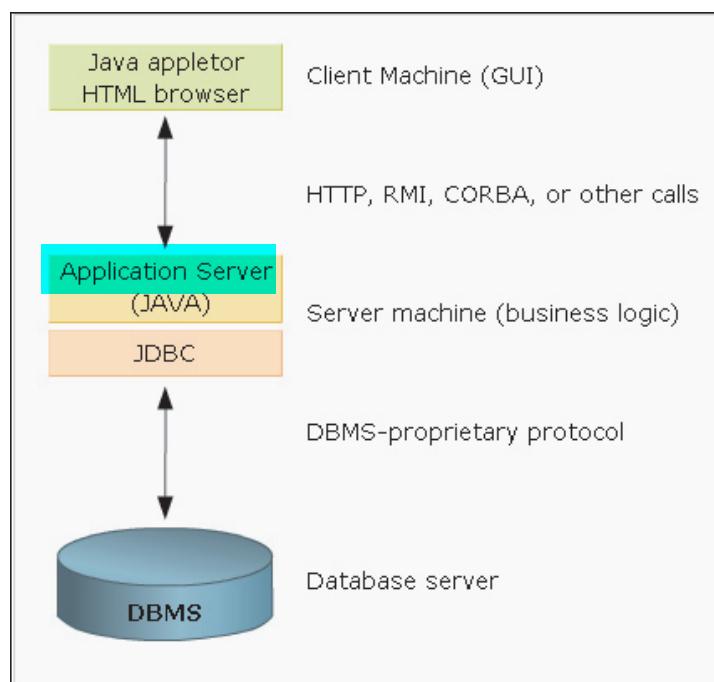


Figure 9.5: Three-Tier Data Processing Model

Note - The advantages of a three-tier model over a two-tier model are that it **simplifies and reduces the cost of application deployment**. Also it **offers regulated access and modifications to databases**.

9.3.4 JDBC API

JDBC API is a collection of specifications that defines the way how database and the applications communicate with each other. The core of JDBC API is based on Java, so, it is used as the common platform for building the middle-tier of three-tier architecture.

Hence, JDBC API being a middle-tier, it defines how a connection is opened between an application and database; how requests are communicated to a database, the SQL queries are executed, and the query results retrieved. JDBC achieves these targets through a set of Java interfaces, implemented separately by a set of classes for a specific database engine known as JDBC driver.

Figure 9.6 displays the JDBC-API.

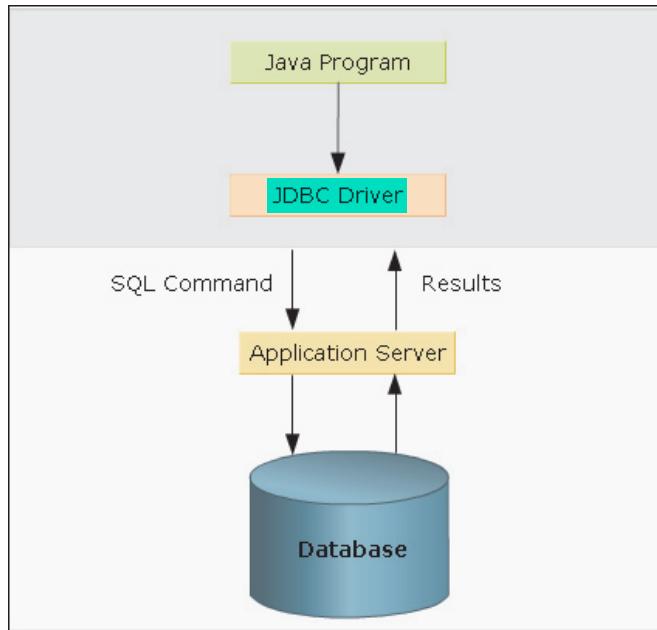


Figure 9.6: JDBC-API

9.4 JDBC Driver Types

The JDBC driver is the base of JDBC API and is responsible for ensuring that an application has a consistent and uniform access to any database. The driver converts the client request to a database understandable, native format and then presents it to the database. The response is also handled by the JDBC driver, and gets converted to the Java format and presented to the client.

The four types of drivers and a brief description of their basic properties are listed in table 9.1.

Driver Type	Driver Name	Description
Type I	ODBC-JDBC Bridge	Translates JDBC calls into ODBC calls <small>Ad: works on Windows</small>
Type II	Native API-Java/ Partly Java	Translates JDBC calls into database-specific calls or native calls
Type III	JDBC Network-All Java <small>three-tier</small>	Maps JDBC calls to the underlying “network” protocol, which in turn calls native methods on the server
Type IV	Native Protocol-All Java	Directly calls RDBMS from the client machine

two-tier

Table 9.1: Types Of Drivers

9.4.1 JDBC Type 1 Driver

The Type 1 driver is a Java software bridge product, also known as JDBC-ODBC bridge plus ODBC driver.

Figure 9.7 displays the JDBC type 1 driver.

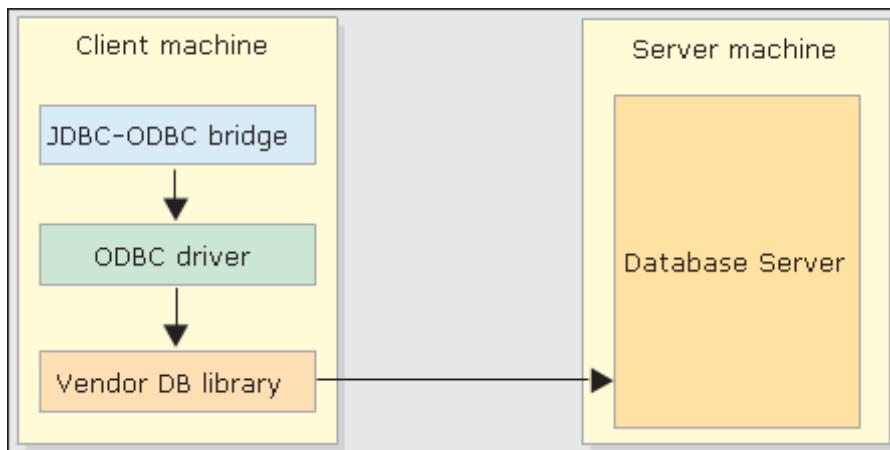


Figure 9.7: JDBC Type 1 Driver

→ Features

The Type 1 drivers use a bridging technology that provides JDBC access via ODBC drivers. This establishes a link between JDBC API and ODBC API. The ODBC API is in turn implemented to get the actual access to the database via the standard ODBC drivers.

Client machine needs to install native ODBC libraries, drivers, and required support files, and in most of the cases, the database client codes. This kind of driver is appropriate for an enterprise network, where installing client is not a problem.

→ Advantages

The Type 1 drivers are written to allow access to various databases through pre-existing ODBC drivers. In some cases, they are the only option to databases like MS-Access or Microsoft SQL Server for having ODBC native call interface.

→ Disadvantages

The Type 1 driver does **not hold good for applications that do not support software installations on client machines**. The native ODBC libraries and the database client codes must reside on the server, which in turn reduces the performance.

9.4.2 JDBC Type 2 Driver

The Type 2 driver is also known as Native-API partly Java driver.

Figure 9.8 displays the JDBC type 2 driver.

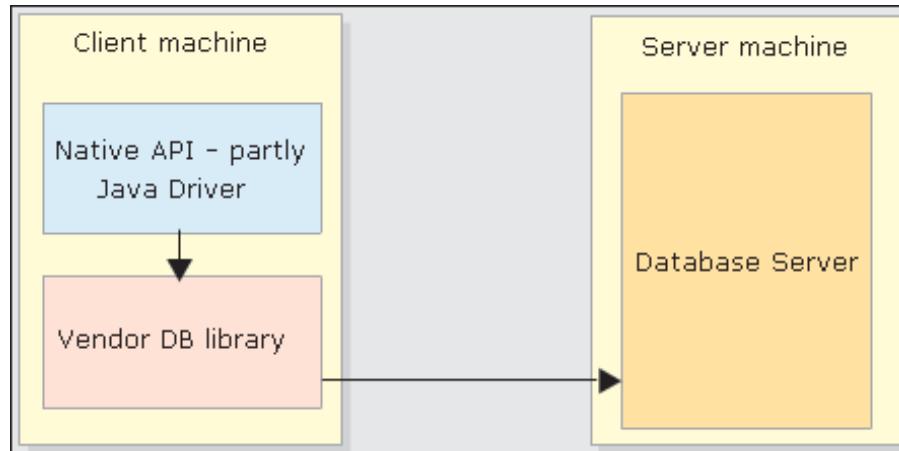


Figure 9.8: JDBC Type 2 Driver

→ Features

The Type 2 driver comprises the Java code that converts JDBC calls into calls on a local database API for Oracle, Sybase, DB2, or any other type of DBMS. This implies that the driver calls the native methods of the individual database vendors to get the database access.

This kind of driver basically comes with the database vendors to interpret JDBC calls to the database-specific native call interface, for example, Oracle provides OCI driver. The Type 2 driver also needs native database-specific client libraries to be installed and configured on the client machine like that of Type 1 drivers.

→ Advantages

The Type 2 driver yields better performance than that of Type 1 driver. Type 2 drivers are generally faster than Type 1 drivers as the calls get converted to database-specific calls.

→ Disadvantages

The Type 2 driver does not support applications that do not allow software installations on client machines as it requires native database codes to be configured on client machines. These database specific native code libraries must reside on the server, in turn reducing the performance.

9.4.3 JDBC Type 3 Driver

The Type 3 drivers are also known as JDBC-Net pure Java driver.

Figure 9.9 displays the JDBC type 3 driver.

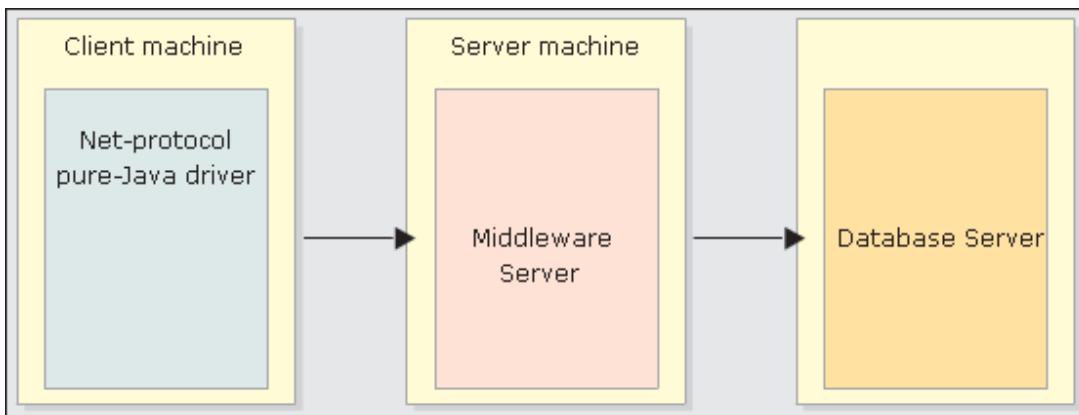


Figure 9.9: JDBC Type 3 Driver

→ Features

The Type 3 driver is a pure Java driver that converts the JDBC calls into a DBMS-independent network protocol, which is again translated to database-specific calls by a middle-tier server. This driver does not require any database-specific native libraries to be installed on the client machines. The Web-based applications should preferably implement Type 3 drivers as this driver can be deployed over the Internet without installing a client.

→ Advantages

The Type 3 driver is the most flexible type as it does not require any software or native services to be installed on client machine. It provides a high degree of adaptability to change and control underlying database without modifying the client side driver.

→ Disadvantages

Database-specific code needs to be executed in the middle-tier server. As it supports Web-based applications, it **needs to implement additional security like access through firewalls.**

9.4.4 JDBC Type 4 Driver

The Type 4 drivers are also known as Native-protocol pure Java driver or Java to Database Protocol.

Figure 9.10 displays the JDBC type 4 driver.

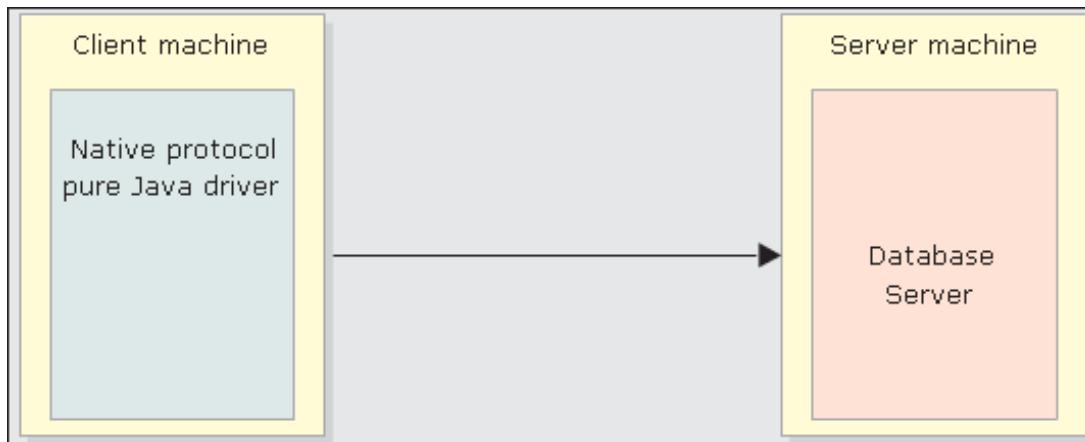


Figure 9.10: JDBC Type 4 Driver

→ Features

Type 4 drivers are pure Java drivers that convert JDBC calls into the network protocol that communicates directly with the database. This links the client call directly with the DBMS server and provides a practical solution for accessing Intranet. In most cases, the drivers are provided by the database vendors. These drivers also do not need any database-specific native libraries to be configured on client machine and can be deployed on the Web without installing a client, as required for Type 3 drivers.

→ Advantages

Type 4 drivers communicate directly with the database engine using Java sockets, rather than through middleware or a native library. This is the reason that these drivers are the fastest JDBC drivers available. No additional software like native library is needed for installation on clients.

→ Disadvantages

The only drawback in Type 4 drivers is that they are **database-specific**. Hence, if **in case, the back-end database changes, the application developer may need to purchase and deploy a new Type 4 driver specific to the new database.** change database -> change driver

Type 1 driver is **helpful for prototyping** and not for production. **Type 3 driver adds security, caching, and connection control.** **Type 3 and Type 4 driver need not be pre-installed and are portable.**

9.5 *java.sql Package*

JDBC API defines a set of interfaces and classes to communicate with the database. They are contained in the `java.sql` package. The API has different framework where the drivers can be installed dynamically to access different data sources. The JDBC API not only passes SQL statements to a database but it also provides facility for reading and writing data from any data source with a tabular format. Some of the interfaces in this package have been summarized in table 9.2.

Interface	Description
Connection	This is used to maintain and monitor database sessions. Data access can also be controlled using the transaction locking mechanism.
DatabaseMetaData	This interface provides database information such as the version number, names of the tables, and functions supported. It also has methods that help in discovering database information such as the current connection, tables available, schemas, and catalogues.
Driver	This interface is used to create Connection objects.
PreparedStatement	This is used to execute pre-compiled SQL statements.
ResultSet	This interface provides methods for retrieving data returned by a SQL statement.
ResultSetMetaData	This interface is used to collect the meta data information associated with the last ResultSet object. print result based on user's requirements
Statement	It is used to execute SQL statements and retrieve data into the ResultSet.

Table 9.2: java.sql Package

Some of the classes included in the `java.sql` package are shown in table 9.3.

Class Name	Description
Date	This class contains methods for performing conversion of SQL date formats to Java Date formats.
DriverManager	This class is used to handle loading and unloading of drivers and establish connection with the database.
DriverPropertyInfo	The methods in this class are used to retrieve or insert driver properties.
Time	This class provides formatting and parsing operations for time values.

Table 9.3: Classes in java.sql Package

The `exceptions` defined by the `java.sql` package are listed in table 9.4.

Exception	Description
DataTruncation	This exception is raised when a data value is truncated . <small>data values exceed the limit</small>
SQLException	This exception provides information on database access errors or other errors.
SQLWarning	This exception provides information on database access warnings.
BatchUpdateException	This exception is raised when an error occurs during batch update operation.

Table 9.4: Exceptions in java.sql Package

9.5.1 Steps to Develop a JDBC Application

The process of accessing a database and processing queries via JDBC involves five basic steps:

→ **Register the JDBC Driver**

The first step in a database connection is to register the JDBC driver with the `DriverManager` class. The `DriverManager` is a static class, which manages the set of JDBC drivers available for an application. It is the responsibility of `DriverManager` responsibility to manage all the references to all the driver objects that are available to a JDBC client.

→ **Establish a Database Connection**

After the driver is loaded, the connection with the database can be established. A database Uniform Resource Locator (URL) identifies a JDBC connection and tells the driver manager which driver and data source to use.

→ **Create and Execute an SQL Statement**

Once a connection is established with the database, the SQL statements can be submitted to the database for processing. SQL statement is the universally accepted query language to perform retrieval, insertion, updation, or deletion operations on a database.

→ **Process the Results**

After executing the SQL statements, the result is processed, and extracted by using the `ResultSet` object.

→ **Close the Database Connection**

As database connection is an important and limited resource, the connection should be closed using the `close()` method once the processing is done.

Figure 9.11 displays the steps to develop a JDBC application.

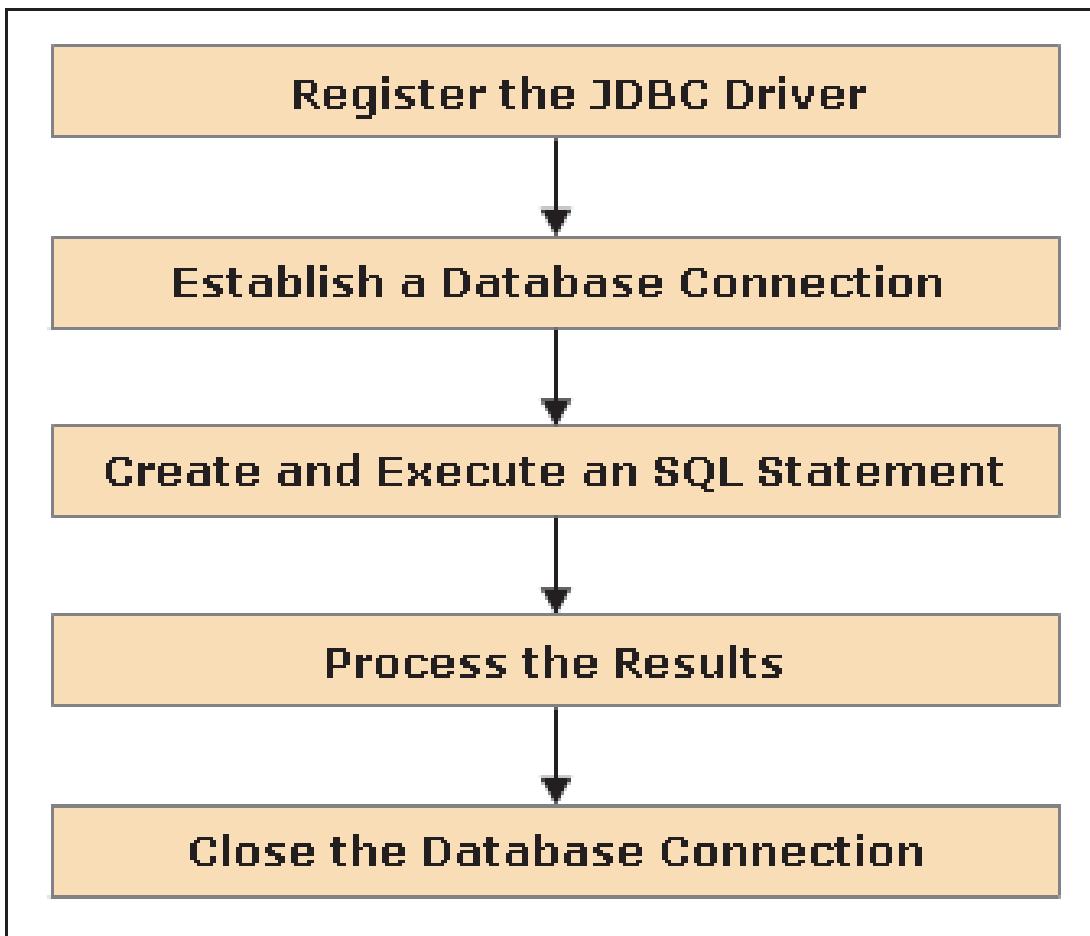


Figure 9.11: Steps to Develop a JDBC Application

9.5.2 Loading a Driver

The first step in establishing a database connection and using a JDBC driver, is to load the driver class by using the `Class.forName()` method. Invoking the `Class.forName()` method creates an instance of the specified driver class and registers it with the `DriverManager` class. Once the driver is loaded successfully, the connection with a database can be established.

Syntax:

`Class.forName(<protocol>)`

where,

`protocol` is the specification for the driver to be used.

Code Snippet 1 shows how to load the driver class.

Code Snippet 1:

```
Class.forName("jdbc.odbc.JdbcOdbcDriver");
```

This line of code sets up a connection with a Type 1, JDBC-ODBC bridge driver.

9.5.3 Establishing a Connection

The second step in establishing a database connection is to connect the loaded driver to the DBMS. The request for this connection is done by using the `DriverManager.getConnection()` method, which checks all the available drivers for the eligibility to make a connection. This also checks for a driver that recognizes the URL, sent by the client. Establishing a connection is achieved by the use of following elements:

→ Connection URL

The connection URL specifies necessary information that will be required for connecting to a particular database.

Syntax:

```
protocol:<subprotocol>:<subname>
```

The component protocol in the JDBC URL identifies the protocol name for accessing the database. The second component subprotocol recognizes the underlying database source. The last component subname identifies the database name.

Code Snippet 2 shows a sample connection URL.

Code Snippet 2:

```
jdbc:odbc:demo
```

In this url, `jdbc` is used to access the database, `odbc` is the database source for any database accessed via a JDBC-ODBC bridge driver. The database name on a local machine is represented by `demo`. The attributes `username` and `password` are provided with the attribute values.

→ DriverManager.getConnection()

The calling of `getConnection()` method involves a two-step process. The first step, matching the URL and Driver, is followed by connecting to the database, which is the second step.

Syntax:

```
Connection cn = DriverManager.getConnection(<connection url>, <username>, <password>);
```

The `getConnection()` method usually accepts two arguments. The first is the connection URL string of the database and the second is the set of login attributes like `username` and `password`.

Code Snippet 3 shows `DriverManager.getConnection()`.

Code Snippet 3:

```
Connection cn = DriverManager.getConnection("jdbc:odbc:demo", "sa", "playware");
```

Here, the URL used is `jdbc:odbc:demo`. The next string is the `username` `sa` and finally the `password` is `playware`.

9.5.4 Creating Statements and Queries

Once a connection is established with a database, a Statement object needs to be created for query execution. The Statement object is the most frequently used object to execute SQL queries that do not need any parameters to be passed. The Statement object is created by using the Connection.createStatement() method.

A Statement object can be classified into three categories based on the type of SQL statements sent to the database. Statement and PreparedStatement is inherited from Statement interface. The CallableStatement is inherited from the PreparedStatement interface and executes a call to stored procedure. A PreparedStatement object executes a precompiled SQL statement with or without IN parameters.

Syntax:

```
public Statement createStatement() throws SQLException
```

where,

Statement is the Statement object that is returned by the execution of the createStatement() method.

Code Snippet 4 demonstrates creating a Statement object.

Code Snippet 4:

```
Connection cn = DriverManager.getConnection("jdbc:odbc:demo", "sa",
"playware");
Statement st = cn.createStatement();
```

Here, st is the Statement object, which is created and associated with a single connection object cn.

9.5.5 Using executeQuery() and ResultSet Objects

A Statement object when created has methods to perform various database operations.

The executeQuery() method is one of those methods that retrieves information from the database. It accepts a simple SQL SELECT statement as a parameter and returns the database rows in form of a ResultSet object. Hence, the database query execution is based on the following:

→ executeQuery()

This method is used to execute any SQL statement with a “SELECT” clause, that return the result of the query as a result set. It takes the SQL query string as the parameter.

→ ResultSet object

ResultSet objects are used to receive and store the data in the same form as it is returned from the SQL queries. The ResultSet object is generally created by using the executeQuery() method.

The `ResultSet` object contains the data returned by the query as well as the methods to retrieve data.

Syntax:

```
public ResultSet executeUpdate(String sql) throws SQLException  
where,
```

`sql` is a `String` object containing SQL statement to be sent to the database.

`ResultSet` object is created to store the result of SQL statement.

Code Snippet 5 demonstrates use of the `executeQuery()` method to select some rows.

Code Snippet 5:

```
ResultSet rs = st.executeQuery("SELECT * FROM Department");
```

Here, `rs` is the `ResultSet` object created to store the result of the `SELECT` statement.

9.5.6 Using `executeUpdate()` and `execute()` Methods

The other methods of `Statement` interface are:

→ **`executeUpdate()`**

The `executeUpdate()` method is used to execute `INSERT`, `DELETE`, `UPDATE`, and other SQL DDL (Data Definition Language) such as `CREATE TABLE`, `DROP TABLE`, and so on. The method returns an integer value indicating the row count.

Syntax:

```
public int executeUpdate(String sql) throws SQLException  
where,
```

`sql` is `String` object created to store the result of SQL statement,

`int` is a integer value storing the number of affected rows.

→ **`execute()`**

The `execute()` method is used to execute SQL statements that returns more than one result set. The method returns true if a result set object is generated by the SQL statements.

Syntax:

```
public boolean execute (String sql) throws SQLException  
where,  
sql is String object created to store the result of SQL statement.
```

9.5.7 Creating Parameterized Queries

The `PreparedStatement` object implements the execution of dynamic SQL statements with parameters. Hence, to use a parameterized query, the first step should be the creation of `PreparedStatement` object. As done for `Statement` objects, the `PreparedStatement` object also need to be created by a `Connection` object.

→ Creating Parameterized Query

The `PreparedStatement` object is created by using the `prepareStatement()` method of `Connection` class. This method accepts an SQL statement as an argument, unlike the `Statement` object. As this is supposed to be a pre-compiled statement, the database must know the SQL statement.

For runtime parameters to a dynamic SQL statement, a placeholder “?” is used to indicate that a variable is expected in that place.

Code Snippet 6 shows how to create parameterized query.

Code Snippet 6:

```
String sqlStmt = "UPDATE Employees SET Dept_ID = ? WHERE Dept_Name LIKE ?";  
PreparedStatement pStmt = cn.prepareStatement(sqlStmt);
```

The `PreparedStatement` object `pStmt` is created and assigned the SQL statement, “`UPDATE Employees SET Dept_ID = ? WHERE Dept_Name LIKE ?`”, which is a pre-compiled query.

→ Passing Parameters

At the time of compilation, the parameter values are passed and used in the place of “?” placeholder. While compiling, the placeholder becomes a part of the statement and appears as static to the compiler. Hence, database does not have to recompile the statement regardless what value is assigned to the variable. To substitute the “?” placeholder with a supplied value, one of the `setXXX()` method of the required primitive type is used. For example, If an integer value needs to be set in the placeholder then the `setInt()` method is used.

Code Snippet 7 shows how to pass parameters.

Code Snippet 7:

```
pStmt.setInt(1, 25);  
pStmt.setString(2, "Production");
```

The first line of code sets the first question mark placeholder to a Java int with a value of 25. The “1” indicates which question mark (“?”) place holder to be set and “25” is the value for the question mark (“?”) place holder. The second line of code sets the second placeholder parameter to the string “Production”.

→ Executing Parameterized Query

The executeUpdate() method is used to execute both the Statement and the PreparedStatement objects. This is associated with the tasks like update, insert, and delete. The return type of this method is integer, which specifies the number of rows affected by that operation.

Code Snippet 8 demonstrates the executeUpdate() method.

Code Snippet 8:

```
pStmt.executeUpdate();
```

Here, no argument is supplied to executeUpdate() method when it is invoked to execute the PreparedStatement object pStmt. This is because pStmt is already assigned the SQL statement to be executed.

9.5.8 Handling Exceptions in JDBC applications

While working with database applications and JDBC API, occasionally there may be situations that can cause exceptions. Commonly occurring exceptions during database handling with JDBC are:

→ ClassNotFoundException

While loading a driver using `Class.forName()`, if the class for the specified driver is not present in the package, then the method invocation throws a `ClassNotFoundException`.

Hence, the method should be wrapped inside a `try` block, followed by a `catch` block which deals with the thrown exception (if any).

Code Snippet 9 shows use of the `ClassNotFoundException`.

Code Snippet 9:

```
try {
    Class.forName("SpecifiedDriverClassName");
} catch (java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: ");
    System.err.println(e.getMessage());
}
```

→ SQLException

Every method defined by the JDBC objects such as `Connection`, `Statement`, and `ResultSet` can throw `java.sql.SQLException`. Hence, whenever, these methods are used, they should be wrapped inside a `try/catch` block to handle the exceptions. The vendor-specific error code is inserted inside the `SQLException` object, which is retrieved using the `getErrorCode()` method.

Code Snippet 10 shows use of SQLException.

Code Snippet 10:

```
try {
    // Code that could generate an exception goes here.

    // If an exception is generated, the catch block below will print out
    // information about it.

}

catch (SQLException ex)
{
    System.err.println("SQLException: " + ex.getMessage());
    System.out.println("ErrorCode: " + ex.getErrorCode());
}
```

9.5.9 Need for Processing Queries

Once the database query is executed and `ResultSet` object is created, the next step is to process and retrieve the result from the `ResultSet`. As the data in the `ResultSet` is in a tabular format and the cursor is positioned before the first row, it needs to traverse the rows using the `next()` method. The `next()` method allows to traverse forward by moving the cursor one row forward. It returns a boolean true if the current cursor position is on a valid row and returns a false when the cursor is placed at a position after the last row.

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

Code Snippet 11:

```
ResultSet rs1=st1.executeQuery("SELECT Employee_Name FROM Employees");
while (rs1.next()) {
    String name=rs1.getString("Employee_Name");
    System.out.println(name);
}
```

Here, the code retrieves the employee name from the `Employee` table and displays them in a sequence using the `next()` method.

9.5.10 Methods to Process Queries

The getter methods are used to extract the data from the current row of the `ResultSet` object and store it into a Java variable of corresponding data type. These methods are declared by the `ResultSet` interface for retrieving the column values using either the index number or name of the column.

If in case there are more than one columns having the same name, then the value of the first matching column will be returned. There are several getter methods to retrieve the different Java type values. They are:

→ **getString()**

The `getString()` method is used for retrieving a string value (SQL type VARCHAR) and assigning into Java String object.

→ **getInt()**

The `getInt()` method is used for retrieving an integer value from the current row and assigning into a Java integer variable.

→ **getFloat()**

The `getFloat()` method is used for retrieving a float value from a current row and assigning into a Java float variable.

→ **getObject()**

The `getObject()` method is used for retrieving the value from a current row as an `Object` and assigning to Object in Java programming language.

Code Snippet 12 shows use of some of the query processing methods.

Code Snippet 12:

```
while (rs1.next()) {
    String name=rs1.getString("Employee_Name");
    int deptId=rs1.getInt("Department_Id");
    float rating=rs1.getFloat("Rating");
    System.out.println(name + " " + deptId+" "+rating);
}
```

The name, department, and rating values are extracted from the respective columns and stored in variables of the data types of Java programming language. The values in the variables are finally displayed.

9.5.11 Closing the Database Connection

The last but important step in a database application is to close both the connection and any open statements, once the processing is complete. The open connection can trigger security troubles. A simple `close()` method is provided by the `Connection`, `Statement`, and `ResultSet` objects to achieve the purpose. To keep the database-releasing methods tidy, and always released (even though an exception may be thrown), database connections should be closed within a `finally` block.

Syntax:

```
public void close()
```

The method releases the JDBC resources and the database connection immediately.

Code Snippet 13 shows use of the `close()` method.

Code Snippet 13:

```
st1.close();
cn1.close();
```

The Statement and the Connection object have been closed using the `close()` method.

9.6 Database Meta Information

The dictionary meaning of metadata is data about data. In the context of databases it can also be defined as information that defines the structure and properties of the data stored in a database. JDBC support the access of metadata by providing several methods. For example, a table in a database has its defined name, column names, data types for its columns and the owner for the table, this description is known as the metadata.

9.6.1 DatabaseMetaData Interface

The information that describes about the database itself is known as the database meta data. This meta data information is stored by the objects of `DatabaseMetaData` interface in the `java.sql` package.

To create a `DatabaseMetaData` object, a new instance of the `DatabaseMetaData` interface is created. This newly created object is assigned with the results fetched by the `getMetaData()` method, called by the `Connection` object.

Code Snippet 14 demonstrates the `getMetaData()` method.

Code Snippet 14:

```
DatabaseMetaData dmd = cn.getMetaData();
```

When a Java application gets a valid connection, this code creates a metadata object.

9.6.2 DatabaseMetaData Methods

The `DatabaseMetaData` interface also exposes a library of methods to display the database related required information. Based upon the type of information that can be retrieved by the methods. Some of them are listed in table 9.5.

Category	Method Name	Description
Escape Characters	<code>getSearchStringEscape()</code>	Retrieves and returns the string to escape wildcard characters.

Category	Method Name	Description
Database Information	getDatabaseProductName() getDatabaseProductVersion() isReadOnly()	Retrieves the name of the connected database product and returns the same as a string. Retrieves the version number of the connected database product and returns the same in a string format. Returns a boolean value of true if the database is in read-only form, else returns a false.
Driver Information	getDriverName() getDriverVersion() getDriverMajorVersion()	Retrieves the name of the currently used JDBC driver and returns the same as a string. Retrieves the version number of the currently used JDBC driver and returns the same as a string. Retrieves the major version number of the currently used JDBC driver and returns the same as an integer.

Table 9.5: Categorizing the Methods of DatabaseMetaData Interface

Some of the other methods are listed in table 9.6.

Method	Description
getURL()	This method returns the URL for this DBMS as a String.
getUserName()	This method returns the user name as known to the database in the form of a String object.
getConnection()	This method returns the connection object that produced this metadata object.
supportsPositionedDelete()	This method returns a boolean value of true or false depending on whether the database supports positioned DELETE statements.
supportsPositionedUpdate()	This method returns a boolean value of true or false depending on whether the database supports positioned UPDATE statements.
supportsTransactions()	This method returns whether the database supports transactions and returns a boolean value of true if transaction is supported, otherwise returns false.

Table 9.6: Methods of DatabaseMetaData Interface

9.6.4 ResultSetMetaData Interface

The information that describes about the data contained in a `ResultSet` is known as the `ResultSet` metadata. This metadata information is stored by the objects of `ResultSetMetaData` interface in the `java.sql` package. This object can give the information about attributes like column names, number of columns and data types for the columns in the result set.

To retrieve all these information, a `ResultSetMetaData` object is created and assigned with the results fetched by the `getMetaData()` method, called by the `ResultSet` object.

Code Snippet 15 shows the creation of a `ResultSetMetaData` object.

Code Snippet 15:

```
ResultSetMetaData rmd = rs.getMetaData();
```

When a Java application gets a valid connection and successfully retrieves the data from the database to a result set, this code creates a `ResultSetMetaData` object.

9.6.5 ResultSetMetaData Methods

The `ResultSetMetaData` interface offers a number of methods to get the information about the rows and columns in the `ResultSet`. These methods can only be called by using the `ResultSet` object rather than using the `Connection` object. Some of the widely used methods provided by the `ResultSetMetaData` interface are listed as follows:

- ➔ `getColumnName()`

The method retrieves the name of the specified column and returns the same as a `String`.

Syntax:

```
public String getColumnName(int column) throws SQLException
```

where,

`String` is the return type of the method containing the column name.

`column` is the column number it takes in integer format, like 1 for the first column, 2 for the second and so on. The method throws an SQL exception in case, any database access error occurs.

Code Snippet 16 shows use of the `getColumnName()` method.

Code Snippet 16:

```
String colName = rmd1.getColumnName(2);
```

Here, the method will retrieve the name of the second column in the resultset and store the retrieved column name value in the variable `colName`.

→ `getColumnCount()`

The method retrieves and returns the number of columns as an integer in the current `ResultSet` object.

Syntax:

```
public int getColumnCount() throws SQLException
```

where,

`int` is the return type of the method returning the number of columns. This method also throws an SQL exception in case, any database access error occurs.

Code Snippet 17 shows use of the `getColumnCount()`.

Code Snippet 17:

```
int totalCols = rmd1.getColumnCount();
```

Here, the method will retrieve the number of columns present in the resultset and store the integer value in the variable `totalCols`.

→ `getColumnType()`

The method retrieves the SQL type of the specified column and returns the same as a `String`.

Syntax:

```
public int getColumnType(int column) throws SQLException
```

where,

`int` is the return type of the method returning the SQL type of the specified columns,

`column` is the column number it takes in integer format, like 1 for the first column, 2 for the second and so on. The method throws an SQL exception in case, any database access error occurs.

Code Snippet 18 shows use of the `getColumnType()` method.

Code Snippet 18:

```
String colType = rmd1.getColumnType(1);
```

Here, the method will retrieve the SQL type of the first column in the resultset and store the string value in the variable `colName`.

9.6.6 Connecting to Database Using Type 4 Driver

SQL Server 2012 is a widely used database from Microsoft. It allows the user to manipulate data with a large number of data types, stored procedures, and offers a secure platform for developing enterprise database applications. Java applications can be connected to SQL Server 2012 through Type 4 JDBC Driver.

Syntax:

```
jdbc:sqlserver://serverName;instanceName:portNumber;property=value[;property  
=value]
```

where,

`jdbc:sqlserver://` is a required string that stands for the sub-protocol and is constant.

`serverName` and `instanceName` are optional and identifies the address of the server and the instance on the server to connect. The `serverName` can be a localhost or IP address of the local computer.

`portNumber` specifies the port to connect to on the `serverName`. If not specified, the default is 1433.

`property` identifies the login information such as username and password.

Code Snippet 19 shows how to connect to SQL Server database.

Code Snippet 19:

```
jdbc:sqlserver://localhost;user=Author;password=*****;
```

The line of code connects the default database on the local computer by specifying a user name and password.

9.7 Check Your Progress

1. Which of the following statements about JDBC are true?

A.	The implementation of the software API for database connectivity varies as per the programming languages, underlying database systems, and operating systems.		
B.	JDBC is the software API for database connectivity, which provides a set of classes and interfaces written in machine level languages to access any kind of database.		
C.	JDBC is independent of the underlying database management software.		
D.	In a two-tier model, the JDBC API is used to translate and send the client request to a local or remote database on the network.		
E.	In a three-tier model, a “middle tier” of services, a third server is employed to send the client request to the database server.		
(A)	A, C, E	(C)	C, D, E
(B)	B, C, D	(D)	A, B, D

2. Match the properties of drivers with the corresponding descriptions.

Description		Driver	
a.	Converts the JDBC calls into a network protocol which is again translated into database-specific calls by a middle tier.	1.	Type-1
b.	Interprets JDBC calls to the database-specific native call interface.	2.	Type-3
c.	Core of JDBC API and converts the client request to a database-understandable, native format.	3.	Type-4
d.	Uses a bridging technology that provides JDBC access via ODBC drivers.	4.	Database drivers
e.	Communicates directly with the database engine using Java sockets, without a middleware or native library.	5.	Type-2
(A)	a-2, b-4, c-5, d-1, e-3	(C)	a-2, b-1, c-5, d-1, e-4
(B)	a-2, b-5, c-1, d-3, e-4	(D)	a-2, b-5, c-4, d-1, e-3

3. Choose the correct code to create a JDBC connection.

(A)	<pre> Class.forName("jdbc.odbc.JdbcOdbcDriver"); Connection cn1 = DriverManager.getConnection("jdbc:odbc: demo", "sa", "playware"); try { Statement st1 = cn.createStatement(); ResultSet rs1 = st. executeQuery("SELECT * FROM Department"); while (rs1.next()) { String name = rs1.getString("Employee_Name"); } } catch(java.lang.ClassNotFoundException e) { } catch(SQLException ce){System.out.println(ce);} </pre>
(B)	<pre> try {Class.forName("jdbc.odbc.JdbcOdbcDriver"); Connection cn1 = DriverManager.getConnection("jdbc:odbc: demo", "sa", "playware"); ResultSet rs1 = st. executeQuery("SELECT * FROM Department"); Statement st1 = cn.createStatement(); while (rs1.next()) { String name = rs1.getString("Employee_Name"); } } catch(java.lang.ClassNotFoundException e) { } catch(SQLException ce){System.out.println(ce);} </pre>

(C)	<pre> try { Class.forName("jdbc.odbc.JdbcOdbcDriver"); Connection cn1 = DriverManager.getConnection("jdbc:odbc:demo", "sa", "playware"); Statement st1 = cn1.createStatement(); ResultSet rs1 = st1.executeQuery("SELECT * FROM Department"); while (rs1.next()) { String name = rs1.getString("Employee_Name"); } } catch(java.lang.ClassNotFoundException e) { } catch(SQLException ce) {System.out.println(ce);} </pre>
(D)	<pre> try { Connection cn1 = DriverManager.getConnection("jdbc:odbc:demo", "sa", "playware"); Class.forName("jdbc.odbc.JdbcOdbcDriver"); Statement st1 = cn1.createStatement(); ResultSet rs1 = st1.executeQuery("SELECT * FROM Department"); while (rs1.next()) { String name = rs1.getString("Employee_Name"); } } catch(java.lang.ClassNotFoundException e) { } catch(SQLException ce) {System.out.println(ce);} </pre>

4. Identify the correct syntax of connecting to a SQL Server database using Type 4 driver.

A.	jdbc:sql://serverName;instanceName:portNumber;property=value[;property=value]
B.	jdbc:sqlserver://serverName;instanceName:portNumber;property=value[;property=value]
C.	jdbc:sqlserver://instanceName:portNumber;property=value[;property=value]
D.	jdbc:sqlserver://serverName;instanceName;property=value[;property=value]

9.7.1 Answers

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

Summary

- JDBC is a software API to access database connectivity for Java applications. This API provides a collection of application libraries and database drivers, whose implementation is independent of programming languages, database systems, and operating systems.
- JDBC offers four different types of drivers to connect to the databases. Each driver has its own advantages and disadvantages. The drivers are chosen based on the client/server architecture and the requirements.
- The java.sql package offers classes that sets up a connection with databases, sends the SQL queries to the databases and retrieving the computed results. The JDBC Application development starts with the loading and registering the JDBC driver followed by creating a connection object and establishing the connection.
- The Statement object is used to send and execute the SQL queries and the ResultSet object is used to retrieve the computed data rows with the help of methods.
- SQL Server 2012 is a popular database from Microsoft. To develop a database application using SQL Server 2012 as the DBMS, JDBC type 4 driver can be used to establish a connection. To execute parameterized runtime queries, the PreparedStatement object is used.
- The information that describes the data in database is known as the metadata. The java.sql package mainly defines the DatabaseMetaData interfaces to access metadata for the database itself and the ResultSetMetaData interface to access the different columns in a result set.

Session 10

Advanced JDBC Features

Welcome to the Session, **Advanced JDBC Features**.

This session describes how to work with scrollable result sets using JDBC. The session provides explanations on stored procedure, batch updates, and transactions along with the various ways of implementing them. Finally, the session introduces you to JDBC 4.0 and 4.1 features, such as RowSet and its various types. The session explains RowSetProvider, RowSetFactory, and RowSet Interfaces. Finally, the session discusses how to use RowSet, JdbcRowSet, and CachedRowSet objects for transferring data to and from databases.

In this Session, you will learn to:

- List and describe scrollable result sets
- List different types of ResultSet and row-positioning methods
- Explain stored procedures
- Explain how to call a stored procedure using JDBC API
- Describe the steps to update records
- Explain the steps of implementing transactions using JDBC
- List the enhancements of JDBC 4.0 API
- Explain RowSet and its type
- Describe JDBC 4.1 RowSetProvider and RowSetFactory Interfaces



10.1 Introduction

The `ResultSet` object in JDBC API represents a SQL result set in the JDBC application. An SQL result set is the result table returned by the database in query execution. It includes a set of rows from a database as well as meta-information about the query such as the column names, and the types and sizes of each column.

A result set in the JDBC API can be thought as a table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A default result set object cannot be updated or scrolled backward and forward. By default the cursor moves forward only. So, it can be iterated only once and only from the first row to the last row. In addition to moving forward, one row at a time, through a `ResultSet`, the JDBC Driver also provides the capability to move backward or go directly to a specific row. Additionally, updating and deleting rows of result is also possible. Even, the `ResultSet` can be kept open after a `COMMIT` statement.

The characteristics of `ResultSet` are as follows:

→ **Scorable**

It refers to the ability to move backward as well as forward through a result set.

→ **Updatable**

It refers to the ability to update data in a result set and then copy the changes to the database. This includes inserting new rows into the result set or deleting existing rows.

→ **Holdable**

It refers to the ability to [check whether the cursor stays open after a COMMIT](#).

10.1.1 Scrollable ResultSet

A scrollable result set allows the cursor to be moved to any row in the result set. This capability is useful for GUI tools for browsing result sets. Since scrollable result sets involve overhead, they should be used only when the application needs scrolling.

You can create a scrollable `ResultSet` through methods of the `Connection` interface.

Table 10.1 lists the methods that can be invoked on the connection instance for returning a scrollable `ResultSet`.

Method	Syntax	Code
<code>createStatement()</code>	<code>public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLException</code>	<code>Statement st = cn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,</code>

Method	Syntax	Code
	<p>where,</p> <p>resultSetType: It is this argument that will help to create a scrollable ResultSet. It represents the constant values that can be ResultSet.TYPE_FORWARD, ResultSet.TYPE_SCROLL_SENSITIVE, or ResultSet.TYPE_SCROLL_INSENSITIVE.</p> <p>resultSetConcurrency: Represents one of the two ResultSet constants for specifying whether a result set is read-only or updatable. The constant values for concurrency type can be ResultSet.CONCUR_READ_ONLY or ResultSet.CONCUR_UPDATABLE.</p>	<pre>ResultSet.CONCUR_READ_ONLY); ResultSet rs = st.executeQuery("SELECT EMP_NAME, DEPT FROM EMPLOYEES");</pre>
prepareCall()	<p>public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency) throws SQLException</p> <p>where,</p> <p>sql represents a String object containing the SQL statements to be sent to the database.</p> <p>resultSetType has the same attributes as in the createStatement method.</p> <p>resultSetConcurrency represents one of the two ResultSet constants for specifying whether a result set is read-only or updatable.</p>	<pre>CallableStatement cs = cn.prepareCall("? = CALL EMPLOYEE(?, ?, ?)", ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);</pre>

Table 10.1: Methods Returning Scrollable ResultSet

10.1.2 Types of ResultSet Values

The `ResultSet` interface in Java provides access to a table of data that represents a database result set. A `ResultSet` object is usually generated by executing a statement that queries the database. The different static constant values that can be specified for the result set type are as follows:

- ➔ `TYPE_FORWARD_ONLY`

A cursor that can only be used to process from the beginning of a `ResultSet` to the end of it. The cursor only moves forward. This is the default type. You cannot scroll through this type of result set nor can it be positioned, and lastly, it is not sensitive to the changes made to the database.

Code Snippet 1 demonstrates the `TYPE_FORWARD_ONLY` cursor.

Code Snippet 1:

```
PreparedStatement pst = cn.prepareStatement  
("SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",  
ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);  
pst.setString(1, "28959");  
ResultSet rs = pst.executeQuery();
```

The field's `EMP_NO` and `SALARY` is insensitive to changes made to the database while it is open.

- ➔ `TYPE_SCROLL_INSENSITIVE`

A cursor that can be used to scroll in various ways through a `ResultSet`. This type of cursor is insensitive to changes made to the database while it is open. It contains rows that satisfy the query when the query was processed or when data is fetched.

Code Snippet 2 demonstrates the `TYPE_SCROLL_INSENSITIVE` cursor.

Code Snippet 2:

```
PreparedStatement pst = cn.prepareStatement  
("SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",  
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);  
pst.setString(1, "28959");  
ResultSet rs = pst.executeQuery();
```

Here, too, the field's `EMP_NO` and `SALARY` is insensitive to changes made to the database while it is open.

- ➔ `TYPE_SCROLL_SENSITIVE`

A cursor that can be used to scroll in various ways through a `ResultSet`. This type of cursor is sensitive to changes made to the database while it is open. Changes to the database have a direct

impact on the `ResultSet` data.

Code Snippet 3 shows the use of `TYPE_SCROLL_SENSITIVE` cursor.

Code Snippet 3:

```
PreparedStatement pst = cn.prepareStatement
("SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",
ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
pst.setString(1, "28959");
ResultSet rs = pst.executeQuery();
```

The field's `EMP_NO` and `SALARY` will have a direct impact on the `ResultSet` data.

10.1.3 Row Positioning Methods

By default, `ResultSet` always allows forward movement only, meaning that the only valid cursor-positioning method to call is `next()`. You have to explicitly request for a scrollable `ResultSet`. Table 10.2 describes the cursor-positioning methods of `ResultSet`.

Method	Description
<code>next()</code>	This method moves the cursor forward one row in the <code>ResultSet</code> from the current position. The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.
<code>previous()</code>	The method moves the cursor backward one row in the <code>ResultSet</code> . The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.
<code>first()</code>	The method moves the cursor to the first row in the <code>ResultSet</code> . The method returns <code>true</code> if the cursor is positioned on the first row and <code>false</code> if the <code>ResultSet</code> is empty.
<code>last()</code>	The method moves the cursor to the last row in the <code>ResultSet</code> . The method returns <code>true</code> if the cursor is positioned on the last row and <code>false</code> if the <code>ResultSet</code> is empty.
<code>beforeFirst()</code>	The method moves the cursor immediately before the first row in the <code>ResultSet</code> . There is no return value from this method.
<code>afterLast()</code>	The method moves the cursor immediately after the last row in the <code>ResultSet</code> . There is no return value from this method.

Method	Description
<code>relative (int rows)</code> <i>lay gia tri current row</i>	The method moves the cursor relative to its current position. If row value is 0, this method has no effect. If row value is positive, the cursor is moved forward that many rows. If there are fewer rows between the current position and the end of the <code>ResultSet</code> than specified by the input parameters, this method operates like <code>afterLast()</code> method. If row value is negative, the cursor is moved backward that many rows. The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.
<code>absolute (int row)</code> <i>lay gia tri row dau tien</i>	The method moves the cursor to the row specified by row value. If row value is positive, the cursor is positioned that many rows from the beginning of the <code>ResultSet</code> . The first row is numbered 1, the second is 2, and so on. If row value is negative, the cursor is positioned that many rows from the end of the <code>ResultSet</code> . The last row is numbered -1, the second to last is -2, and so on. If row value is 0, this method operates like <code>beforeFirst</code> method. The method returns <code>true</code> if the cursor is positioned on a valid row and <code>false</code> otherwise.

Table 10.2: Cursor-positioning Methods of `ResultSet`

Note - Calling `absolute(1)` is equivalent to calling `first()` and calling `absolute(-1)` is equivalent to calling `last()`.

10.1.4 Updatable `ResultSet`

Updatable `ResultSet` is the ability to update rows in a result set using Java programming language methods rather than SQL commands.

An updatable `ResultSet` will allow the programmer to change the data in the existing row, to insert a new row, or delete an existing row. The `newUpdateXXX()` methods of `ResultSet` interface can be used to change the data in an existing row.

When using an updatable result set, it is recommended to make it scrollable. This allows to position to any row that is to be changed.

10.1.5 Concurrency in ResultSet

Updatability means to update the data in a result set and then incorporate or copy the changes to the database such as inserting new rows into the result set or deleting existing rows.

Updatability might also require database write locks to provide access to the underlying database. Because you cannot have multiple write locks concurrently, updatability in a result set is associated with concurrency in database access. **Concurrency is a process wherein two events take place in parallel.**

The concurrency type of a result set determines whether it is updatable or not. The constant values that can be assigned for specifying the concurrency types are as follows:

- CONCURRENCY.READ_ONLY

The result set cannot be modified and hence, it is not updatable in any way.

- CONCURRENCY.UPDATABLE

The update, insert, and delete operations can be performed on the result set and the changes are copied to the database.

10.1.6 Updating a Row

Rows may be updated in a database table by using an object of the `ResultSet` interface.

There are two steps involved in this process. The first step is to change the values for a specific row using various `update<Type>` methods, where `<Type>` is a Java data type. These `update<Type>` methods correspond to the `get<Type>` methods available for retrieving values. The second step is to apply the changes to the rows of the underlying database.

The database itself is not updated until the second step. Updating columns in a `ResultSet` without calling the `updateRow()` method does not make any changes to the database. Once the `updateRow()` method is called, changes to the database are final and cannot be undone.

Consider a scenario where you want to update the `EMPLOYEE_ID` field of the first row retrieved in the result set.

This operation could be accomplished in a number of steps.

- **Step 1: Positioning the Cursor**

Code Snippet 4 shows how to position the cursor.

Code Snippet 4:

```
// Create an updatable result set
Statement st = cn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

// Retrieve the scrollable and updatable result set
ResultSet rs = st.executeQuery("SELECT NAME, EMPLOYEE_ID FROM
EMPLOYEES");

// Move to the first row in the result set
rs.first();
```

The `first()` method moves the cursor to the first row in the result set. After invoking this method, any call to update methods affects the values of first row until the cursor is moved to another row.

Similarly, there are other methods such as `last()`, `next()`, `previous()`, `beforeFirst()`, `afterLast()`, `absolute(int)`, and `relative(int)` that can be used to move the cursor to the required location in the result set.

→ Step 2: Updating the Columns

After using the `first()` method to navigate to the first row of the result set, call `updateInt()` to change the value of the `EMPLOYEE_ID` column in the result set.

Code Snippet 5 shows how to update the columns.

Code Snippet 5:

```
// Update the second column in the result set
rs.updateInt(2, 34523);
```

The `updateInt()` method is used because the column to be updated has integer values. The `updateInt()` method has two parameters. The first parameter specifies the column number that is to be updated. The second parameter specifies the new value that will replace with the existing column value for that particular record.

Similarly, there are other update methods such as `updateString()`, `updateFloat()`, and `updateBoolean()` that can be used to update a particular column consisting of a string, float, and boolean value respectively.

→ Step 3: Committing the Update

After making the change, call `updateRow()` method of the `ResultSet` interface to actually reflect the change in the underlying database as shown in Code Snippet 6.

Code Snippet 6:

```
// Committing the row updation
rs.updateRow();
```

If the `updateRow()` method is not called before moving to another row in the result set, any changes made will be lost. To make a number of changes in a single row, make multiple calls to `updateXXX()` methods and then a single call to `updateRow()`. Be sure to call `updateRow()` before moving on to another row.

10.1.7 Steps for Inserting a Row

The procedure for inserting a row is similar to that of updating data in an existing row, with a few differences. To insert a row, you first position the cursor, next update one or more columns and then finally, commit the changes.

→ Step 1: Positioning the Cursor

An updatable result supports a row called the insert row. It is a buffer for holding the values of a new row. The first step is to move to the insert row, using the `moveToInsertRow()` method.

Code Snippet 7 shows the use of `moveToInsertRow()` method.

Code Snippet 7:

```
// Create an updatable result set
ResultSet rs = stmt.executeQuery("SELECT NAME, EMPLOYEE_ID FROM
EMPLOYEES");
// Move cursor to the "insert row"
rs.moveToInsertRow();
```

Here, the insert row is an empty row containing all the fields but no data and is associated with the `ResultSet` object. It can be thought of as a temporary row in which you can compose a new row.

→ Step 2: Updating the Columns

After moving to the insert row, use `updateXXX()` methods to load new data into the insert row.

Code Snippet 8 shows the code to update the columns.

Code Snippet 8:

```
// Set values for the new row
rs.updateString(1, "William Ferris");
rs.updateInt(2, 35244);
```

The first line of code inserts the Employee Name as “William Ferris” in the first column using the `updateString()` method. The second line of code inserts the `Employee_ID` as “35244” in the second column using the `updateInt()` method.

→ Step 3: Inserting the Row

After using the `updateXXX()` method, the `insertRow()` method is called to append the new row to the `ResultSet` and the underlying database.

Code Snippet 9 shows the code to insert the row.

Code Snippet 9:

```
// Commit appending of new row to the result set
rs.insertRow();
```

After calling the `insertRow()` method, another new row can be created. Using the various navigation methods, you can also move back to the `ResultSet`. There is one more navigation method named `moveToCurrentRow()` that takes you back to where you were before you called `moveToInsertRow()`. Note, that it can only be called while you are in the insert row.

10.1.8 Steps for Deleting a Row

Deleting a row from an updateable result set is easy. The two steps to delete the row are to position the cursor and delete the row. Deleting a row from the result set has been described in steps.

→ **Step 1: Positioning the Cursor**

The first step is to positioning the cursor by moving the cursor to the desired row that is to be deleted as shown in Code Snippet 10.

Code Snippet 10:

```
// Move the cursor to the last row of the result set
rs.last();
```

Here, the cursor is moved to the last record of the result set that is to be deleted. One can move to the desired row by using any of the various navigation methods of `ResultSet` interface for deleting a particular row.

→ **Step 2: Deleting the Row**

The second step is to delete the row. After moving to the last row of the result set, call the `deleteRow()` method to commit the deletion of the row.

Code Snippet 11 shows the code to delete the row.

Code Snippet 11:

```
// Deleting the row from the record set
rs.deleteRow();
```

Calling the `deleteRow()` method of the `ResultSet` interface also deletes the row from the underlying database.

Some JDBC driver will remove the row and the row will not be visible in the result set. Other JDBC drivers will place a blank row in place of the deleted row. Then the `absolute()` method can be used with the original row positions to move the cursor because the row numbers in the result set are not changed by deletion.

10.2 Prepared Statement and Callable Statement

A stored procedure can be defined as a group of SQL statements performing a particular task. Stored procedures are used to group or batch a set of operations or queries to be executed on a database server. Stored procedures having any combination of input, output, or input/output parameters can be compiled and executed. As stored procedures are pre-compiled, they are faster and more efficient than using individual SQL query statements.

Figure 10.1 displays the stored procedure.

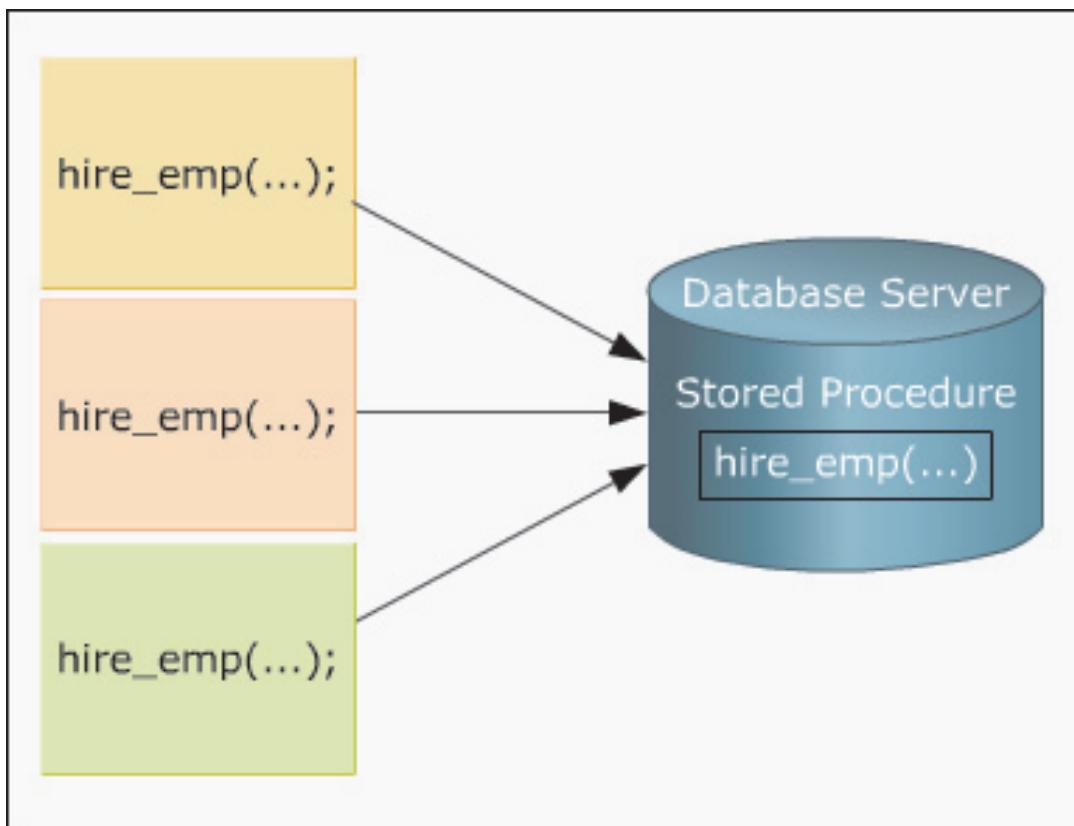


Figure 10.1: Stored Procedure

Stored procedures are supported by database systems such as SQL Server, Oracle, or Sybase.

Note - Consider an employee database having functions such as calculating total salary, looking up for a particular employee or department and calculating salary based on department. These functions could be coded as stored procedures and executed by the application code.

A stored procedure is a named group of SQL statements that has been created earlier and stored in the server database. A stored procedure is like a program that is kept and executed within a database server. Stored procedures accept input parameters so that a single procedure can be used over the network by several clients with different input data. The stored procedure is called from a Java class using a special syntax. When the procedure is called, the name of the procedure and the parameters you specify are sent over the JDBC connection to the DBMS, which executes the procedure and returns the results back over the connection.

Stored procedures reduce database network traffic, improve performance and help ensure the integrity of the database. A stored procedure is written in a metalanguage, defined by the vendor. It is used to group or batch multiple SQL statements that are stored in executable form at the database. Generally, it can be written using PL/SQL or Sybase Transact SQL. These languages are non-portable from one DBMS to another.

The basic elements that a stored procedure consists of are: SQL statements, variables, and one or more parameters. These elements can be used to perform a range of tasks varying from simple SELECT statement to complex business rule validations.

Statements form the main element of the stored procedure. Statements consists of standard SQL statements, control-flow statements such as if, if..else, while, and so on and other SQL Server specific statements that allows to control server settings or create or modify database objects and so on. Variables are like any other variables found in a programming language. Variables store temporary values in the memory for the duration of the stored procedures. Parameters are used to pass and retrieve values from a stored procedure.

A stored procedure can be user-defined, SQL Server system-defined, or extended. User-defined stored procedures are created by users. System stored procedures begin with `sp_` and are stored in the `Master` database. Extended stored procedures are compiled DLLs and are present in the master database.

Stored procedures in SQL Server are similar to procedures in other programming languages. There are lot of advantages in using stored procedures. The benefits that an application has when it uses a stored procedure are: reduced network usage between clients and servers, enhanced hardware and software capabilities, increased security, reduced development cost, increased reliability, centralized security, administration, and maintenance for common routines.

Stored procedures also have many other advantages that are as follows:

- Accepts input parameters and returns values as output parameters to the calling procedure.
- Contains call to other procedures and also contains programming statements that perform operations in the database.
- Indicates success or failure by returning a status value to the calling procedure or batch.
- Compiles each stored procedure once and then reutilizes the output. Thus, there is an improvement in performance when stored procedures are called repeatedly.
- Can be executed independently of underlying table permissions after having been granted permissions.
- Executes on either a local or remote SQL Server. This enables to run processes on other machines and work with information across servers.
- Provides optimum solution between client-side software and SQL Server because an application program written in C or Visual Basic can also execute stored procedures.

- Allows faster execution because the optimized execution path is calculated earlier and saved.
- Separates or abstract server-side functions from the client-side. It is much easier to code a GUI application to call a procedure than to build a query through the GUI code.

The drawback of using stored procedures is portability. For each targeted DBMS, the stored procedure code needs to be written.

10.2.1 Characteristics of Stored Procedures

Stored procedures have the following characteristics:

- They contain SQL statements using constructs and control structures.
- They can be invoked by name in an application that is using SQL.
- They allow an application program to run in two parts such as the application on the client and the stored procedure on the server.

A client application not using stored procedures increases the network traffic whereas an application using stored procedures reduces network traffic. An application using stored procedure also reduces the number of times a database is accessed.

Figure 10.2 displays the stored procedure usage.

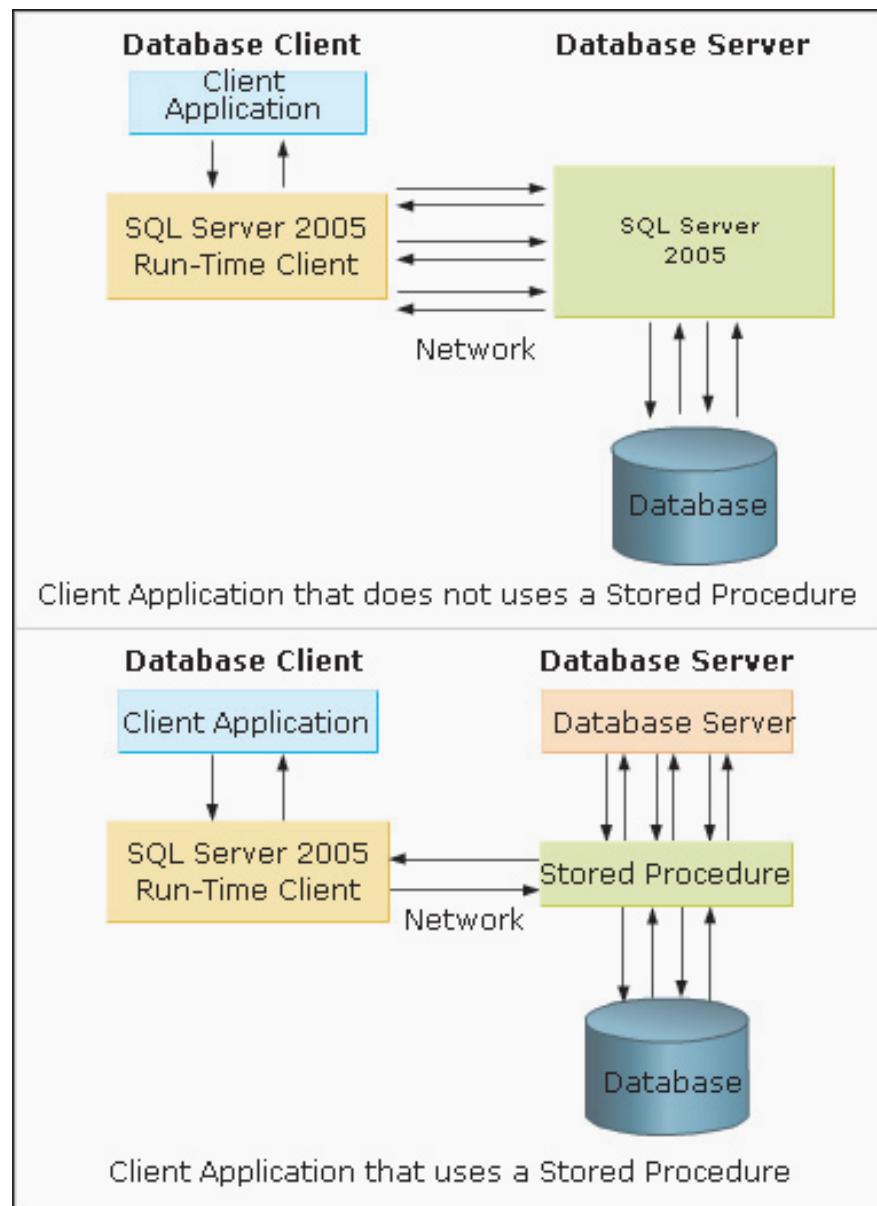


Figure 10.2: Use of Stored Procedures

10.2.2 Benefits of Stored Procedures

The benefits that an application has when it uses a stored procedure are:

→ **Reduced network traffic**

On invoking a stored procedure, the control is passed to the database server by the application using the stored procedure. Intermediate processing of data is performed by the stored procedure and the records required by the client applications are transferred. Thus, network traffic is reduced and performance enhanced.

→ Enhanced hardware and software capabilities

An application using stored procedures accesses the memory, disk space, and the installed software of the database server. Thus, the executable business logic can be distributed across machines having the required hardware and software capabilities.

→ Increased security

The security is increased if the stored procedure is given the database privileges. The Database Administrator (DBA) or the stored procedure developer will have the same privileges as required by the stored procedure. On the other hand, the client application users need not have the same privilege and this will increase the security.

→ Decrease in development cost

In a database application, the tasks that are repeated might return a fixed set of data, or perform the same set of multiple requests to a database. Reusing a common stored procedure will lead to decrease in development cost and address these recurrent situations.

→ Centralized control

Security, administration, and maintenance of common routines become easier if shared logic is located in one place at the server. Client applications can call stored procedures that run SQL queries with little or no additional processing.

Note - The drawback of using a stored procedure is **lack of portability**. For each targeted DBMS, **the stored procedure code may have to be rewritten**.

10.2.3 Creating a Stored Procedure Using Statement Object

Stored procedures can be created using a Statement object. Creating a stored procedure with the Statement object involves two steps.

→ Creating stored procedure and storing it in a String variable

Code Snippet 12 shows the code to declare the string variable containing the definition of a stored procedure.

Code Snippet 12:

```
//String createProcedure = "Create Procedure DISPLAY_PRODUCTS " + "as
" + "select PRODUCTS.PRD_NAME, COFFEES.COF_NAME " + "from PRODUCTS,
COFFEES " + "where PRODUCTS.PRD_ID = COFFEES.PRD_ID " + "order by
PRD_NAME";
```

The code creates a procedure using SQL statement and stores it in a string variable `createProcedure`.

→ Using the Statement object

Code Snippet 13 shows the use of the Statement object.

Code Snippet 13:

```
// An active connection cn is used to create a Statement object
Statement st = cn.createStatement();
// Execute the stored procedure
st.executeUpdate(createProcedure);
```

The Connection object `cn` is used to create a Statement object. The Statement object is used to send the SQL statement creating the stored procedure to the database. The procedure `DISPLAY_PRODUCTS` is compiled and stored in the database as a database object and it can be called. On successful completion of `executeUpdate()` method, a procedure named `DISPLAY_PRODUCTS` is created.

Figure 10.3 displays the stored procedure using Statement object.

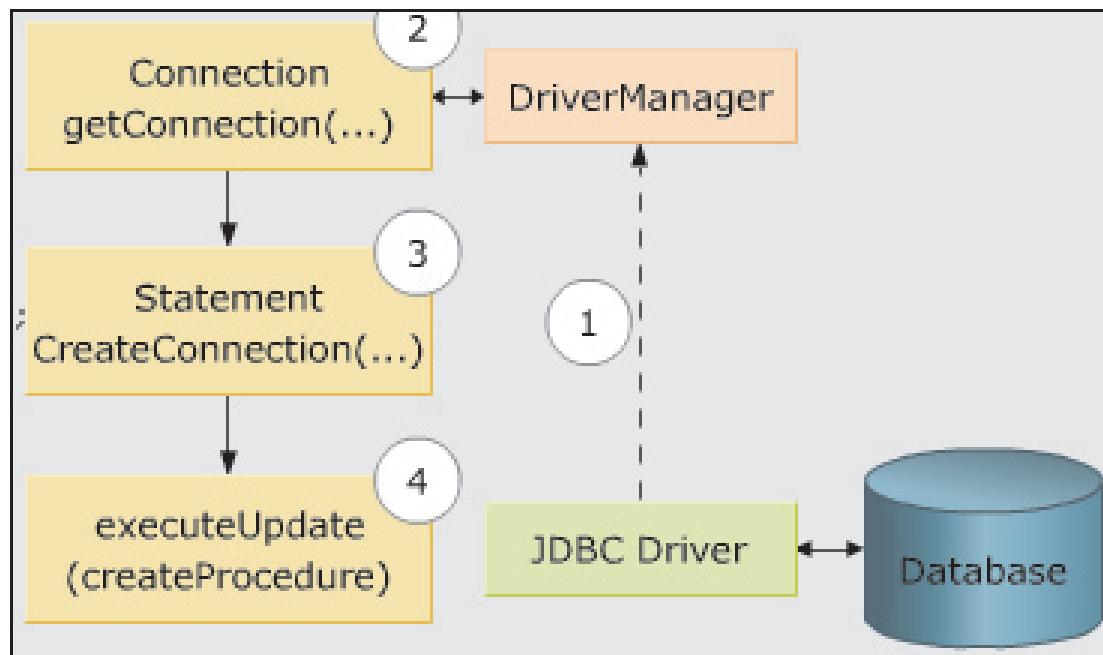


Figure 10.3: Stored Procedure Using Statement Object

10.2.4 Parameters of a Stored Procedure

Stored procedures can accept data in the form of input parameters that are specified at execution time. There can be zero or more parameters per stored procedure.

The different parameters used in stored procedure are as follows:

→ IN

An `IN` parameter is used to pass values into a stored procedure. The value of an `IN` parameter cannot be changed or reassigned within the module and hence is constant.

→ OUT

An `OUT` parameter's value is passed out of the stored procedure module, back to the calling block. A value can be assigned to an `OUT` parameter in the body of a module. The value stored in an `OUT` parameter is a variable and not a constant.

→ IN/OUT

An `IN/OUT` parameter is a parameter that can act as an `IN` or an `OUT` parameter or both. The value of the `IN/OUT` parameter is passed in the stored procedure and a new value can be assigned to the parameter and passed out of the module. An `IN/OUT` parameter behaves like an initialized variable.

Figure 10.4 displays the parameters in stored procedure.

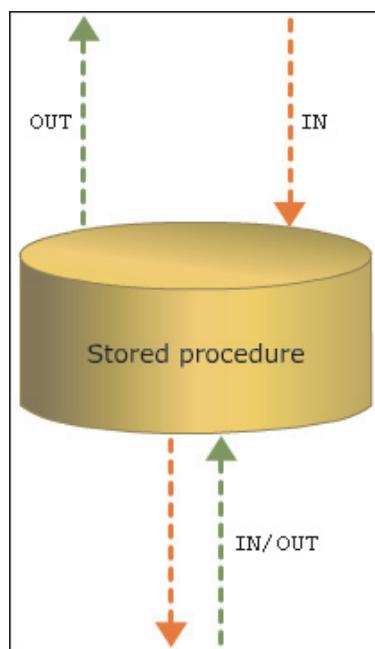


Figure 10.4: Parameters in Stored Procedure

Note - Both, the `IN` and the `IN/OUT` parameters must be variables, and not constants.

10.2.5 Creating a CallableStatement Object

A stored procedure can be called from a Java application with the help of a `CallableStatement` object. A `CallableStatement` object does **not contain** the **stored procedure** itself but **contains only a call to the stored procedure**.

The call to a stored procedure is written in an escape syntax, that is, the call is enclosed within curly braces. The call may take two forms, such as with a result parameter, and without a result parameter.

The result parameter is a value returned by a stored procedure, similar to an `OUT` parameter. Both the forms have a different number of parameters used as input (`IN` parameters), output (`OUT` parameters), or both (`INOUT` parameters). A question mark (?) is used to represent a placeholder for a parameter.

Syntax for calling a stored procedure without parameters is as follows:

Syntax:

```
{call procedure_name}
```

Syntax for calling a stored procedure in JDBC is as follows:

Syntax:

```
{call procedure_name[ (?, ?, ...) ]}
```

Placeholders enclosed in square brackets indicate that they are optional.

Syntax for a procedure that returns a result parameter is as follows:

Syntax:

```
{? = call procedure_name[ (?, ?, ...) ]}
```

The `CallableStatement` inherits methods from the `Statement` and `PreparedStatement` interfaces. All methods that are defined in the `CallableStatement` interface deal with `OUT` parameters. The `getXX()` methods like `getInt()`, `getString()` in a `ResultSet` will retrieve values from a result set whereas in a `CallableStatement`, they will retrieve values from the `OUT` parameters or return values of a stored procedure.

`CallableStatement` objects are created using the `prepareCall()` method of the `Connection` interface. The section enclosed within the curly braces is the escape syntax for stored procedures. The driver converts the escape syntax into native SQL used by the database.

Syntax:

```
CallableStatement cst = cn.prepareCall("{call functionname(?, ?)}");
```

where,

`cst` is the name of the `CallableStatement` object.

`functionname` is the name of function/procedure to be called.

Consider the following statement:

```
CallableStatement cs = cn.prepareCall( "{call sal(?)}");
```

The statement creates an instance of `CallableStatement`. It contains a call to the stored procedure `sal()`, which has a single arguments and no result parameter. The type of the (?) placeholder parameters whether it is `IN` or `OUT` parameter is totally dependent on the way the stored procedure `sal()` has been defined.

→ IN Parameters

The `set<Type>()` methods are used to pass any `IN` parameter values to a `CallableStatement` object. These `set<Type>()` methods are inherited from the `PreparedStatement` object. The type of the value being passed in determines which `set<Type>()` method to use. For example, `setFloat()` is used to pass in a float value, and so on.

→ OUT Parameters

In case the stored procedure returns some values (`OUT` parameters), the JDBC type of each `OUT` parameter must be registered before executing the `CallableStatement` object. The `registerOutParameter()` method is used to register the JDBC type. After the registration is done, the statement has to be executed. The `get<Type>()` methods of `CallableStatement` are used to retrieve the `OUT` parameter value. The `registerOutParameter()` method uses a JDBC type (so that it matches the JDBC type that the database will return), and `get<Type>()` casts this to a Java type.

Some common JDBC types are as follows:

- **Char:** used to represent fixed-length character string.
- **Varchar:** used to represent variable-length string.
- **Bit:** used to represent single bit value that can be zero or one.
- **Integer:** used to represent a 32-bit signed integer value.
- **Double:** used to represent a double-precision floating point number that supports 15 digits of mantissa.
- **Date:** used to represent a date consisting of the day, month, and year.

Code Snippet 14 demonstrates how to use the `registerOutParameter()` method.

Code Snippet 14:

```
. . .
CallableStatement cs = cn.prepareCall("{call getData(?, ?)}");
cs.registerOutParameter(1, java.sql.Types.INTEGER);
cs.registerOutParameter(2, java.sql.Types.DECIMAL, 3);
cs.executeQuery();
int x = cs.getInt(1);
java.math.BigDecimal n = cs.getBigDecimal(2, 3);
. . .
```

The code registers the `OUT` parameters and executes the stored procedure called by `cs`. Then it retrieves the values returned in the `OUT` parameters. The `getInt()` method retrieves an integer from the first `OUT` parameter, and the `getBigDecimal()` method retrieves a `BigDecimal` object (with three digits after the decimal point) from the second `OUT` parameter.

`CallableStatement` does not provide a special mechanism to retrieve large `OUT` values incrementally, which can be done with the `ResultSet` object.

Code Snippet 15 demonstrates how to retrieve an `OUT` parameter returned by a stored procedure.

Code Snippet 15:

```
import java.sql.*;
import java.util.*;
class CallOutProc {
    Connection con;
    String url;
    String serverName;
    String instanceName;
    String databaseName;
    String userName;
    String password;
    String sql;
    CallOutProc() {
        url = "jdbc:sqlserver://";
        serverName = "10.2.1.51";
        instanceName = "martin";
        databaseName = "DeveloperApps";
        userName = "sa";
        password = "playware";
    }
    private String getConnectionUrl() {
        // Constructing the connection string
        return url + serverName + ";instanceName = " +instanceName +" ;Da
        tabaseName = " +databaseName;
    }
}
```

```
private java.sql.Connection getConnection() {  
    try {  
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");  
        // Establishing the connection  
        con = DriverManager.getConnection(getConnectionUrl(), userName,  
password);  
        if(con != null)  
            System.out.println("Connection Successful!");  
    } catch(Exception e) {  
        e.printStackTrace();  
        System.out.println("Error Trace in getConnection(): "  
+ e.getMessage());  
    }  
    return con;  
}  
  
public void display(){  
    try {  
        con = getConnection();  
        CallableStatement cstmt = con.prepareCall("{call recalculatetotal  
(?, ?)}");  
        cstmt.setInt(1,2500);  
        cstmt.registerOutParameter(2, java.sql.Types.INTEGER);  
        cstmt.execute();  
        int maxSalary = cstmt.getInt(2);  
        System.out.println(maxSalary);  
    } catch(SQLException ce) {  
        System.out.println(ce);  
    }  
}
```

```

    }
}

public static void main(String args[]) {
    CallOutProc proObj = new CallOutProc();
    proObj.display();
}
}

```

The CallableStatement makes a call to the stored procedure called `recalculatetotal`. The integer variable ‘`a`’ from the stored procedure is initialized to a value of 2500 by passing an argument through the `setInt()` method. The `OUT` parameter is then registered with JDBC as belonging to data type `java.sql.Types.Integer`. The CallableStatement is then executed and this, in turn, executes the stored procedure `recalculatetotal`. The value of the `OUT` parameter is retrieved by invoking the `getInt()` method. This value is the maximum salary from the `Employee` table and stored in an integer variable `maxSalary` and displayed.

A procedure for recalculating the salary of the highest salary earner is shown in Code Snippet 16. The procedure will accept a value and will store the value in an `OUT` parameter.

Code Snippet 16:

```

create procedure recalculatetotal
    @a int, @inc_a int OUT
as
select @inc_a = max(salary) from Employee
set @inc_a = @a * @inc_a;

```

10.3 Batch Update

Batch update can be defined as a set of **multiple update statements that is submitted to the database for processing as a batch**. In Java, the `Statement`, `PreparedStatement`, and `CallableStatement` objects can be used to submit batch updates.

The benefits of batch updating is that it allows you to request records, bring them to the client, make changes to the records on the client side, and then send the updated record back to the data source at some other time. Submitting multiple updates together, instead of individually, can greatly improve performance. Also, batch updating is used when there is no need to maintain a constant connection to the database.

10.3.1 Batch Update Using Statement Interface

The batch update facility allows a `Statement` object to submit multiple update commands together as a single unit or batch, to the underlying DBMS.

The steps to implement batch update using the `Statement` interface are:

→ **Disable the auto-commit mode**

The auto-commit mode of `Connection` object is set to `false` in order to allow multiple statements to be sent together as a transaction. For handling the errors correctly, **the auto-commit mode should be always disabled before beginning a batch update.**

Code Snippet 17 shows how to disable auto-commit mode.

Code Snippet 17:

```
// turn off auto-commit
cn.setAutoCommit(false);
```

To start sending a batch update to a database, first the auto-commit mode of the connection object is set to `false`. Since the connection's auto-commit mode is disabled, the application is free to decide whether or not to commit the transaction if an error occurs or if some of the commands in the batch fail to execute.

→ **Create a Statement instance**

An instance of `Statement` interface is created by calling the `createStatement()` method on the `Connection` object. Initially, the newly created `Statement` object has no list of commands associated with it.

Code Snippet 18 shows use of the `createStatement` method.

Code Snippet 18:

```
//Create an instance of Statement object
Statement st = cn.createStatement();
```

An instance of `Statement` object `st` is created that has initially an empty list of commands associated with it.

→ **Add SQL commands to the batch**

Commands are added to the list of commands associated with the `Statement` object.

The commands are added to the list with the `addBatch()` method of the `Statement` interface. A `Statement` object has the ability to keep track of a list of commands or batch that can be submitted together for execution.

Code Snippet 19 shows how to add SQL commands to the batch.

Code Snippet 19:

```
// Adding the calling statement batch
st.addBatch("INSERT INTO EMPLOYEES VALUES (1000, 'William John')");
st.addBatch("INSERT INTO EMPLOYEES VALUES (1001, 'Jacky Lawrence')");
st.addBatch("INSERT INTO EMPLOYEES VALUES (1002, 'Valentina Watson')");
```

When a Statement object is created, its associated list of commands is empty. Each of these `st.addBatch()` method adds a command to the calling statement's list of commands. These commands are all `INSERT INTO` statements, each one adding a row consisting of two column values.

→ Execute the batch commands

The `executeBatch()` method is called to submits the list of commands of the Statement object to the underlying DBMS for execution. The command gets executed in the order in which it was added to the batch and returns an update count for each command in the batch, also in order.

Code Snippet 20 shows how to execute the batch commands.

Code Snippet 20:

```
// submit a batch of update commands for execution
int[] updateCounts = st.executeBatch();
```

The commands are executed by DBMS in the order in which they were added to the list of commands. First it will add the row of values for 1000; next it will add the row of values for 1001, and finally 1002. The DBMS will return an update count for each command in the order in which it was executed, provided all the three commands are executed successfully. The integer values indicating how many rows were affected by each command are stored in the array `updateCounts`.

If all the three commands in the batch were executed successfully, `updateCounts` will contain three values, all of which are 1 because an insertion affects one row.

→ Commit the changes in the database

The `commit()` method makes the batch of updates to the table permanently. This method needs to be called explicitly because the auto-commit mode for this connection was disabled earlier.

Code Snippet 21 shows how to commit changes in the database.

Code Snippet 21:

```
// Enabling auto-commit mode
cn.commit();
cn.setAutoCommit(true);
```

The first line of code will commit the changes and makes the batch of updates to the table permanent. The second line of code will automatically enable the auto-commit mode of the Connection interface. The statement will be committed after it is executed, and an application no longer needs to invoke the `commit()` method.

→ Remove the commands from the batch

The `clearBatch()` method empties the current list of SQL commands for the Statement object. This method is specified in the `java.sql.Statement` interface.

Code Snippet 22 shows how to remove commands from a batch.

Code Snippet 22:

```
// Emptying the current list of SQL commands
st.clearBatch();
```

The method `st.clearBatch()` can be called to reset a batch if the application decides not to submit a batch of commands that has been constructed for a statement.

10.3.2 Batch Update Using PreparedStatement Interface

The batch update facility is used with a `PreparedStatement` to associate multiple sets of input parameter values with a single `PreparedStatement` object. The `addBatch()` method of the `Statement` interface is given an SQL update statement as a parameter, and the SQL statement is added to the `Statement` object's list of commands to be executed in the next batch. It is an example of static batch updates.

`PreparedStatement` interface allows creating parameterized batch update. The `setXXX()` methods of the `PreparedStatement` interface are used to create each parameter set, while the `addBatch()` method adds a set of parameters to the current batch. Finally, the `executeBatch()` method of the `PreparedStatement` interface is called to submit the updates to the DBMS, which also clears the statement's associated list of batch elements.

Code Snippet 23 shows how to perform batch updates using `PreparedStatement`.

Code Snippet 23:

```
// Turn off auto-commit
cn.setAutoCommit(false);

// Creating an instance of Prepared Statement
PreparedStatement pst = cn.prepareStatement("INSERT INTO EMPLOYEES
VALUES (?, ?)");
```

```
// Adding the calling statement batches
pst.setInt(1, 5000);
pst.setString(2, "Roger Hoody");
pst.addBatch();
pst.setInt(1, 6000);
pst.setString(2, "Kelvin Keith");
pst.addBatch();
// Submit the batch for execution
int[] updateCounts = pst.executeBatch();
// Enable auto-commit mode
cn.commit();
```

The `pst.executeBatch()` method is called to submit the updates to the DBMS. Calling `pst.executeBatch()` clears the statement's associated list of batch elements. The array returned by `pst.executeBatch()` contains an element for each set of parameters in the batch, similar to the case for Statement interface.

10.3.3 Batch Update Using CallableStatement Interface

The functionality of a `CallableStatement` object and a `PreparedStatement` object is same. The batch update facility on a `CallableStatement` object can call only stored procedures that take input parameters or no parameters at all. Also, the stored procedure must return an update count. The `executeBatch()` method of the `CallableStatement` interface that is inherited from `PreparedStatement` interface will throw a `BatchUpdateException` if the return value of stored procedure is anything other than an update count or takes `OUT` or `IN/OUT` parameters.

Code Snippet 24 shows batch update using `CallableStatement` interface.

Code Snippet 24:

```
// Creating an instance of Callable Statement
CallableStatement cst = cn.prepareCall("{call updateProductDetails(?, ?)}");
// Adding the calling statement batches
cst.setString(1, "Cheese");
cst.setFloat(2, 70.99f);
cst.addBatch();
cst.setString(1, "Almonds");
```

```

cst.setFloat(2, 80.99f);
cst.addBatch();
// Submitting the batch for execution
int [] updateCounts = cst.executeBatch();
// Enabling auto-commit mode
cn.commit();

```

The CallableStatement object, cst, contains a call to the stored procedure named updateProductDetails with two sets of parameters associated with it. When cst is executed, two update statements will be executed together as a batch. The first parameter will have the value as "Cheese" and "70.99f", and a second one as "Almond" and "80.99f". The letter f followed by a number, as in "70.99f", tells the Java compiler that the value is a float.

10.4 Transactions

In a Web-centric world, it is common to have distributed databases, where related data may be stored in different computers located in the same place or computers, which are globally distributed.

Database integrity is an essential feature of relational DBMS. There are times when one does not want one statement or action on a database to be performed unless another one succeeds. Using the example of a database of student scores at an educational institution, when a student record is being deleted from the Student table, it must also be deleted from the Results table; otherwise the data will be inconsistent. To be sure that either both actions takes place or none of the action takes place, transaction is used.

A transaction is a set of one or more statements that are executed together as a unit. This ensures that either all the statements in the set are executed or none of them is executed. Transactions also help to preserve the integrity of the data in a table. To avoid conflicts during a transaction, a DBMS will use locks, which are mechanisms for blocking access by others to the data that is being accessed by the transaction. Once a lock is set, it will remain in force until the transaction is committed or rolled back.

Consider the scenario of a bank funds transfer that requires withdrawing money from one account to deposit into another. If the process of withdrawal is completed but the deposit fails then the customer faces a loss. If the deposit succeeds but the withdrawal fails then the bank faces a loss. Since both the steps form a single transaction, a transaction is often defined as an indivisible unit of work.

10.4.1 Properties of Transaction

The properties Atomicity, Consistency, Isolation, and Durability (ACID) guarantee that database transactions are processed reliably.

→ Atomicity

It refers to the ability of the database management system to guarantee that either all of the tasks of a transaction are performed or none of them are performed.

→ Consistency

It refers to the database being in a legal state in that a transaction cannot break rules such as integrity constraints. For example, if an integrity constraint says that account holder cannot have negative balance, then a transaction failing to adhere to this rule will be aborted.

→ Isolation

It refers to the ability of the database management system (DBMS) to ensure that there are no conflicts between concurrent transactions. For example, if two people are updating the same item, then one person's changes should not be clobbered when the second person saves a different set of changes. The two users should be able to work in isolation.

→ Durability

It refers to the ability of DBMS to recover committed transactions even if the system or storage media fails.

10.4.2 Implementing Transactions Using JDBC

There are four steps to implement transactions using JDBC, such as Start the transaction, Perform transactions, Use Savepoint, and Close or End transaction.

→ Step 1: Start the Transaction

The first step is to start the transaction. When a connection is created, by default, it is in the auto-commit mode. Hence, each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. When you disable auto-commit, the start and end of a transaction is defined which lets you determine whether to commit or rollback the entire transaction.

To disable a connection's auto-commit mode, you invoke the `setAutoCommit()` method, which accepts a single boolean parameter, as shown `cn.setAutoCommit(false);`

To start the transaction, the active connection auto-commit mode is disabled. Once auto-commit mode is disabled, no SQL statements will be committed until the method `commit` is called explicitly. All statements executed after the previous call to the method `commit` will be included in the current transaction and will be committed together as a unit.

→ Step 2: Perform Transactions

The second step is to perform the transaction as shown in Code Snippet 25.

Code Snippet 25:

```
// Creating an instance of Callable Statement
CallableStatement cst = cn.prepareCall("{call updateProductDe-
tails(?, ?)}");

// Adding the calling statement batches
cst.setString(1, "Cheese");
cst.setFloat(2, 70.99f);
cst.addBatch();
cst.setString(1, "Almonds");
updateTotal.setString(2, "Russian");
updateTotal.executeUpdate();
```

As learnt, once auto-commit mode is disabled, no SQL statements are committed until the `commit()` method is explicitly called. Here, the two prepared statements `updateSales` and `updateTotal` will be committed together after the call to `commit()` method.

→ Step 3: Use SavePoint

The third step is to use Savepoint in the transaction as shown in Code Snippet 26.

Code Snippet 26:

```
// Create an instance of Statement object
Statement st = cn.createStatement();

int rows = st.executeUpdate("INSERT INTO EMPLOYEE (NAME) VALUES
(?COMPANY?)");

// Set the Savepoint
Savepoint svpt = cn.setSavepoint("SAVEPOINT_1");

rows = st.executeUpdate("INSERT INTO EMPLOYEE (NAME) VALUES (?FAC-
TORY?)");
```

The code inserts a row into a table, sets the Savepoint `svpt`, and then inserts a second row. When the transaction is later rolled back to `svpt`, the second insertion is undone, but the first insertion remains intact. Thus, when the transaction is committed, only the row containing `?COMPANY?` will be added to `EMPLOYEE`. The method `cn.releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction.

→ **Step 4: Close the Transaction**

The last step is to close or end the transaction. A transaction can end either with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction.

Code Snippet 27 demonstrates how to close the transaction.

Code Snippet 27:

```
. . .
// End the transaction
cn.rollback(svpt);
OR
. . .
cn.commit();
```

10.5 JDBC 4.0 Features

JDBC 4.0 is a major new release in Java SE 6. JDBC 4.0 introduced many new features, with few major changes and enhancements. These changes and enhancements have taken place in DriverManager and ResultSet interfaces, exception handling mechanism, scalar functions, and large binary object types. Besides, a new ROWID data type has been introduced in JDBC 4.0.

Following are the new features:

- **SQLExceptions:** JDBC 4.0 has redefined subclasses of SQLExceptions.
- **Wrapped JDBC Objects:** Application servers use these objects to look for vendor-specific extensions inside standard JDBC objects like Statements and Connections. Note: Derby does not expose any vendor-specific extension.
- **Statement Events:** There are new methods included in connection pool such as javax.sql.PooledConnection, addStatementEventListener, and removeStatementEventListener. Connection pools can use them to listen for Statement closing and error events.
- **DataSources:** There are new implementations of javax.sql.DataSource that help easy development with JDBC 4.0.
- **New Overloads:** There are new overloads of the streaming methods in CallableStatement, PreparedStatement, and ResultSet to define long lengths or omit length arguments.

The methods `setXXX` and `updateXXX` use `java.io.InputStream` and `java.io.Reader` arguments.

- **New Methods in Interfaces:** There are new methods added to the following interfaces:
 - `javax.sql.DatabaseMetaData`
 - `javax.sql.Statement`
 - `javax.sql.Connection`
- **Autoloading of JDBC Drivers:** With JDBC 4.0, the JDBC application **does not have to register drivers programmatically**. There is no need for applications to issue a `Class.forName()` on the driver name, instead `DriverManager` can find appropriate JDBC driver.

10.6 RowSet

`RowSet` is an interface in the new standard extension package `javax.sql.rowset` and is derived from the `ResultSet` interface. It typically contains a set of rows from a source of tabular data like a result set. It can be configured to connect to and read/write data from a JDBC data source. A JDBC `RowSet` object is more easier to use than a result set.

Since the `RowSet` object is a JavaBeans component, it shares some common features of JavaBeans. They are as follows:

→ Properties

A field in a row set is known as the property of a `RowSet`. Every property has appropriate getter and setter methods provided by the interface implementation.

→ JavaBeans Notification Mechanism

A cursor movement, updating, insertion, or deletion of a row or a change to the entire `RowSet` contents generates a notification event. The `RowSet`'s event model notifies all the components that implement the `RowSetListener` interface and are registered as listeners of the `RowSet` object's events.

Figure 10.5 displays the RowSet.

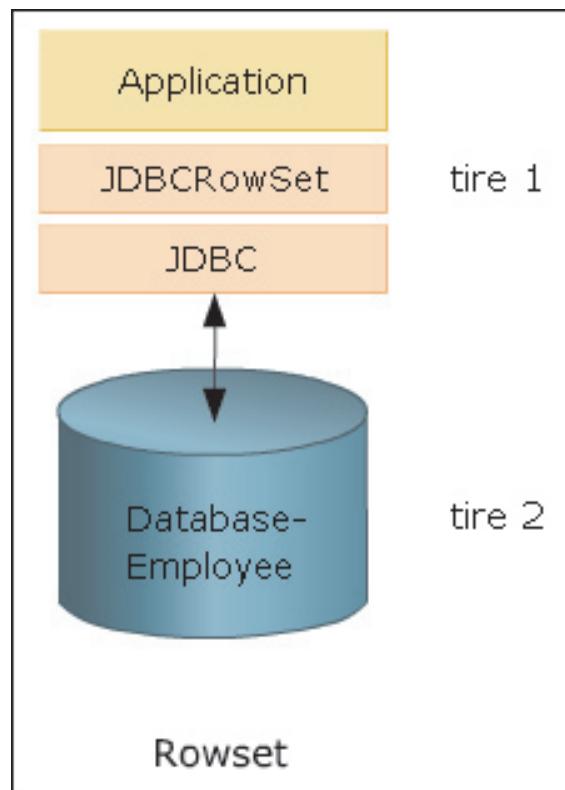


Figure 10.5: RowSet

10.6.1 Benefits of Using RowSet over ResultSet

A RowSet object has some additional advantages as compared to a simple ResultSet object. They are as follows:

- Database Management Systems or the drivers provided by some database vendors do not support result sets that are scrollable and/or updatable. A **RowSet object provides scrollability and updatability for any kind of DBMS or driver.** Scrollability is used in GUI application and for programmatic updating.
- A RowSet object, being a JavaBeans component can be used to notify other registered GUI components of a change.

The main features of using a RowSet are as follows:

- **Scrollability**

The ability to move a result set's cursor to a specific row is known as Scrollability.

→ Updatability

The ability to use Java programming language commands rather than SQL is known as Updatability.

10.6.2 Different Types of RowSets

RowSets are classified depending on the duration of their connection to the database. Therefore, a RowSet can be either connected or disconnected.

A connected RowSet object uses a JDBC driver to connect to a relational database. This connection is maintained throughout the lifespan of the RowSet object.

A disconnected RowSet object connects to a data source only to read data from a ResultSet or write data back to the data source. On completion of the read/write operation, the RowSet object disconnects from the data source.

Disconnected -> read-only

Figure 10.6 displays the connected and disconnected RowSets.

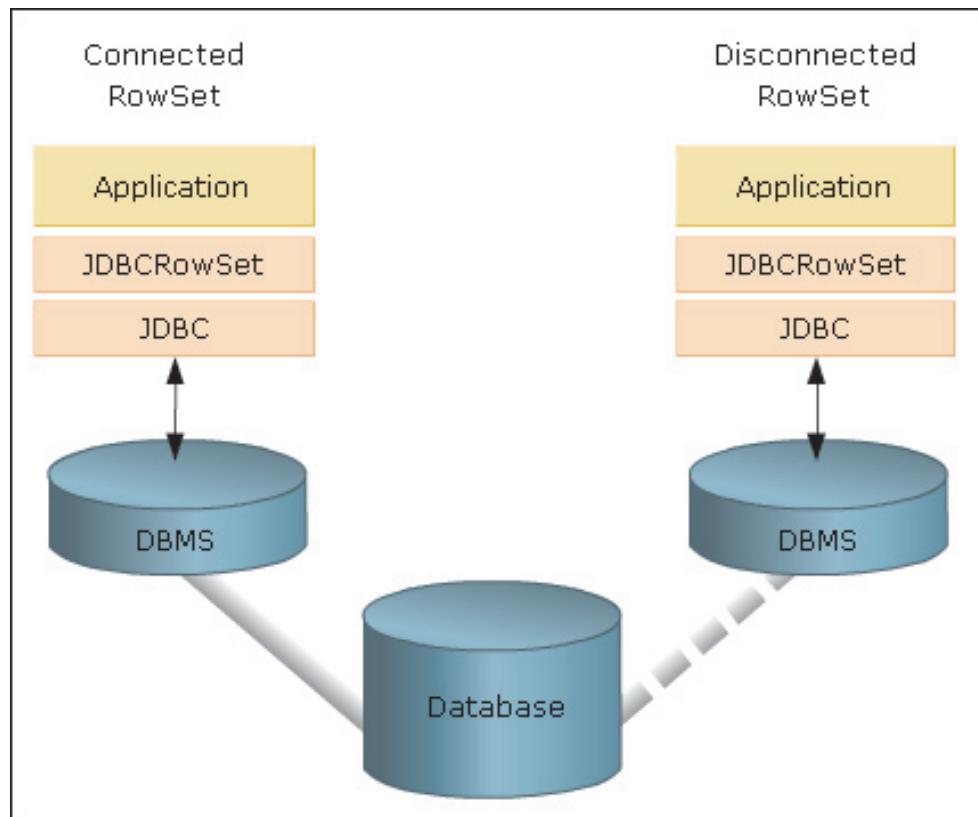


Figure 10.6: Connected and Disconnected RowSets

Some of the interfaces that extend from the RowSet interface are as follows:

- JdbcRowSet
- CachedRowSet
- WebRowSet

- JoinRowSet
- FilteredRowSet

Figure 10.7 shows the RowSet hierarchy.

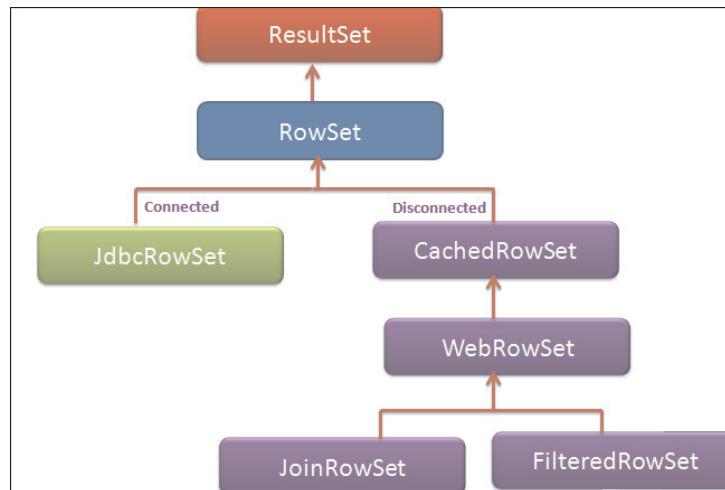


Figure 10.7: RowSet Hierarchy

JDK 7 provides new RowSet 1.1 API. This API includes `javax.sql.rowset.RowSetProvider` and `javax.sql.rowset.RowSetfactory` classes to construct instances of RowSet. The `javax.sql.RowSetProvider` class is used to create a `RowsetFactory` object as shown in the following statement: `RowSetFactory rowsetFactory = RowSetProvider.newFactory();`

The `RowSetFactory` interface includes the following methods to create various RowSet implementations:

- ➔ `createCachedRowSet()`
- ➔ `createFilteredRowSet()`
- ➔ `createJdbcRowSet()`
- ➔ `createJoinRowSet()`
- ➔ `createWebRowSet()`

10.6.3 Implementation of Connected RowSet

The `JdbcRowSet` is the only implementation of a connected RowSet. It is similar to a `ResultSet` object and is often used as a wrapper class to convert a non-scrollable read-only `ResultSet` object into a scrollable, updatable object. To establish a connection and to populate the `RowSet` object, properties like `username`, `password`, `url` of the database, and the `datasourceName` must be set. The `command` property is the query that determines what data the `RowSet` object will hold.

→ JdbcRowSet Interface

`JdbcRowSet` is an interface in the `javax.sql.rowset` package. The `JdbcRowSetImpl` is the reference implementation class of the `JdbcRowSet` interface. If a `JdbcRowSet` object has been constructed using the default constructor then the new instance is not usable until its `execute()` method is invoked. The `execute()` method can only be invoked on such an instance if all the other properties like command, username, password and URL have been set. These properties can be set using the methods of the `RowSet` and the `JdbcRowSet` interface.

The methods are described in table 10.3.

Method	Description
<code>setUserName(String user)</code>	This method sets the RowSet object's username as "user".
<code>setPassword(String pass)</code>	This method sets the RowSet object's database password as "pass".
<code>setURL(String url)</code>	This method sets the RowSet object's URL as "url" that will be used to create a connection.
<code>setCommand(String sql)</code>	This method sets the RowSet object's command property to the given SQL query.
<code>execute()</code> throws <code>SQLException</code>	This method uses the properties of the RowSet object to establish a connection to the database, execute the query set in the command property and read the data from the resulting result set object into the row set object.
<code>commit()</code> throws <code>SQLException</code>	This method makes all changes that have been made since the last commit/rollback call permanent. The invoking of this method releases all database locks currently held by the connection object wrapped by the RowSet object.
<code>rollback()</code> throws <code>SQLException</code>	This method reverses all the changes made in the on-going transaction and releases any database locks held by the connection object contained in the RowSet object.

Table 10.3: Methods of `JdbcRowSet` Interface

→ `JdbcRowSetImpl` Class

The `JdbcRowSetImpl` class is the standard implementation of the `JdbcRowSet` interface. An instance of this class can be obtained by using any of the constructors shown in table 10.4.

Constructor	Description
-------------	-------------

JdbcRowSetImpl()	This constructor creates a default JdbcRowSet object.
JdbcRowSetImpl(Connection con)	This constructor creates a default JdbcRowSet object if the given connection object is valid. The RowSet object serves as a proxy for the result set object created.
JdbcRowSetImpl(String url, String user, String password)	This constructor creates a JdbcRowSet object with the specified url, user name and password.
JdbcRowSetImpl(java.sql.ResultSet res)	This constructor creates a JdbcRowSet object which serves as a thin wrapper around the result set object.

Table 10.4: Constructors of JdbcRowSetImpl Class

All the constructors throw the `java.sql.SQLException` if invalid JDBC driver properties are set or if there is a database access error. A default `JdbcRowSet` object does not show deleted rows in its data. It does not impose any time limit on the time taken by a driver to execute the `RowSet` command. There is no limit for the number of rows such a `RowSet` may contain.

Also, there is no limit as to how many bytes a column of the `RowSet` may contain. The `RowSet` has a scrollable cursor and does not show the changes made by other users to the database. The `RowSet` does not show uncommitted data, hence the problem of dirty reads is eliminated. Each `RowSet` object contains an empty `HashTable` object for storing any parameters that are set.

There are many ways to create a `JdbcRowSet` object that are as follows.

→ Using a ResultSet object

A `ResultSet` object is passed to the `JdbcRowSetImpl` constructor. This creates a `JdbcRowSet` object and populates it with data from the `ResultSet` object.

Code Snippet 28 creates a Statement object to execute a query.

Code Snippet 28:

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("Select * From Employee");
JdbcRowSet jRs = new JdbcRowSetImpl(rs);
```

The `Connection` object, `con` creates a `Statement` object, `st`. The `statement` object then executes the query that produces the `ResultSet` object, `rs` which is passed to the `JdbcRowSetImpl` constructor.

The constructor creates a new `JdbcRowSet` object initialized with the data in `ResultSet` object.

→ Using a default constructor

The default constructor creates an empty `JdbcRowSet` object. The `execute()` method can be invoked once all the properties are set, to establish a connection to the database. This executes the `command` property and as a result the `JdbcRowSet` object is populated with data from the resulting `ResultSet` object.

Code Snippet 29 shows use of the default constructor of `JdbcRowSet`.

Code Snippet 29:

```
JdbcRowSet jrs2 = new JdbcRowSetImpl();
jrs2.setUsername("scott");
jrs2.setPassword("tiger");
jrs2.setUrl("jdbc:protocolName:datasourceName");
jrs2.setCommand("Select * from Employees");
jrs2.execute();
```

→ **Using the RowSetFactory interface**

An instance of the `RowSetFactory` interface can be used to create a `JdbcRowSet` object.

Code Snippet 30 shows how to create the `JdbcRowSet` object.

Code Snippet 30:

```
public void createJdbcRowSetWithRowSetFactory(
    String username, String password) throws SQLException {
    RowSetFactory myRowSetFactory = null;
    JdbcRowSet jdbcRs = null;
    ResultSet rs = null;
    Statement stmt = null;
    try {
        myRowSetFactory = RowSetProvider.newFactory();
        jdbcRs = myRowSetFactory.createJdbcRowSet();
        jdbcRs.setUrl("jdbc:myDriver:myAttribute");
```

```

        jdbcRs.setUsername(username);

        jdbcRs.setPassword(password);

        jdbcRs.setCommand("select * from COFFEES");

        jdbcRs.execute();

        // ...

    }

}

```

The statement `myRowSetFactory = RowSetProvider.newFactory();` creates the `RowSetProvider` object, `myRowSetFactory` with the default `RowSetFactory` implementation, `com.sun.rowset.RowSetFactoryImpl`. If the JDBC driver has its own `RowSetFactory` implementation, use it as an argument of the `newFactory` method.

10.6.4 Using `JdbcRowSet` Object

Some of the most commonly used methods of the `JdbcRowSetImpl` class are listed in table 10.5.

Method	Description
<code>absolute(int row)</code>	This method moves the cursor to the specified row in the <code>RowSet</code> .
<code>afterLast()</code>	This method moves the cursor to the end of the <code>RowSet</code> object just after the last row.
<code>beforeFirst()</code>	This method moves the cursor to the beginning of the <code>RowSet</code> object just before the first row.
<code>first()</code>	This method moves the cursor to the first row of the <code>RowSet</code> object.
<code>deleteRow()</code>	This method deletes the current row from the <code>RowSet</code> object and the underlying database. An event notification is sent to all the listeners that a row has changed.
<code>insertRow()</code>	This method inserts the contents of the insert row into the <code>RowSet</code> and in the database. An event notification is sent to all the listeners that a row has changed.
<code>last()</code>	This method moves the cursor to the last row in the <code>RowSet</code> .
<code>boolean isLast()</code>	This method returns true if the cursor is on the last row of the <code>RowSet</code> .
<code>boolean isAfterLast()</code>	This method returns true if the cursor is after the last row of the <code>RowSet</code> .

Method	Description
isBeforeFirst()	This method returns true if the cursor is before the first row of the RowSet.
next()	This method moves the cursor to the next row below the current row in the RowSet.
previous()	This method moves the cursor to the previous row above the current row in the RowSet.
moveToCurrentRow()	This method moves the cursor to the remembered position. This method is generally used when inserting a new row in the RowSet object.
moveToInsertRow()	This method moves the cursor to the insert row of a RowSet object.
updateDate(int column, java.sql.Date date)	This method updates the specified column with the date value given in the parameter.
updateInt(int column, int i)	This method updates the specified column with the given int value.
updateString(int column, String str)	This method updates the specified column with the given string value.
updateRow()	This method updates the database with the new contents of the current row of the RowSet. An event notification is sent to all the listeners that a row has changed.

Table 10.5: Methods of JdbcRowSetImpl Class

A row of data can be updated, inserted and deleted in a way similar to an updatable ResultSet object. Any changes made to a JdbcRowSet object data are also reflected on the database.

→ Update

Updating a record from a RowSet object involves navigating to that row, updating data from that row and finally updating the database as shown in Code Snippet 31.

Code Snippet 31:

```
jdbcRs.absolute(3);
jdbcRs.updateFloat("PRICE", 15.75f);
jdbcRs.updateRow();
```

In the code, the method `absolute(int lineNumber)` moves the cursor to the specified row number. The different update methods are used to change the values of the individual cells (columns) as shown in the code snippet. The `updateRow()` method is invoked to update the database.

→ Insert

To insert a record into the `JdbcRowSet` object the `moveToInsertRow()` method is invoked. The current cursor position is remembered and the cursor is then positioned on the insert row. The insert row is a special buffer row provided by an updatable result set for constructing a new row. When the cursor is in this row only the update, get and `insertRow()` methods can be called. All the columns must be given a value before the `insertRow()` method is invoked. The `insertRow()` method inserts the newly created row in the result set as shown in Code Snippet 32.

Code Snippet 32:

```
jdbcRs.moveToInsertRow();
jdbcRs.updateString("COF_NAME", "HouseBlend");
jdbcRs.updateInt("SUP_ID", 49);
jdbcRs.updateFloat("PRICE", 7.99f);
jdbcRs.updateInt("SALES", 0);
jdbcRs.updateInt("TOTAL", 0);
jdbcRs.insertRow();
```

The `moveToCurrentRow()` method moves the cursor to the remembered position.

→ Delete

Deleting a row from the `JdbcRowSet` object is simple as shown in Code Snippet 33.

Code Snippet 33:

```
jdbcRs.last();
jdbcRs.deleteRow();
```

The code snippet deletes the last row from the RowSet as well as from the underlying database. The cursor is moved to the last row and the `deleteRow()` method deletes the row from the RowSet and the database as well.

→ Retrieve

A `JdbcRowSet` object is scrollable, which means that the cursor can be moved forward and backward by using the `next()`, `previous()`, `last()`, `absolute()`, and `first()` methods. Once the cursor is on the desired row, the getter methods can be invoked on the RowSet to retrieve the desired values from the columns.

10.6.5 Implementation of a Disconnected RowSet

The use of high bandwidth networks is still unable to ensure an un-interrupted connection to the database at all times. Reliable database connections are limited resources and must be used wisely. Hence, a disconnected RowSet is useful since it does not require a continuous connection with the database. A disconnected RowSet stores its data in memory and operates on that data rather than directly operating on the data in the database.

A CachedRowSet object is an example of a disconnected RowSet object. These objects store or cache data in memory and operate on this data rather than manipulate data stored in the database.

The CachedRowSet is the parent interface for all disconnected RowSet objects. A CachedRowSet object generally derives its data from a relational database but, it is also capable of retrieving and storing data from a data source which stores data in a tabular form.

A CachedRowSet object includes all the capabilities of a JdbcRowSet object. In addition, it can perform the following:

- Connect to a data source and execute a query.
- Read the data from the resulting ResultSet object and get populated.
- Manipulate data when disconnected.
- Reconnect to the data source to write the changes back to it.
- Resolve conflicts with the data source, if any.

A disconnected RowSet needs reader and writer objects to read and write data from its data source. This is done using a SyncProvider implementation. These reader and writer objects work behind the scenes and their implementation is beyond the scope of this session.

The CachedRowSet object is capable of getting data from a relational database or from any other data source that stores its data in a tabular format. A key column must be set before data can be saved to the data source.

A CachedRowSet object can be created in any of the following ways:

→ Using the default constructor

The CachedRowSetImpl class is a standard implementation of the CachedRowSet interface. A SyncProvider is a synchronization mechanism which provides reader/writer capabilities for disconnected RowSet objects. The default constructor passes a default SyncProvider to the CachedRowSet instance.

Code Snippet 34 shows use of the default constructor of the `CachedRowSetImpl`.

Code Snippet 34:

```
CachedRowSet crs = new CachedRowSetImpl();
```

The `username`, `password`, `url`, and `command` properties are set before the `execute()` method is invoked. This helps connect to a data source and select the data to hold as illustrated in Code Snippet 35.

Code Snippet 35:

```
crs.setUsername("scott");
crs.setPassword("tiger");
crs.setUrl("jdbc:protocolName:datasourceName");
crs.setCommand("Select * From Employee");
crs.execute();
```

→ Using the SyncProvider implementation

The fully qualified name of a `SyncProvider` implementation is passed as an argument to the `CachedRowSetImpl` constructor. The `CachedRowSet` object can be populated with data by setting properties similar to the `JdbcRowSet` object.

Code Snippet 36 shows use of `SyncProvider`.

Code Snippet 36:

```
CachedRowSet crs2 = new CachedRowSetImpl(com.myJava.providers.HighAvailabilityProvider);
```

The code creates a `CachedRowSet` instance using `SyncProvider` implementation class, `HighAvailabilityProvider`.

A `CachedRowSet` object cannot be populated with data until the `username`, `password`, `url`, and `datasourceName` properties for the object are set. These properties can be set by using the appropriate setter methods of the `CachedRowSet` interface. However, the `RowSet` is still not usable by the application. For the `CachedRowSet` object to retrieve data the `command` property needs to be set. This can be done by invoking the `setCommand()` method and passing the SQL query to it as a parameter.

Code Snippet 37 sets the `command` property with a query that produces a `ResultSet` object containing all the data in the table **STUDENT**.

Code Snippet 37:

```
crs.setCommand("select * from STUDENT");
```

If any updates are to be made to the `RowSet` object then the key columns need to be set. The key columns help to uniquely identify a row, as a primary key does in a database.

Each RowSet object has key columns set for it, similar to a table in a database which has one or more columns set as primary keys for it.

Code Snippet 38 demonstrates how key columns are set for the CachedRowSet object.

Code Snippet 38:

```
CachedRowSet crsStudent = new CachedRowSetImpl();
int[] keyColumns = {1, 2};
crsStudent.setKeyColumns(keyColumns);
```

Key columns are used internally, hence it is of utmost importance to ensure that these columns can uniquely identify a row in the RowSet object.

The `execute()` method is invoked to populate the RowSet object. The data in the RowSet object is obtained by executing the query in the command property. The `execute()` method of a disconnected RowSet performs many more functions than the `execute()` method of a connected RowSet object. A CachedRowSet object has a SyncProvider object associated with it. This SyncProvider object provides a RowSetReader object which on invocation of the `execute()` method, establishes a connection with the database using the username, password and URL, or datasourceName properties set earlier. The reader object then executes the query set in the command property and the RowSet object is populated with data. The reader object then closes the connection with the datasource and the CachedRowSet object can be used by the application.

Updation, insertion, and deletion of rows is similar to that in a JdbcRowSet object. However, the changes made by the application to the disconnected RowSet object need to be reflected back to the datasource. This is achieved by invoking the `acceptChanges()` method on the CachedRowSet object. Like the `execute()` method this method also performs its operations behind the scenes. The SyncProvider object also has a RowSetWriter object which opens a connection to the database, writes the changes back to the database and then finally, closes the connection to the database.

10.6.6 Using CachedRowSet Object

A row of data can be updated, inserted, and deleted in a CachedRowSet object. Changes in data are reflected on the database by invoking the `acceptChanges()` method.

→ Update

Updating a record from a RowSet object involves navigating to that row, updating data from that row and finally updating the database.

Code snippet 39 illustrates the updation of row.

Code Snippet 39:

```
if (crs.getInt("EMP_ID") == 1235) {
    int currentQuantity = crs.getInt("BAL_LEAVE") + 1;
```

```

System.out.println("Updating balance leave to " + currentQuantity);
crs.updateInt("BAL_LEAVE", currentQuantity + 1);
crs.updateRow();
// Synchronizing the row back to the DB
crs.acceptChanges(con);    save changes to server

```

In the code snippet, the **LeaveBalance** column is updated. The method moves the cursor above the first row of the RowSet. The `updateInt()` method is used to change the value of the individual cell (column) as shown in the code snippet. The `updateRow()` method is invoked to update the memory. The `acceptChanges()` method saves the changes to the data source.

→ Insert

To insert a record into the CachedRowSet object the `moveToInsertRow()` method is invoked. The current cursor position is remembered and the cursor is then positioned on an insert row. The insert row is a special buffer row provided by an updatable result set for constructing a new row. When the cursor is in this row only the update, get and `insertRow()` methods can be called. All the columns must be given a value before the `insertRow()` method is invoked. The `insertRow()` method inserts the newly created row in the result set. The `moveToCurrentRow()` method moves the cursor to the remembered position.

→ Delete

Deleting a row from a CachedRowSet object is simple. Code Snippet 40 illustrates this.

Code Snippet 40:

```

while (crs.next()) {
    if (crs.getInt("EMP_ID") == 12345) {
        crs.deleteRow();
        break;
    }
}

```

The code snippet deletes the row containing the employee name id as 12345 from the row set. The cursor is moved to the appropriate row and the `deleteRow()` method deletes the row from the row set.

→ Retrieve

A `CachedRowSet` object is scrollable, which means that the cursor can be moved forward and backward by using the `next()`, `previous()`, `last()`, `absolute()`, and `first()` methods. Once the cursor is on the desired row the getter methods can be invoked on the `RowSet` to retrieve

the desired values from the columns.

10.6.7 Event Notification Mechanism in RowSet

A RowSet object is inherently a JavaBeans component. Hence, it possesses properties like JavaBeans and has the JavaBeans Event Notification Mechanism. The fields of a RowSet are the JavaBean properties of the RowSet. RowSet objects follow the JavaBeans Event Notification Model for processing events. According to this model, all components that need to be notified of an event need to be registered as event listeners for the component generating the events. The following events trigger notifications in RowSet objects:

- Movement of a cursor
- Insertion, updation, or deletion of a row
- Changing the entire RowSet contents

All components registered as listeners for events generated by RowSet must implement the RowSetListener interface. Consider the example of a JLabel object lblStatus listening for events generated by a RowSet object rs.

Code Snippet 41 registers the label with the RowSet as a listener.

Code Snippet 41:

```
rs.addListener(lblStatus);
```

Due to this line of code, the label lblStatus will be sent an event notification every time a cursor is moved, a row is updated, inserted, or deleted or the entire RowSet contents are changed.

10.7 Check Your Progress

1. Which of these statements about stored procedure are true?

A.	Stored procedure is a group of SQL statements that form a logical unit and perform a particular task.		
B.	Stored procedures are programs that allow an application program to run in two parts such as the application on the client and stored procedure on server.		
C.	Applications that use stored procedures have access to increased memory and disk space on the server computer.		
D.	Users of the client applications that call the stored procedure need database privileges that the stored procedure requires.		
E.	By reusing one common procedure, a stored procedure cannot provide a highly efficient way to address the recurrent situation of repeated tasks.		

(A)	A, B, and C	(C)	B, C, and E
(B)	B, C, and D	(D)	A, D, and E

2. Which of these statements about Scrollable ResultSet are true?

A.	In Scrollable ResultSet, the cursor is positioned on the first row.		
B.	A default ResultSet object is not updatable and has a cursor that moves forward only.		
C.	The ResultSet should be compulsorily closed after a COMMIT statement.		
D.	Holdability refers to the ability to check whether the cursor stays open after a COMMIT.		
E.	The createStatement method has two arguments namely resultSetType and resultSetConcurrency.		

(A)	A, B, and C	(C)	B, C, and E
(B)	B, D, and E	(D)	A, D, and E

3. Which of these statements about ResultSet constant values are true?

A.	TYPE_SCROLL_INSENSITIVE is the default type of result set.		
B.	In TYPE_FORWARD_ONLY result set, the cursor can only be used to process from the beginning of a ResultSet to the end of it.		
C.	In TYPE_SCROLL_INSENSITIVE result set, the SQL queries applied on a particular field will have a direct impact on the ResultSet.		
D.	The TYPE_SCROLL_SENSITIVE type of cursor is sensitive to changes made to the database while it is open.		
E.	The TYPE_FORWARD_ONLY type of result set is not scrollable, not positionable, and not sensitive.		

(A)	A, B, and C	(C)	B, C, and E
(B)	B, D, and E	(D)	A, D, and E

4. Which of the following statements about RowSet are true?

A.	A RowSet contains a set of rows of data.
B.	A RowSet has to be made scrollable and updatable at the time of creation.
C.	A RowSet is a JavaBeans component which has to programmatically notify all registered event listeners.
D.	A connected RowSet can read data from a non relational database source also.
E.	Scrollability and Updatability of a RowSet is independent of the JDBC driver.

(A)	A, C	(C)	C, E
(B)	B, D	(D)	A, E

5. Which of these statements about Updatable “ResultSet” are true?

A.	Updatable ResultSet is the ability to update rows in a result set using SQL commands, rather than using methods in the Java programming language.
B.	The UpdateXXX() method of ResultSet is used to change the data in an existing row.
C.	Updatability in a result set is associated with concurrency in database access because you cannot have multiple write locks concurrently.
D.	In CONCURRENCY.READ_ONLY type of result set, the updates, inserts and deletes can be performed on the result set but it cannot be copied to the database.
E.	The concurrency type of a result set determines whether it is updatable or not.

(A)	A, C	(C)	C, E
(B)	B	(D)	D, E

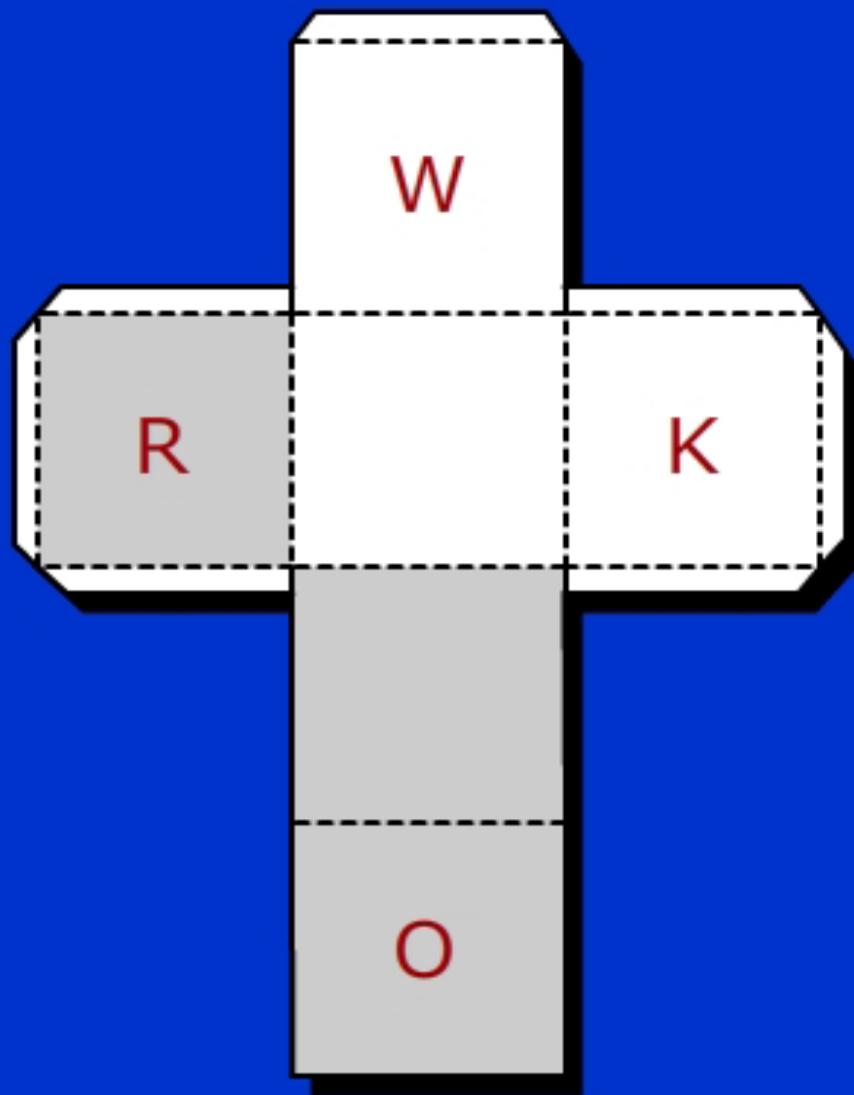
10.7.1 Answers

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

Summary

- A ResultSet object maintains a cursor pointing to its current row of data.
- Updatable ResultSet is the ability to update rows in a result set using methods in the Java programming language rather than SQL commands.
- A stored procedure is a group of SQL statements.
- A batch update is a set of multiple update statements that is submitted to the database for processing as a batch.
- A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.
- RowSet is an interface that is derived from the ResultSet interface.
- A JdbcRowSet is the only implementation of a connected RowSet.

WORK



ASSIGNMENT:

Form the cube to read '**WORK**'.

"Practice does not make perfect. Only perfect practice makes perfect."

- Vince Lombardi

Practice the Practicals for Perfection @

www.onlinevarsity.com

Session 11

Design Patterns

Welcome to the Session, **Design Patterns**.

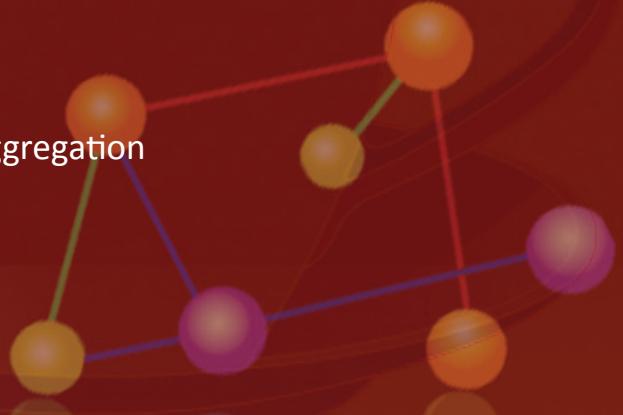
In simplest terms, a design pattern can be thought of as a description of a solution to a frequently observed problem. It can be considered as a template or a best practice suggested by expert programmers for commonly occurring problems. Code for the problem is then created on the basis of the design pattern. Most of the popular design patterns are based on the concept of polymorphism.

The session begins with describing briefly about polymorphism in Java. It then explains how to apply polymorphism by overriding the various methods of the Object class. Then, the session explains design patterns in detail.

In this session, you will learn to:

- Describe polymorphism
- Describe the procedure to override methods of the Object class
- Explain design patterns
- Describe the Singleton, Data Access Object (DAO), and Factory and Observer design patterns
- Describe delegation

→ Explain composition and aggregation



11.1 Introduction

In the biological world, when certain organisms exhibit many different forms, it is referred to polymorphism. A similar concept can be applied to Java, where a subclass can have its own unique behavior even while sharing certain common functionalities with the parent class.

11.2 Implementing Polymorphism

The concept of method overriding is an example of polymorphism in object-oriented programming in which the same method behaves in a different manner in super class and in subclass.

A fundamental feature of OOP is that when a class is inherited from another class, then the subclass inherits the members of the super class, including its methods. These methods can also be overridden, that is, they can be given new functionality, provided the super class method has not been marked as final. In OOP, overriding means to override the functionality of an existing method. The benefit of overriding is the ability to define a behavior that is specific to the subclass type. In other words, it means that the subclass can implement a parent class method based on its requirement.

Code Snippet 1 illustrates method overriding. It assumes that a super class named **Car** has been created having two methods, **accelerate()** and **printDescription()**. In Code Snippet 1, a sub class named **LuxuryCar** is created based on **Car** and overrides the methods of the **Car** class.

Code Snippet 1:

```
class LuxuryCar extends Car {
    // LuxuryCar defines an additional feature named perks
    public String perks;
    public LuxuryCar(int mileage, String color, String make, String perks)
    {
        super(mileage, color, make);
        this.perks = perks;
    }
    public void accelerate() {
        System.out.println("Luxury Car is Accelerating");
    }
    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "Luxury car is a: " + this.perks +
            ".");
    }
}
```

```
}
```

```
}
```

In Code Snippet 1, the following have been done:

- The `printDescription()` and `accelerate()` methods are overridden.
- The subclass, that is `LuxuryCar` class, overrides the `printDescription()` and `accelerate()` method and prints unique information.

Code Snippet 2 creates two instances of type `Car`, instantiates them, and invokes the `accelerate()` and `printDescription()` methods on each instance respectively.

Code Snippet 2:

```
public class PolymorphismTest {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Car objCar, objLuxuryCar;
        objCar = new Car(80, "Red", "BMW");
        objLuxuryCar = new LuxuryCar(120, "Yellow", "Ferrari", "Sports
                                   Car");
        objCar.accelerate();
        objCar.printDescription();
        //System.out.println("Now inside LuxuryCar");
        objLuxuryCar.accelerate();
        objLuxuryCar.printDescription();
    }
}
```

When Code Snippet 2 is executed, the Java Virtual Machine (JVM) calls the appropriate method, which is defined in each variable. This behavior is called virtual method invocation. This is an important aspect of polymorphism features in Java.

Note: JVM does not call the method defined by the variable's type.

11.3 Overriding the Methods of Object Class

Since `java.lang.Object` is by default the superclass of all classes, its methods too can be overridden.

In the class hierarchy tree, the `java.lang.Object` is the root class of every Java class. All other classes descend from the root class. In addition, there is no need for a programmer to declare that the class extends from the `Object` class. It is implicitly done by the compiler.

As `Object` is the root class, its methods can be overridden by any class (unless the methods are marked as final).

Following are the methods that can be overridden with a different functionality as compared to the root class:

- `public boolean equals(Object obj)`
- `public int hashCode()`
- `public String toString()`

11.3.1 `equals()` and `hashCode()` Methods

The `equals()` method compares two objects to determine if they are equal.

In general, the two different types of equality are namely, reference equality and logical equality. When the physical memory locations of the two strings are same, this is called reference equality. When data in the objects are the same, it is called logical equality.

The `equals()` method helps to [check logical or value equality](#). To test reference equality, you will use the `==` operator.

Code Snippet 3 shows an example that checks reference equality.

Code Snippet 3:

```
public class EqualityTest {  
    /**  
     * @param args the command line arguments  
     */  
  
    public static void main(String[] args) {  
        String strAObj = new String("JOHN");  
        String strBObj = new String("JOHN");  
    }  
}
```

```

String strCObj = new String("ANNA");
// Create a String reference and assign an existing
// String's reference to it so that both references point
// to the same String object in memory.

String strEObj = strAObj;
// Print out the results of various equality checks
System.out.println(strAObj == strBObj);
System.out.println(strAObj == strCObj);
System.out.println(strAObj == strEObj);
}
}

```

In Code Snippet 3, the equality operator (`==`) compares the memory addresses of the two strings. Therefore, when `strAObj` is compared to `strBObj`, the result is false, although their value is same, which is JOHN. A comparison between `strAObj` and `strCObj` returns false because the references of the two different String objects are different addresses. However, notice that when `strAObj` is compared to `strEObj`, the result is true because they point to the same memory location.

Figure 11.1 displays the output.

```

run:
false
false
true
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 11.1: Reference Equality Output

Code Snippet 4 shows an example that checks for logical or value equality.

Code Snippet 4:

```

public class LogicalEqualityTest {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
// TODO code application logic here

```

```

String strAOBJ = new String("JOHN");
String strBObj = new String("JOHN");
String strCObj = new String("ANNA");
// Create a String reference and assign an existing
// String's reference
// to it so that both references point to the same
// String object in memory.
String strEObj = strAOBJ;

// Print the results of the equality checks
System.out.println("=====");
System.out.println("Logical or Value Equality");
System.out.println("=====");
//Tests logical or value equality
System.out.println(strAOBJ.equals(strBObj));
System.out.println(strAOBJ.equals(strCObj));
System.out.println(strAOBJ.equals(strEObj));
}
}

```

Figure 11.2 displays the output.

```

=====
Logical or Value Equality
=====
true
false
true
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 11.2: Logical Equality - Output

In Code Snippet 4, the `equals()` method is used to check for logical equality. Note that the `equals()` method is implicitly inherited from the `Object` class. The `String` class overrides the `equals()` method and compares the two `String` objects character by character. This is done because the `equals()` method inherited from `Object` class performs only reference equality and not value equality.

The default implementation of the `equals()` method in `Object` class is as follows:

```
public boolean equals(Object other)
{
    return this == other;
}
```

In this code, the `equals()` method in the `Object` class uses the equality operator (`==`) to check if the object references are equal.

To test if two objects contains the same information, the `equals()` method can be overridden in a class. However, any code that overrides the `equals()` method must override the `hashCode()` method because overriding the `equals()` method makes the `Object`'s implementation of `hashCode()` invalid.

The `hashCode()` method of `Object` class returns the object's memory address in hexadecimal format. Consider the following points about this method:

- It is used along with the `equals()` method in hash-based collections such as `Hashtable`.
- If two objects are equal, their hash code should also be equal.

Code Snippet 5 overrides the default implementation of the `equals()` and `hashCode()` methods to include testing for logical equality.

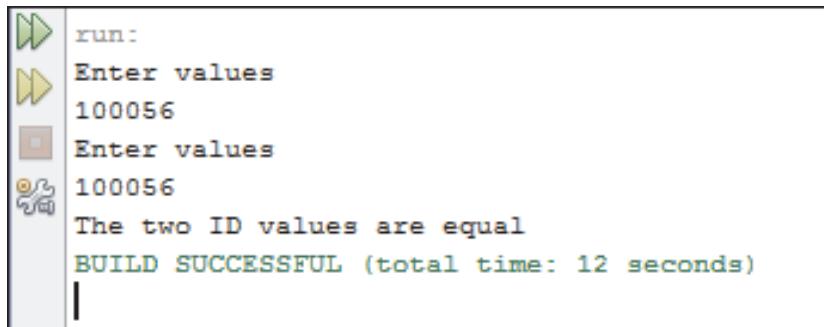
Code Snippet 5:

```
public class Student
{
    private int ID;
    public int getID()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter values");
        int ID = Integer.parseInt(sc.nextLine());
        return ID;
    }
    public boolean equals(Object obj)
    {
        if (getID() == ((Student) obj).getID())
            return true;
    }
}
```

```
        else
            return false;
    }
    public int hashCode()
    {
        return ID;
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        Student s1 = new Student();
        Student s2 = new Student();
        if (s1.equals(s2))
            System.out.println("The two ID values are equal");
        else
            System.out.println("The two ID values are not equal");
    }
}
```

Here, the **Student** class defines three methods in addition to `main()`. The `getID()` method accepts an ID value from the standard input. The overridden `equals()` method accepts a parameter of type `Object`. It compares the ID value of this object with the ID value of the current object. The overridden `hashCode()` method returns the ID value. In the `main()` method, two objects of `Student` class, namely, `s1` and `s2` are created. Then, the `equals()` method is invoked on `s1` and the object `s2` is passed as a parameter. This means, that the ID value of the object `s1` will be compared with the ID value of `s2`. Depending on the user input, the output will be displayed accordingly.

Figure 11.3 shows an example of the output of the code.



```

run:
Enter values
100056
Enter values
100056
The two ID values are equal
BUILD SUCCESSFUL (total time: 12 seconds)
|
```

Figure 11.3: Output of Code Snippet 5

11.3.2 `toString()` Method

The `toString()` method of `Object` class returns a string representation of the object. It is typically used for debugging. It is recommended that the method should always be overridden because the `String` representation depends on the object.

Code Snippet 6 shows an example of overriding the `toString()` method in a class.

Code Snippet 6:

```

public class Exponent {
    private double num, exp;
    public Exponent(double num, double exp) {
        this.num = num;
        this.exp = exp;
    }
    /* Returns the string representation of this number.
       The format of string is "Number + e Value" where Number is the
       number value and e Value is the exponent part.*/
    @Override
    public String toString() {
        return String.format(num + "E+" + exp);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
```

```
Exponent c1 = new Exponent(10, 15);  
System.out.println(c1);  
}  
}
```

After executing, the code prints an output as shown in figure 11.4.

```
run:  
10.0E+15.0  
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 11.4: Output of Code Snippet 6

An alternative to polymorphism (that is, overriding methods) is to use the `instanceof` operator.

11.4 The `instanceof` Operator

The `instanceof` operator is used to compare an object to a specified type such as instance of a class and an instance of a subclass. When using the `instanceof` operator, it should be noted that null is not an instance of anything.

Code Snippet 7 illustrates an example of the `instanceof` operator.

Code Snippet 7:

```
class Employee {  
    int empcode;  
    String name;  
    String dept;  
    int bonus;  
}  
  
class Manager extends Employee {  
    String name;  
    int mgrid;  
}  
  
public class Square {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {
```

```

Employee emp1 = new Employee();
Scanner sc = new Scanner(System.in);
System.out.println("Enter values");
emp1.name = sc.nextLine();
Employee m1 = new Manager();
m1.name = sc.nextLine();
if (emp1 instanceof Employee) {
    emp1.bonus = 7000;
    System.out.println(emp1.name + " is an employee and has bonus
                           "+emp1.bonus);
}
if (emp1 instanceof Manager) {
    emp1.bonus = 12000;
    System.out.println(emp1.name + " is a manager and has bonus
                           "+emp1.bonus);
}
if (m1 instanceof Employee) {
    m1.bonus = 7000;
    System.out.println(m1.name + " is an employee and has bonus "++
                           m1.bonus);
}
if (m1 instanceof Manager) {
    m1.bonus = 12000;
    System.out.println(m1.name + " is a manager and has bonus "++
                           m1.bonus);
}
}
}

```

In Code Snippet 7, the code defines the following:

- Parent class, **Employee**
- Child class, **Manager** that inherits from the parent

Figure 11.5 displays the output of Code Snippet 7.

```

run:
Enter values
Jake Kreston
Peter Smith
Jake Kreston is an employee and has bonus 7000
Peter Smith is an employee and has bonus 7000
Peter Smith is a manager and has bonus 12000
BUILD SUCCESSFUL (total time: 12 seconds)

```

Figure 11.5: Output of Code Snippet 7

As seen in the output, Peter Smith is both an employee and a manager. Therefore, Peter will get the bonus of an employee as well as that of a manager. The `instanceof` operator can help in determining the type of the object at runtime.

11.5 Design Patterns

A design pattern is a clearly defined solution to problems that occur frequently. It can be considered as a template or a best practice suggested by expert programmers. So, if an experienced developer educates another developer about a particular factory pattern being used to solve a problem, the other developer can precisely understand how to deal with a similar problem. A design pattern is a great help to inexperienced developers. They can study the patterns and the related problems and learn good details about the software design. This reduces the learning curve. The proper use of design patterns results in increased code maintainability.

Design patterns are based on the fundamental principles of object oriented design. A design pattern is not an implementation, nor is it a framework. It cannot be installed using code.

As of today, there are certain standard and popularly used design patterns that have been developed after long periods of research and trial and error by various software developers.

11.5.1 Types of Patterns

Table 11.1 lists different types of design patterns.

Pattern Category	Description	Types
Creational Patterns	They offer ways to create objects while hiding the logic of creation, instead of instantiating objects using the new operator.	Singleton Pattern
		Factory Pattern
		Abstract Factory Pattern
		Builder Pattern
		Prototype Pattern

Pattern Category	Description	Types
Structural Patterns	They are related to class and object composition. Interfaces are composed using the concept of inheritance and new ways are defined to compose objects to get different functionalities.	Adapter Pattern Composite Pattern Proxy Pattern Flyweight Pattern Facade Pattern Bridge Pattern Decorator Pattern
Behavioral Patterns	They are related to communication between various objects.	Template Method Pattern Mediator Pattern Chain of Responsibility Pattern Observer Pattern Strategy Pattern Command Pattern State Pattern Visitor Pattern Iterator Pattern Memento Pattern

Table 11.1: Types of Design Patterns

11.5.2 The Singleton Pattern

Certain class implementations can be instantiated only once. The singleton design pattern provides complete information on such class implementations. To do so, usually, a static field is created representing the class. The object that the static field references can be created at the time when the class is initialized or the first time the `getInstance()` method is invoked. The constructor of a class using the singleton pattern is declared as private to prevent the class from being instantiated.

It is recommended to use singleton classes to concentrate access to particular resources into a single class instance.

To implement a singleton design pattern, perform the following steps:

1. Use a static reference to point to the single instance.
2. Then, add a single private constructor to the singleton class.
3. Next, a public factory method is declared static to access the static field declared in Step 1. A factory method is a method that instantiates objects. Similar to the concept of a factory that manufactures products, the job of the factory method is to manufacture objects.

Note - A public factory method returns a copy of the singleton reference.

4. Use the `static getInstance()` method to get the instance of the singleton.

Consider the following when implementing the singleton design pattern:

- The reference is finalized so that it does not reference a different instance.
- The private modifier allows only same class access and restricts attempts to instantiate the singleton class.
- The factory method provides greater flexibility. It is commonly used in singleton implementations.
- The singleton class usually includes a private constructor that prevents a constructor to instantiate the singleton class.
- To avoid using the factory method, a public variable can be used at the time of using a static reference.

Code Snippet 8 illustrates the implementation of the singleton design pattern.

Code Snippet 8:

```
class SingletonExample {  
    private static SingletonExample singletonExample = null;  
    private SingletonExample() {  
    }  
    public static SingletonExample getInstance() {  
        if (singletonExample == null) {  
            singletonExample = new SingletonExample();  
        }  
        return singletonExample;  
    }  
    public void display() {  
        System.out.println("Welcome to Singleton Design Pattern");  
    }  
}
```

In Code Snippet 8, note the following:

- The `SingletonExample` class contains a `private static SingletonExample` field.
- There is a `private constructor`. Therefore, the `class cannot be instantiated by outside classes`.
- The public static `getInstance()` method returns the only `SingletonExample` instance. If the `instance does not exist`, the `getInstance()` method creates it.
- There is a public `sayHello()` method that can test the singleton.

Note - `SingletonExample` class is an example of a typical singleton class.

Code Snippet 9 includes the `SingletonTest` class that calls the static `SingletonExample.getInstance()` method to get the `SingletonExample` singleton class.

Code Snippet 9:

```
public class SingletonTest {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        SingletonExample singletonExample = SingletonExample.getInstance();
        singletonExample.display();                                can't write like this
                                                               new SingletonExample()
    }
}
```

In Code Snippet 9, note the following:

- The `display()` method is called on the singleton class.
- The output of the program is “Welcome to Singleton Design Pattern”.

11.5.3 Design Patterns and Interfaces

Consider a scenario where following programs are created to automate certain tasks of an automobile:

- A program that stops the vehicle on red light
- A program that accelerates the vehicle

- A program that turns the vehicle in different directions

There can be such many other programs for automobile automation.

Now, all these programs need not be made by a single individual. Each program can be owned by different programmers or organizations.

To integrate all these programs from different sources on an automobile, there has to be a medium that describes how a software interacts. This medium is the interface. So, when the programmers decide to write a code for a similar target (such as automation of automobile), they comply to this interface.

In Java, interfaces include constant fields. They can be used as reference types. In addition, they are important components of many design patterns.

Java uses interfaces to define type abstraction. Outlining abstract types is a powerful feature of Java. The following are the benefits of abstraction:

- **Vendor-specific Implementation:** Developers define the methods for the `java.sql` package. The communication between the database and the `java.sql` package occurs using the methods. However, the implementation is vendor-specific.
- **Tandem Development:** Based on the business API, the application's UI and the business logic can be developed simultaneously.
- **Easy Maintenance:** Improved classes can replace the classes with logic errors anytime.

Note - Java also uses abstract classes to define type abstraction.

The following provides additional information on interfaces:

- Interfaces cannot be instantiated.
- Only classes can implement interfaces.
- An interface can be defined just as a new class is created.
- Interfaces can also be extended by other interfaces. An interface can extend any number of interfaces.
- They can include constant fields.
- Most design patterns use interfaces.

An interface declaration includes the following:

- Modifiers
- The keyword interface

- The interface name
- A comma-separated list of parent interfaces that it can extend
- The interface body

Code Snippet 10 shows an interface declaration.

Code Snippet 10:

```
public interface IAircraft {
    public int passengerCapacity = 400;
    // method signatures
    void fly();
    .....
    // more method signatures
}
```

Only constant fields are allowed in an interface. A field is implicitly `public`, `static`, and `final` when an interface is declared. As a good design practice, it is recommended to distribute constant values of an application across many classes and interface.

Defining a new interface defines a new reference data type. Consider the following points regarding reference types:

- Interface names can be used anywhere. In addition, any other data type name can be used.
- The `instanceOf` operator can be used with interfaces.
- If a reference variable is defined whose type is an interface, then any object assigned to it must be an instance of a class that implements the interface.
- Interfaces implicitly include all the methods from `java.lang.Object`.
- If an object includes all the methods outlined in the interface but does not implement the interface, then the interface cannot be used as a reference type for that object.

11.5.4 Difference between Class Inheritance and Interface Inheritance

A class can extend a single parent class whereas it can implement multiple interfaces. When a class extends a parent class, only certain functionalities can be overridden in the inherited class. On the other hand, when a class inherits an interface, it implements all functionalities of the interfaces.

A class is extended because certain classes require detailed implementation based on the super class. However, all the classes require some of the methods and properties of the super class. On the other hand, an interface is used when there are multiple implementations of the same functionality.

Table 11.2 provides the comparison between an interface and an abstract class.

Interface	Abstract Class
Inheritance of several interfaces by a class is supported.	Inheritance of only one abstract class by a class is supported.
Requires more time to find the actual method in the corresponding classes.	Requires less time to find the actual method in the corresponding classes.
Best used when various implementations only share method signatures.	Best used when various implementations use common behavior or status.

Table 11.2: Comparison between an Interface and an Abstract Class

11.5.5 Extending Interfaces

It is recommended to specify all uses of the interface right from the beginning. However, this is not always possible. In such an event, more interfaces can be created later. This way, interfaces can be used to extend interfaces.

Code Snippet 11 creates an interface called **IVehicle**.

Code Snippet 11:

```
public interface IVehicle {
    int getMileage(String s);
    . . .
}
```

Code Snippet 12 shows how a new interface can be created that extends **IVehicle**.

Code Snippet 12:

```
public interface IAutomobile extends IVehicle {
    boolean accelerate(int i, double x, String s);
}
```

In Code Snippet 12, an **IAutomobile** interface is created that extends **IVehicle**. Users can now either use the old interface or upgrade to the new interface. If a class implements the new interface, it must override all the methods from **IVehicle** and **IAutomobile**.

11.5.6 Implementation of IS-A and a HAS-A Relationships

It is a concept based on class inheritance or interface implementation. An IS-A relationship displays class hierarchy in case of class inheritance. For example, if the class **Ferrari** extends the class **Car**, the statement

Ferrari IS-A Car is true.

Figure 11.6 illustrates an IS-A relationship.

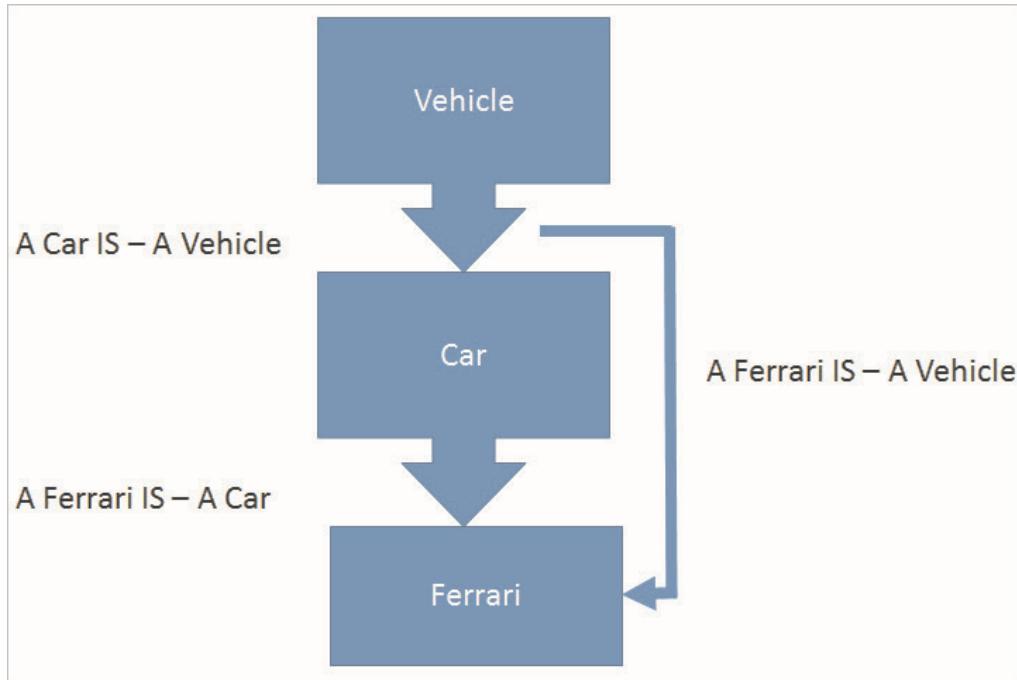


Figure 11.6: IS-A Relationship

The IS-A relationship is also used for interface implementation. This is done using keyword implements or extends.

An HAS-A relationship or a composition relationship uses instance variables that are references to other objects, such as a Ferrari includes an Engine. Figure 11.7 illustrates the example.

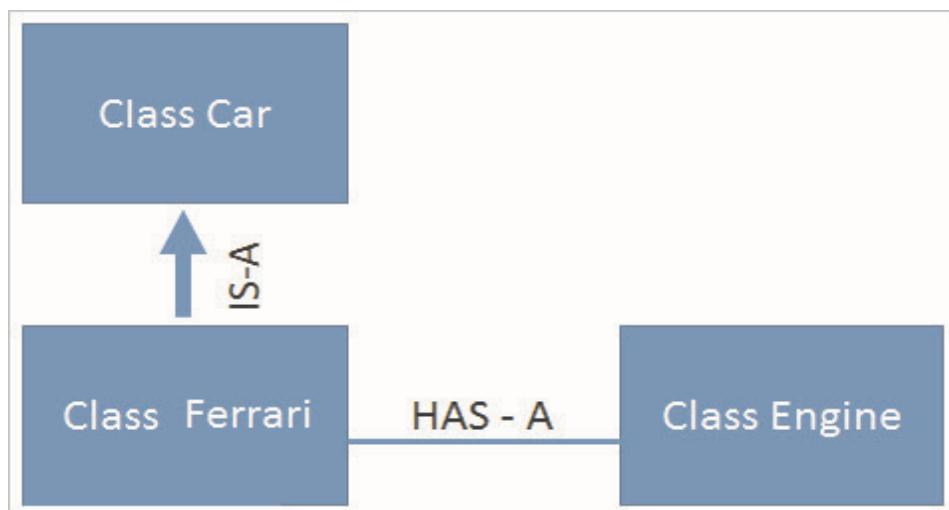


Figure 11.7: HAS-A Relationship Example

11.5.7 The Data Access Object (DAO) Design Pattern

The DAO pattern is used when an application is created that needs to persist its data. The DAO pattern involves a technique for separating the business logic from persistence logic, making it easier to implement and maintain an application.

The advantage of the Data Access Object approach is that it makes it flexible to change the persistence mechanism of the application without changing the entire application, as the data access layer would be separate and unaffected.

The DAO pattern uses the following:

- **DAO Interface:** This defines the standard operations for a model object. In other words it defines the methods used for persistence.
- **DAO Concrete Class:** This implements the DAO interface and retrieves data from a data source, such as a database.
- **Model Object or Value Object:** This includes the get/set methods that store data retrieved by the DAO class.

As the name suggests, DAOs can be used with any data objects, not necessarily databases. Thus, if your data access layer comprises data stores of XML files, DAOs can still be useful there.

Some of the various types of data objects that DAOs can support are as follows:

- **Memory based DAOs:** These represent temporary solutions.
- **File based DAOs:** These may be required for an initial release.
- **JDBC based DAOs:** These support database persistence.
- **Java persistence API based DAOs:** These also supports database persistence.

Figure 11.8 shows the structure of a DAO design pattern that will be created. In the figure, the following points are to be noted:

- **Book** object will act as a Model or Value Object
- **BookDao** is the DAO Interface
- **BookDaoImpl** is the concrete class that implements the DAO interface
- **DAOPatternApplication** is the main class. It will use **BookDao** to display the use of the DAO pattern

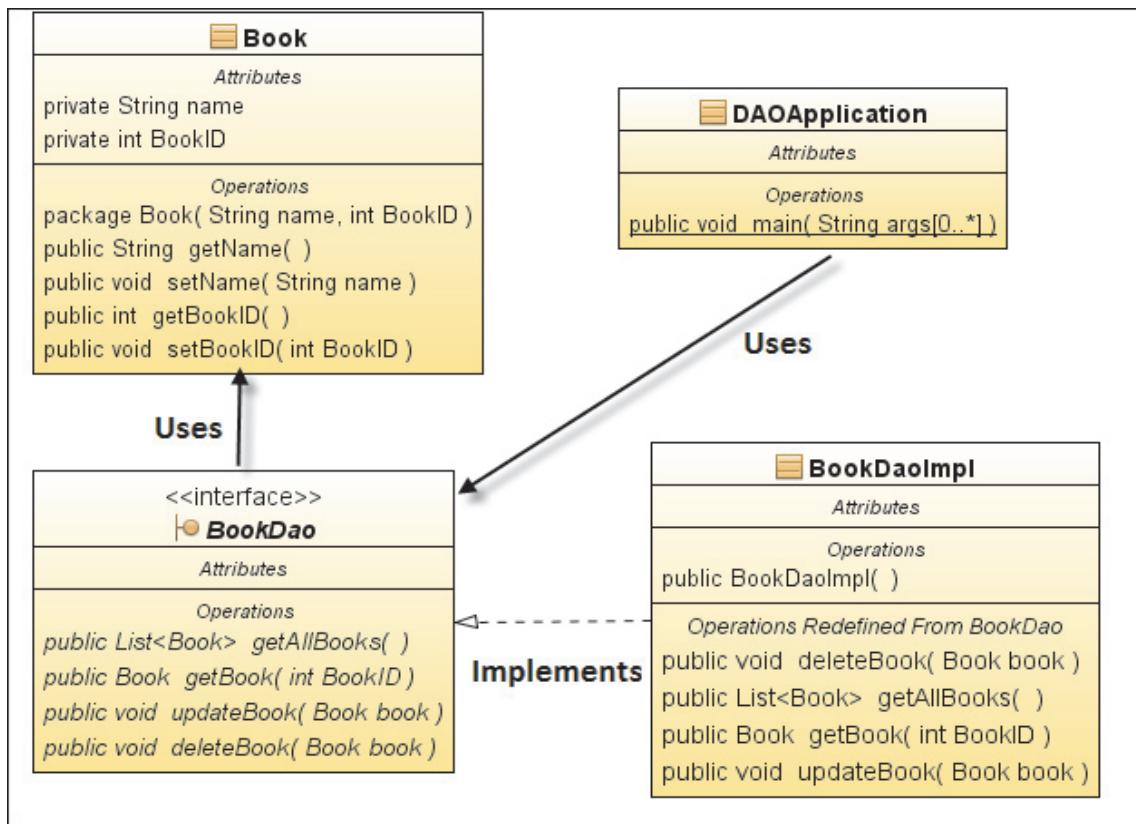


Figure 11.8: Structure of DAO Design Pattern

Based on figure 11.8, the following code snippets will create a DAO design pattern.

First, the Model or Value object is created which will store the book information.

Code Snippet 13 illustrates this.

Code Snippet 13:

```

class Book {
    private String name;
    private int BookID;
    Book(String name, int BookID) {
        this.name = name;
        this.BookID = BookID;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
    }
}
  
```

```

        this.name = name;
    }

    public int getBookID() {
        return BookID;
    }

    public void setBookID(int BookID) {
        this.BookID = BookID;
    }
}

```

Code Snippet 13 defines a **Book** class having a constructor and get/set methods.

Code Snippet 14 creates a DAO interface that makes use of this class.

Code Snippet 14:

```

interface BookDao {
    public java.util.List<Book> getAllBooks();
    public Book getBook(int BookID);
    public void updateBook(Book book);
    public void deleteBook(Book book);
}

```

Code Snippet 15 creates a class that implements the interface.

Code Snippet 15:

```

class BookDaoImpl implements BookDao {
    // list is working as a database
    java.util.List<Book> booksList;
    public BookDaoImpl(){
        booksList = new java.util.ArrayList<Book>();
        Book bookObj1 = new Book("Anna",1);
        Book bookObj2 = new Book("John",2);
        booksList.add(bookObj1);
        booksList.add(bookObj2);
    }
}

```

```

@Override
public void deleteBook(Book book) {
    booksList.remove(book.getBookID());
    System.out.println("Book: Book ID " + book.getBookID()
        + ", deleted from database");
}

// retrieve list of booksList from the database

@Override
public java.util.List<Book> getAllBooks() {
    return booksList;
}

@Override
public Book getBook(int BookID) {
    return booksList.get(BookID);
}

@Override
public void updateBook(Book book) {
    booksList.get(book.getBookID()).setName(book.getName());
    System.out.println("Book: Book ID " + book.getBookID() + ", updated in
        the database");
}
}

```

Code Snippet 16 displays the use of the DAO pattern.

Code Snippet 16:

```

public class DAOPatternApplication {
public static void main(String[] args) {
    BookDao bookDao = new BookDaoImpl();
    System.out.println("Book List:");
    //print all booksList
}
}

```

```

for (Book book : bookDao.getAllBooks()) {
    System.out.println("\nBookID : " + book.getBookID() + ", Name :
        " + book.getName() + " ");
}

//update book
Book book = bookDao.getAllBooks().get(0);
book.setName("Harry Long");
bookDao.updateBook(book);

//get the book
bookDao.getBook(0);
System.out.println("Book: [BookID : " + book.getBookID() + ", Name
        :" + book.getName() + " ]");
}
}

```

Figure 11.9 displays the output of the DAO design pattern.

```

Book List:

BookID : 1, Name : Anna

BookID : 2, Name : John
Book: Book ID 1, updated in the database
Book: [BookID : 1, Name : Harry Long ]
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 11.9: The DAO Design Pattern - Output

11.5.8 The Factory Design Pattern

Factory pattern is one of the commonly used design patterns in Java. It belongs to the creational pattern category and provides many ways to create an object. This pattern does not perform direct constructor calls when invoking a method. It prevents the application from being tightly coupled to a specific DAO implementation.

In factory pattern, note the following:

- The client is **not aware** of the **logic** that **helps create object**.
- It uses a **common interface** to **refer** to the **newly created object**.

Consider a scenario where certain automobile classes need to be designed. A general interface **Vehicle** with a common method **move ()** will be created. Classes, **Car** and **Truck** will implement this interface. A **VehicleFactory** class is created to get a **Vehicle** type and based on which the respective objects will be returned.

Code Snippet 17 creates a common interface for implementing the Factory pattern.

Code Snippet 17:

```
interface Vehicle {
    void move();
}
```

Code Snippet 18 creates a class that implements the interface.

Code Snippet 18:

```
class Car implements Vehicle {
    @Override
    public void move() {
        System.out.println("Inside Car::move() method.");
    }
}
```

Code Snippet 19 creates another class that implements the interface.

Code Snippet 19:

```
class Truck implements Vehicle {
    @Override
    public void move() {
        System.out.println("Inside Truck::move() method.");
    }
}
```

Code Snippet 20 creates a factory to create object of concrete class based on the information provided.

Code Snippet 20:

```
class VehicleFactory {
    //use getVehicle method to get object of type Vehicle
    public Vehicle getVehicle(String vehicleType) {
        if(vehicleType == null){
            return null;
        }
        if(vehicleType.equalsIgnoreCase("Car")){
            return new Car();
        } else if(vehicleType.equalsIgnoreCase("Truck")){
            return new Truck();
        }
        return null;
    }
}
```

Code Snippet 21 uses the factory pattern to get objects of concrete classes by passing the **Vehicle** type information.

Code Snippet 21:

```
public class FactoryPatternExample {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        VehicleFactory vehicleFactory = new VehicleFactory();
        //get an object of Car and call its move method.
        Vehicle carObj = vehicleFactory.getVehicle("Car");
        //call move method of Car
        carObj.move();
        //get an object of Truck and call its move method.
    }
}
```

```

Vehicle truckObj = vehicleFactory.getVehicle("Truck");
//call move method of truck
truckObj.move();
}
}

```

In the code, the correct object is created and based on the type of object, the appropriate **move()** method is invoked.

Figure 11.10 depicts the factory pattern diagram.

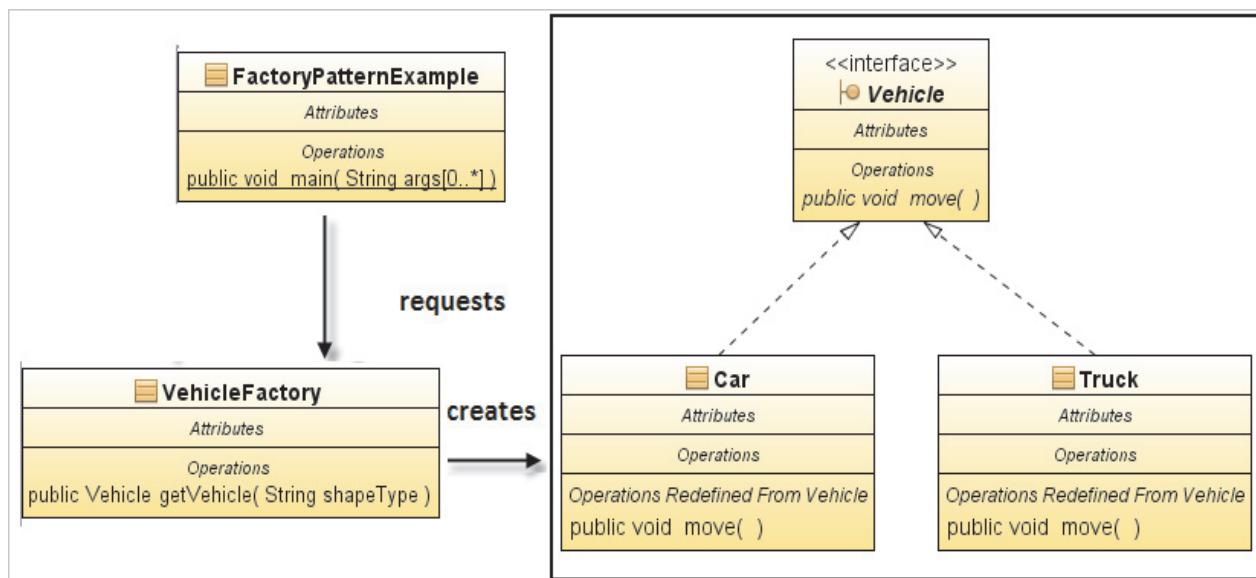


Figure 11.10: Factory Design Pattern

Figure 11.11 displays the output.

```

run:
Inside Car::move() method.
Inside Truck::move() method.
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 11.11: Factory Design Pattern - Output

11.5.9 The Observer Pattern

The Observer pattern helps to observe the behaviour of objects such as change in state or change in property. Consider an example of an online shopping store. The items that you are looking for are currently unavailable. You want to be notified whenever the items become available. For this, the status of those items need to be observed on a regular basis and the notification should be sent when the status

changes. This can be implemented using the Observer pattern.

In the observer pattern, an object called the subject maintains a collection of objects called observers. Whenever the subject changes, it notifies the observers. Observers can be added or removed from the collection of observers in the subject. The status change for the subject can be passed to the observers so that the observers will reflect this change by changing their own state.

Thus, the Observer pattern includes the following components:

→ **Subject:**

- A collection of observers
- A subject knows its observers
- It provides an interface for attaching and detaching to/from the observer object during runtime
- It can have any number of observers

→ **Observer:**

- It provides an update interface to receive signal from subject

→ **ConcreteSubject:**

- It stores and indicates status to the ConcreteObserver
- It also sends notifications

→ **ConcreteObserver:**

- Preserves a reference to an object of ConcreteSubject type
- Preserves observer state
- Implements the update tasks

Java offers built-in APIs to implement the Observer pattern in your programs. The `java.util.Observable` and `java.util.Observer` interfaces are part of these APIs.

Using these interfaces, the Item and Customer classes are created as shown in Code Snippets 22 and 23.

Code Snippet 22:

```
public class Item extends Observable{
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    private String itemName;
    private String itemType;
    String availability;
```

```
public Item(String itemName, String itemType, String availability)
{
    super();
    this.itemName = itemName;
    this.itemType = itemType;
    this.availability = availability;
}

public ArrayList<Observer> getObservers() {
    return observers;
}

public void setObservers(ArrayList<Observer> observers) {
    this.observers = observers;
}

public String getItemName() {
    return itemName;
}

public void setItemName(String itemName) {
    this.itemName = itemName;
}

public String getItemType() {
    return itemType;
}

public void setItemType(String itemType) {
    this.itemType = itemType;
}

public String getAvailability() {
    return availability;
}
```

```

public void setAvailability(String availability) {
    if(! (this.availability.equalsIgnoreCase(availability)))
    {
        this.availability = availability;
        setChanged();
        notifyObservers(this,availability);
    }
}

public void notifyObservers(Observable observable, String
                           availability)
{
    System.out.println("Notifying to all the subscribers when item
                       became available");
    for (Observer ob : observers) {
        ob.update(observable, this.availability);
    }
}

public void registerObserver(Observer observer) {
    observers.add(observer);
}

public void removeObserver(Observer observer) {
    observers.remove(observer);
}
}

```

Code Snippet 23:

```

public class Customer implements Observer{
    String customerName;
    public Customer(String customerName) {
        this.customerName = customerName;
    }
    public String getCustomerName() {

```

```

        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public void update(Observable arg0, Object arg1) {
        System.out.println("Hello "+customerName+", Item is now "+arg1+
            " on our online shopping store");
    }
}

```

Code Snippet 24 creates two **Customer** instances and an **Item** instance. Then, it registers the observers for the customers on that item. Whenever the item becomes available, the two customers will be notified of its availability.

Code Snippet 24:

```

public class ObserverPatternMain {

    public static void main(String[] args) {
        Customer cust1=new Customer("Sarah Heyward");
        Customer cust2=new Customer("Jim Gordon");
        Item newitem=new Item("EBook Reader", "Device", "Not
available");
        newitem.registerObserver(cust1);
        newitem.registerObserver(cust2);
        //Now the item becomes available
        newitem.setAvailability("Available");
    }
}

```

11.6 Delegation

Besides the standard design patterns, one can also use the delegation design pattern. In Java, delegation means using an object of another class as an instance variable, and forwarding messages to the instance. Therefore, delegation is a relationship between objects. Here, one object forwards method calls to another object, which is called its delegate.

Delegation is different from inheritance. Unlike inheritance, delegation does not create a super class. In this case, the instance is that of a known class. In addition, delegation does not force to accept all the methods of the super class. Delegation supports code reusability and provides run-time flexibility.

Note- Run-time flexibility means that the delegate can be easily changed at run-time.

Code Snippet 25 displays the use of delegation using a real-world scenario.

Code Snippet 25:

```
public interface Employee() {
    public Result sendMail();
}

public class Secretary() implements Employee {
    public Result sendMail() {
        Result myResult = new Result();
        return myResult;
    }
}

public class Manager() implements Employee {
    private Secretary secretary;
    public Result sendMail() {
        return secretary.sendMail();
    }
}
```

In Code Snippet 25, the **Manager** instance forwards the task of sending mail to the **Secretary** instance who in turn forwards the request to the employee. On a casual observation, it may seem that the manager is sending the mail but in reality, it is the employee doing the task. Thus, the task request has been delegated.

11.7 Composition and Aggregation

Composition is-a relationship
Aggregation has-a relationship

Composition refers to the process of **composing a class from references to other objects**. This way, references to the main objects are fields of the containing object. Composition forms the building blocks for data structures. Programmers can **use object composition to create more complex objects**. Aggregation is a similar concept with some differences. In **aggregation**, one class owns another class. In **composition**, when the **owning object is destroyed**, so are the objects within it but **in aggregation, this is not true**.

For example, an organization comprises various departments and each department has a number of managers. If the organization shuts down, these departments will no longer exist, but the managers will continue to exist. Therefore, an organization can be seen as a composition of departments, whereas departments have an aggregation of managers. In addition, a manager could work in more than one department (if he/she is handling multiple responsibilities), but one department cannot exist in more than one organization.

Composition and aggregation are **design concepts** and **not actual patterns**.

Code Snippet 26 demonstrates a minimal example of composition.

Code Snippet 26:

```
// Composition
class House
{
    // House has door.
    // Door is built when House is built,
    // it is destroyed when House is destroyed.
    private Door dr;
}
```

To implement object composition, perform the following steps:

1. Create a class with reference to other classes.
2. Add the same signature methods that forward to the referenced object.

Consider an example of a student attending a course. The student ‘has a’ course. The composition for the **Student** and **Course** classes is depicted in Code Snippets 27 and 28.

Code Snippet 27:

```
package composition;
public class Course {
```

```

private String title;
private long score;
private int id;

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public long getScore() {
    return score;
}

public void setScore(long score) {
    this.score = score;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}
}

```

The **Course** class is then used in the **Student** class as shown in Code Snippet 28.

Code Snippet 28:

```

package composition;

public class Student {
    //composition has-a relationship
    private Course course;
    public Student(){
}

```

```
this.course=new Course();
course.setScore(1000);
}

public long getScore() {
    return course.getScore();
}

/**
 * @param args the command line arguments
 */

public static void main(String[] args)
{
    Student p = new Student();
    System.out.println(p.getScore());
}
```

11.8 Check Your Progress

1. Identify the component required to create a hash function.

(A)	The instanceof operator
(B)	String
(C)	Non zero integer constant
(D)	Design pattern

(A)	A	(C)	C
(B)	B	(D)	D

2. Which of the following statements are true for the logical equality?

(A)	When physical memory locations of the two strings are same, this is called logical equality.
(B)	When data of the objects are same, it is called logical equality.
(C)	The equality operator (==) is used to check for logical equality.
(D)	The equals() method that is used to check for logical equality is inherited from the Object class.

(A)	A, D	(C)	A, C
(B)	B, D	(D)	D

3. Which of the statements are true for Factory design pattern?

(A)	Prevents the application from being tightly coupled to a specific DAO implementation
(B)	Does not depend on concrete DAO classes
(C)	Uses a common interface to refer to the newly created object
(D)	Not used to implement the DAO pattern

(A)	A	(C)	A, C
(B)	B, C	(D)	B, D

4. Which one of the following design patterns provides complete information on class implementations that can be instantiated only once?

(A)	Factory
(B)	Singleton
(C)	Adapter
(D)	Proxy

(A)	A	(C)	C
(B)	B	(D)	D

11.8.1 Answers

1.	C
2.	B
3.	C
4.	B

```
g package;
String pac
String pr
ids string
y catch
```

Summary

- The development of application software is performed using a programming language that enforces a particular style of programming, also referred to as programming paradigm.
- In structured programming paradigm, the application development is decomposed into a hierarchy of subprograms.
- In object-oriented programming paradigm, applications are designed around data, rather than focusing only on the functionalities.
- The main building blocks of an OOP language are classes and objects. An object represents a real-world entity and a class is a conceptual model.
- Java is an OOP language as well a platform used for developing applications that can be executed on different platforms. Java platform is a software-only platform that runs on top of the other hardware-based platforms.
- The editions of Java platform are Java SE, Java EE, and Java ME.
- The components of Java SE platform are JDK and JRE. JRE provides JVM and Java libraries that are used to run a Java program. JDK includes the necessary development tools, runtime environment, and APIs for creating Java programs.



Need
HELP
on a topic? = **FAQs**



www.onlinevarsity.com

Session 12

Internationalization and Localization

Welcome to the Session, **Internationalization and Localization**.

This session describes the internationalization and localization process that makes an application serve users in multiple different languages and are suitable for global market. It describes various methods that Java application can use to handle different languages, number formats, and so on. Internationalized applications require meticulous planning, failing which re-engineering of the application can be costly.

In this Session, you will learn to:

- ➔ Describe internationalization
- ➔ Describe localization
- ➔ Describe the Unicode character encoding
- ➔ Explain the internationalization process
- ➔ Define the internationalization elements



12.1 Introduction

With the advent of the Internet, globalization of software products has become an imminent requirement.

The main problems faced in globalization of software products are as follows:

- Not all countries across the world speak or understand English language.
- Symbols for currency vary across countries.
- Date and Time are represented differently in some countries.
- Spelling also varies amongst some countries.

Two possible options for solving the problems faced are as follows:

- **Develop the entire product in the desired language** - This option will mean repetition of coding work. It is a time-consuming process and not an acceptable solution. Development cost will be much higher than the one time cost.
- **Translate the entire product in the desired language** - Successful translation of the source files is very difficult. The menus, labels and messages of most GUI components are hard-coded in the source code. It is not likely that a developer or a translator will have linguistic as well as coding skills.

Thus, when the input and output operations of an application is made specific to different locations and user preferences, users around the world can use it with ease. This can be achieved using the processes called internationalization and localization. The adaptation is done with extreme ease because there are no coding changes required.

12.2 Internationalization

To make an application accessible to the international market, it should be ensured that the input and output operations are specific to different locations and user preferences. The process of designing such an application is called internationalization. Note that the process occurs without any engineering changes.

Internationalization is commonly referred to as **i18n**. 18 in i18n refer to the 18 characters between the first letter i and the last letter n.

Java includes a built-in support to internationalize applications. This is called Java internationalization.

12.3 Localization

While internationalization deals with different locations and user preferences, localization deals with a specific region or language. In localization, an application is adapted to a specific region or language.

Locale-specific components are added and text is translated in the localization process. A locale represents a particular language and country.

Localization is commonly referred as I10n. 10 in I10n refers to the 10 letters between the first letter I and the last letter n.

Primarily, in localization, the user interface elements and documentation are translated. Changes related to dates, currency, and so on are also taken care of. In addition, culturally sensitive data, such as images, are localized. If an application is internationalized in an efficient manner, then it will be easier to localize it for a particular language and character encoding scheme.

12.4 Benefits of I18N and L10N

An internationalized application includes the following benefits:

- **No Recompilation of New Languages:** New languages are supported without recompilation.
- **Same Executable File:** The localized data needs to be incorporated in the application and the same executable runs worldwide.
- **Dynamic Retrieval of Textual Elements:** Textual elements such as the GUI component labels are stored outside the source code. They are not hardcoded in the program. Therefore, these elements can be retrieved dynamically.
- **Conformation to the End User's Region and Language:** Region specific information such as currencies, numbers, date and time follow the specified format of the end user's region and language.
- **Easy Localization:** The application can be easily and quickly localized.

12.5 ISO Codes

In the internationalization and localization process, a language is represented using the alpha-2 or alpha-3 ISO 639 code, such as es that represents Spanish. The code is always represented in lower case letters.

A country is represented using the ISO 3166 alpha-2 code or UN M.49 numeric area code. It is always represented in upper case. For example, ES represents Spain. If an application is well internationalized, it is easy to localize it for a character encoding scheme.

Code Snippet 1 illustrates the use of Japanese language for displaying a message.

Code Snippet 1:

```
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {
```

```
/**  
 * @param args the command line arguments  
 */  
  
public static void main(String[] args) {  
    // TODO code application logic here  
    String language;  
    String country;  
  
    if (args.length != 2) {  
        language = new String("en");  
        country = new String("US");  
    } else {  
        language = new String(args[0]);  
        country = new String(args[1]);  
    }  
  
    Locale currentLocale;  
    ResourceBundle messages;  
  
    currentLocale = new Locale(language, country);  
  
    messages = ResourceBundle.getBundle("internationalApplication/  
    MessagesBundle", currentLocale);  
    System.out.println(messages.getString("greetings"));  
    System.out.println(messages.getString("inquiry"));  
    System.out.println(messages.getString("farewell"));  
}  
}
```

In the code, two arguments are accepted to represent country and language. Depending on the arguments passed during execution of the program the message corresponding to that country and language is displayed. For this, five properties file have been created.

The content of the five properties files are as follows:

→ MessagesBundle.properties

```
greetings = Hello.  
farewell = Goodbye.  
inquiry = How are you?
```

→ MessagesBundle_de_DE.properties

```
greetings = Hallo.  
farewell = Tschüß.  
inquiry = Wie geht's?
```

→ MessagesBundle_en_US.properties

```
greetings = Hello.  
farewell = Goodbye.  
inquiry = How are you?
```

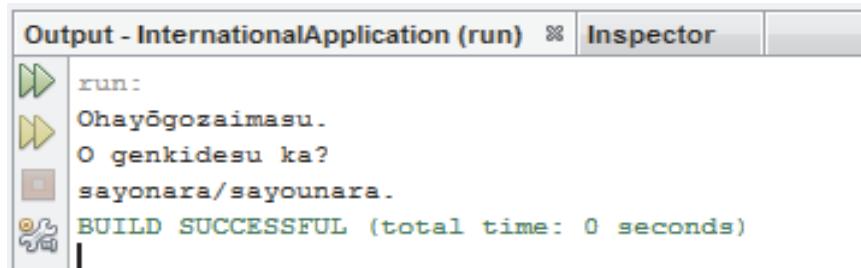
→ MessagesBundle_fr_FR.properties

```
greetings = Bonjour.  
farewell = Au revoir.  
inquiry = Comment allez-vous?
```

→ MessagesBundle_ja_JP.properties

```
greetings = Ohayōgozaimasu.  
farewell = sayonara/sayounara.  
inquiry = O genkidesuka?
```

Figure 12.1 displays the output in Japanese language.



```
Output - InternationalApplication (run) × Inspector
run:  
Ohayōgozaimasu.  
O genkidesu ka?  
sayonara/sayounara.  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 12.1: InternationalApplication - Output

12.6 Unicode

This is a computing industry standard. It is used to uniquely encode characters for various languages in the world using hexadecimal values. In other words, Unicode provides a unique number for every character irrespective of platform, program, or language.

Before Unicode was invented, there were several different encoding systems for assigning these numbers. No single encoding system contained enough number of characters to address international languages. These various encoding systems conflict with each other, that is, two encoding systems can have the same number for two different characters, or can use different numbers for the same character.

The Unicode standard was first designed using 16 bits to encode characters. 16-bit encoding supports 2¹⁶ (65,536) characters where in the hexadecimal they ranged from 0x0000 to 0xFFFF. This was insufficient to define all characters in world languages. So, the Unicode standard was extended to 0x10FFFF hexadecimal values. This supports over one million characters.

However, in Java, it was not easy to change 16 bits characters to 32 bits characters. A scheme was developed to resolve the issue.

Note - Java uses Unicode as its native character encoding.

However, Java programs still need to handle characters in other encoding systems. The `String` class can be used to convert standard encoding systems to and from the Unicode system.

To indicate Unicode characters that cannot be represented in ASCII, such as ö, you use the \uXXXX escape sequence. Each X in the escape sequence is a hexadecimal digit.

The following list defines the terminologies used in the Unicode character encoding:

- **Character:** This represents the minimal unit of text that has semantic value.
- **Character Set:** This represents set of characters that can be used by many languages. For example, the Latin character set is used by English and certain European languages.
- **Coded Character:** This is a character set. Each character in the set is assigned a unique number.
- **Code Point:** This is the value that is used in a coded character set. A code point is a 32-bit `int` data type. Here, the upper 11 bits are 0 and the lower 21 bits represent a valid code point value.
- **Code Unit:** This is a 16-bit `char` value.
- **Supplementary Characters:** These are the characters that range from U+10000 to U+10FFFF. Supplementary characters are represented by a pair of code point values called surrogates that support the characters without changing the `char` primitive data type. Surrogates also provide compatibility with earlier Java programs.
- **Basic Multilingual Plane (BMP):** These are the set of characters from U+0000 to U+FFFF.

12.6.1 Unicode Character Encoding

Consider the following points for Unicode character encoding:

- The hexadecimal value is prefixed with the string U+.
- The valid code point range for the Unicode standard is U+0000 to U+10FFFF.

Table 12.1 shows code point values for certain characters.

Character	Unicode Code Point	Glyph
Latin A	U+0041	A
Latin sharp S	U+00DF	B

Table 12.1: Code Point Values for Certain Characters

12.7 Internationalization Process

If the internationalized source code is observed, notice that the hardcoded English messages are removed. Because the messages are no longer hardcoded and the language code is specified at run time, so that the same executable can be distributed worldwide. No recompilation is required for localization. For internationalization process the steps to be followed are as follows:

- Creating the Properties files
- Defining the Locale
- Creating a ResourceBundle
- Fetching the text from the ResourceBundle class

12.7.1 Creating the Properties Files

A properties file stores information about the characteristics of a program or environment. A properties file is in plain-text format. It can be created with any text editor.

In the following example, the properties files store the text that needs to be translated. Note that before internationalization, the original version of text was hardcoded in the `System.out.println` statements. The default properties file, `MessagesBundle.properties`, includes the following lines:

```
greetings=Hello
farewell=Goodbye
inquiry=How are you?
```

Since the messages are in the properties file, it can be translated into various languages. No changes to the source code are required.

To translate the message in French, the French translator creates a properties file called `MessagesBundle_fr_FR.properties`, which contains the following lines:

```
greetings = Bonjour.  
farewell = Au revoir.  
inquiry = Comment allez-vous?
```

Notice that the values to the right side of the equal sign are translated. The keys on the left side are not changed. These keys must not change because they are referenced when the program fetches the translated text.

The name of the properties file is important. For example, the name `MessagesBundle_fr_FR.properties` file contains the `fr` language code and the `FR` country code. These codes are also used when creating a `Locale` object.

12.7.2 Defining the Locale

The `Locale` object identifies a particular language and country. A `Locale` is simply an identifier for a particular combination of language and region.

A `java.util.Locale` class object represents a specific geographical, political, or cultural region. Any operation that requires a locale to perform its task is said to be locale-sensitive. These operations use the `Locale` object to tailor information for the user.

For example, displaying a number is a locale-sensitive operation. The number should be formatted according to the customs and conventions of a user's native country, region, or culture.

A `Locale` object is created using the following constructors:

- `public Locale(String language, String country)`- This creates a `Locale` object with the specified language and country. Consider the following syntax:

Syntax :

```
public Locale(String language, String country)
```

where,

`language` - is the language code consisting of two letters in lower case.

`country` - is the country code consisting of two letters in upper case.

- `public Locale(String language)`- This creates a locale object with the specified language. Consider the following syntax.

Syntax:

```
public Locale(String language)
```

where,

language - is the language code consisting of two letters in lower case.

Code Snippet 2 defines a `Locale` for which the language is English and the country is the United States.

Code Snippet 2:

```
aLocale = new Locale("en", "US");
```

Code Snippet 3 creates `Locale` objects for the French language for the countries Canada and France.

Code Snippet 3:

```
caLocale = new Locale("fr", "CA");
frLocale = new Locale("fr", "FR");
```

The program will be flexible when the program accepts them from the command line at run time, instead of using hardcoded language and country codes. Code Snippet 4 demonstrates how to accept the language and country code from command line:

Code Snippet 4:

```
String language = new String(args[0]);
String country = new String(args[1]);
currentLocale = new Locale(language, country);
```

`Locale` objects are only identifiers. After defining a `Locale`, the next step is to pass it to other objects that perform useful tasks, such as formatting dates and numbers. These objects are locale-sensitive because their behavior varies according to `Locale`. A `ResourceBundle` is an example of a locale-sensitive object.

The following section describes certain important methods of the `Locale` class:

- **public static Locale getDefault():** This method gets the default `Locale` for this instance of the Java Virtual Machine (JVM). Here, `Locale` is the return type. In other words, the method returns an object of the class `Locale`.
- **Public final String getDisplayCountry():** This method returns the name of the country for the current `Locale`, which is appropriate for display to the user. Here, `String` is the return type. In other words, the method returns a `String` representing the name of the country.
- **public final String getDisplayLanguage():** This method returns the name of the language for the current `Locale`, which is appropriate for display to the user. For example, if the default locale is `fr_FR`, the method returns French. Here, `String` is the return type. In other words, the method returns a `String` representing the name of the language.

12.7.3 Creating a ResourceBundle

ResourceBundle objects contain locale-specific objects. These objects are used to isolate locale-sensitive data, such as translatable text.

The ResourceBundle class is used to retrieve locale-specific information from the properties file.

This information allows a user to write applications that can be:

- Localized or translated into different languages.
- Handled for multiple locales at the same time.
- Supported for more locales later.

The ResourceBundle class has a static and final method called `getBundle()` that helps to retrieve a ResourceBundle instance.

The ResourceBundle `getBundle(String, Locale)` method helps to retrieve locale-specific information from a given properties file and takes two arguments, a String and an object of Locale class. The object of ResourceBundle class is initialized with a valid language and country matching the available properties file.

To create the ResourceBundle, consider the code given in Code Snippet 5.

Code Snippet 5:

```
messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
```

The arguments passed to the `getBundle()` method identify the properties file that will be accessed. The first argument, MessagesBundle, refers to the following family of properties files:

- MessagesBundle_en_US.properties
- MessagesBundle_fr_FR.properties
- MessagesBundle_de_DE.properties
- MessagesBundle_ja_JP.properties

The `currentLocale`, which is the second argument of `getBundle()` method, specifies the selected MessagesBundle files. When the Locale was created, the language code and the country code were passed to its constructor. Note that the language and country codes follow MessagesBundle in the names of the properties files.

To retrieve the locale-specific data from the properties file, the ResourceBundle class object should first be created and then the following methods should be invoked:

- `public final String getString(String key):` The `getString()` method takes a string as an argument that specifies the key from the properties file whose value is to be retrieved. The method returns a string, which represents the value from the properties file associated with the key. In the method, `key` is a string representing an available key from the properties file. The return value is a `String` representing the value associated with the key.
- `public abstract Enumeration<String>getKeys():` The `getKeys()` method returns an enumeration object representing all the available keys in the properties file. In the method, `Enumeration` is an `Enumeration` object representing all the available keys in the properties file.

After invoking all the required methods, the translated messages can be retrieved from the `ResourceBundle` class.

12.7.4 Fetching the Text from the ResourceBundle Class

The properties files contain key-value pairs. The values consists of the translated text that the program will display. The keys are specified when fetching the translated messages from the `ResourceBundle` with the `getString()` method. For example, to retrieve the message identified by the `greetings` key, the `getString()` method is invoked. Code Snippet 6 illustrates how to retrieve the value from the key-value pair using the `getString()` method.

Code Snippet 6:

```
String msg1 = messages.getString("greetings");
```

The sample program uses the key `greetings` because it reflects the content of the message. The key is hardcoded in the program and it must be present in the properties files. If the translators accidentally modify the keys in the properties files, then the `getString()` method will be unable to locate the messages.

12.8 Internationalization Elements

There are various elements that vary with culture, region, and language. Therefore, it should be ensured that all such elements are internationalized.

12.8.1 Component Captions

These refer to the GUI component captions such as text, date, and numerals. These GUI component captions should be localized because their usage vary with language, culture, and region.

Formatting the captions of the GUI components ensures that the look and feel of the application is in a locale-sensitive manner. The code that displays the GUI is locale-independent. There is no the need to write formatting routines for specific locales.

12.8.2 Numbers, Currencies, and Percentages

The format of numbers, currencies, and percentages vary with culture, region, and language.

Hence, it is necessary to format them before they are displayed. For example, the number 12345678 should be formatted and displayed as 12,345,678 in the US and 12.345.678 in Germany.

Similarly, the currency symbols and methods of displaying the percentage factor also vary with region and language.

Formatting is required to make an internationalized application, independent of local conventions with regards to decimal-point, thousands-separators, and percentage representation.

The `NumberFormat` class is used to create locale-specific formats for numbers, currencies, and percentages.

→ Number

The `NumberFormat` class has a static method `getNumberInstance()`. The `getNumberInstance()` returns an instance of a `NumberFormat` class initialized to default or specified locale.

The `format()` method of the `NumberFormat` class should be invoked next, where the number to be formatted is passed as an argument. The argument can be a primitive or a wrapper class object.

The syntaxes for some of the methods that are used for formatting numbers are as follows:

- `public static final NumberFormat getNumberInstance():` Here, `NumberFormat` is the return type. This method returns an object of the class `NumberFormat`.
- `public final String format(double number):` Here, `number` is the number to be formatted. `String` is the return type and represents the formatted text.
- `public static NumberFormat getNumberInstance(Locale inLocale):` Here, `NumberFormat` is the return type. This method returns an object of the class `NumberFormat`. The argument `inLocale` is an object of the class `Locale`.

Code Snippet 7 shows how to create locale-specific format of number for the country Japan.

Code Snippet 7:

```
import java.text.NumberFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {

    static public void printValue(Locale currentLocale) {
        Integer value = new Integer(123456);
        Double amt = new Double(345987.246);
```

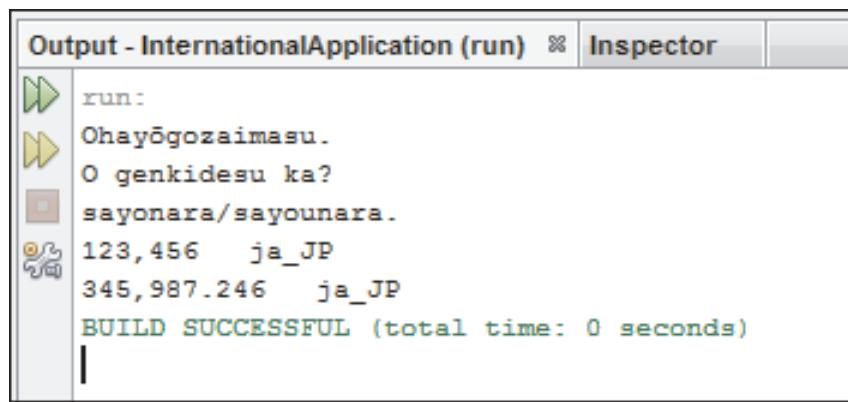
```
NumberFormatnumFormatObj;  
String valueDisplay;  
String amtDisplay;  
numFormatObj = NumberFormat.getNumberInstance(currentLocale);  
valueDisplay = numFormatObj.format(value);  
amtDisplay = numFormatObj.format(amt);  
System.out.println(valueDisplay + " " + currentLocale.toString());  
System.out.println(amtDisplay + " " + currentLocale.toString());  
}  
  
/**  
 * @param args the command line arguments  
 */  
public static void main(String[] args) {  
    // TODO code application logic here  
    String language;  
    String country;  
    if (args.length != 2) {  
        language = new String("en");  
        country = new String("US");  
    } else {  
        language = new String(args[0]);  
        country = new String(args[1]);  
    }  
    Locale currentLocale;  
    ResourceBundle messages;  
    currentLocale = new Locale(language, country);  
    messages =  
        ResourceBundle.getBundle("internationalApplication/MessagesBundle",  
        currentLocale);
```

```

System.out.println(messages.getString("greetings"));
System.out.println(messages.getString("inquiry"));
System.out.println(messages.getString("farewell"));
printValue(currentLocale);
}
}

```

Figure 12.2 displays the output.



The screenshot shows the 'Output' tab in the Android Studio interface. The title bar says 'Output - InternationalApplication (run)'. The main area contains the following text:

```

run:
Ohayōgozaimasu.
O genkidesu ka?
sayonara/sayounara.
123,456 ja_JP
345,987.246 ja_JP
BUILD SUCCESSFUL (total time: 0 seconds)
|
```

Figure 12.2: Number in Japanese Format

→ Currencies

The `NumberFormat` class has a static method `getCurrencyInstance()` which takes an instance of `Locale` class as an argument. The `getCurrencyInstance()` method returns an instance of a `NumberFormat` class initialized for the specified locale.

The syntaxes for some of the methods to format currencies are as follows:

- `public final String format(double currency):` Here, `currency` is the currency to be formatted. `String` is the return type and represents the formatted text.
- `public static final NumberFormat getCurrencyInstance():` Here, `NumberFormat` is the return type, that is, this method returns an object of the class `NumberFormat`.
- `public static NumberFormat getCurrencyInstance(Locale inLocale):` Here, `NumberFormat` is the return type, that is, this method returns an object of the class `NumberFormat`. `inLocale` is the specified `Locale`.

Code Snippet 8 shows how to create locale-specific format of currency for the country, France.

Code Snippet 8:

```

NumberFormat currencyFormatter;
String strCurrency;

// Creates a Locale object with language as French and country
// as France
Locale locale=new Locale("fr", "FR");

// Creates an object of a wrapper class Double
Double currency=new Double(123456.78);

// Retrieves the CurrencyFormatter instance
currencyFormatter=NumberFormat.getInstance(locale);

// Formats the currency
strCurrency=currencyFormatter.format(currency);

```

→ Percentages

This class has a static method `getPercentInstance()`, which takes an instance of the `Locale` class as an argument. The `getPercentInstance()` method returns an instance of the `NumberFormat` class initialized to the specified locale.

The syntaxes for some of the methods to format percentages are as follows:

- `public final String format(double percent):` Here, `percent` is the percentage to be formatted. `String` is the return type and represents the formatted text.
- `public static final NumberFormat getPercentInstance():` `NumberFormat` is the return type, that is, this method returns an object of the class `NumberFormat`.
- `public static NumberFormat getPercentInstance(Locale inLocale):` Here, `NumberFormat` is the return type, that is, this method returns an object of the class `NumberFormat`. `inLocale` is the specified `Locale`.

Code Snippet 9 shows how to create locale-specific format of percentages for the country, France.

Code Snippet 9:

```

NumberFormatpercentFormatter;
String strPercent;

// Creates a Localeobject with language as French and country
// as France
Locale locale = new Locale("fr", "FR");
// Creates an object of a wrapper class Double
Double percent = new Double(123456.78);

// Retrieves the percentFormatter instance
percentFormatter = NumberFormat.getInstance(locale);

// Formats the percent figure
strPercent = percentFormatter.format(percent);

```

12.8.3 Date and Times

The date and time format should conform to the conventions of the end user's locale. The date and time format varies with culture, region, and language. Hence, it is necessary to format them before they are displayed.

For example, in German, the date can be represented as 20.04.07, whereas in US it is represented as 04/20/07.

Java provides the `java.text.DateFormat` and `java.text.SimpleDateFormat` class to format date and time.

The `DateFormat` class is used to create locale-specific formats for date. Next, the `format()` method of the `NumberFormat` class is also invoked. The date to be formatted is passed as an argument.

The `DateFormat getDateInstance(style, locale)` method returns an instance of the class `DateFormat` for the specified style and locale. Consider the following syntax.

Syntax:

```
public static final DateFormat getDateInstance(int style, Locale locale)
where,
```

`style` - is an integer and specifies the style of the date. Valid values are `DateFormat.LONG`, `DateFormat.SHORT`, `DateFormat.MEDIUM`.

locale - is an object of the `Locale` class, and specifies the format of the locale.

`DateFormat` object includes a number of constants such as:

- ➔ SHORT : Is completely numeric such as 12.13.45 or 4 :30 pm
- ➔ MEDIUM : Is longer, such as Dec 25, 1945
- ➔ LONG : Is longer such as December 25, 1945
- ➔ FULL : Represents a complete specification such as Tuesday, April 12, 1945 AD

Code Snippet 10 demonstrates how to retrieve a `DateFormat` object and display the date in Japanese format.

Code Snippet 10:

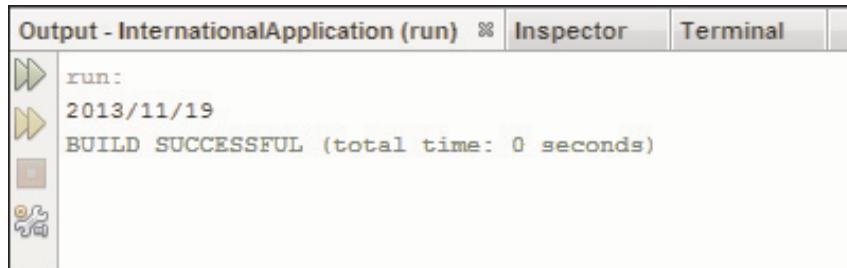
```
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

public class DateInternationalApplication {

    public static void main(String[] args) {
        Date today;
        String strDate;
        DateFormat dateFormatter;
        Locale locale = new Locale("ja", "JP");
        dateFormatter = DateFormat.getDateInstance(DateFormat.MEDIUM, locale);
        today = new Date();
        strDate = dateFormatter.format(today);
        System.out.println(strDate);

    }
}
```

Figure 12.3 displays the output.



```
run:  
2013/11/19  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 12.3: Output - Date in Japanese Format

12.8.4 Messages

Displaying messages such as status and error messages are an integral part of any software.

If the messages are predefined, such as “Your License has expired”, they can be easily translated into various languages. However, if the messages contain variable data, it is difficult to create grammatically correct translations for all languages.

For example, consider the following message in English:

“On 06/03/2007 we detected 10 viruses”.

In French it is translated as:

“Sur 06/03/2007 nous avons détecté le virus 10”.

In German it is translated as:

“Auf 06/03/2007 ermittelten wir Virus 10”.

The position of verbs and the variable data varies in different languages.

It is not always possible to create a grammatically correct sentence with concatenation of phrases and variables. The approach of concatenation works fine in English, but it does not work for languages in which the verb appears at the end of the sentence. If the word order in a message is hard-coded, it is impossible to create grammatically correct translations for all languages. The solution is to use the `MessageFormat` class to create a compound message.

To use the `MessageFormat` class, perform the following steps:

1. Identify the variables in the message. To do so, write down the message, and identify all the variable parts of the message.

For example consider the following message:

At 6:41 PM on April 25, 2007, we detected 7 virus on the disk D:

In the message, the four variable parts are underlined.

2. Create a template. A template is a string, which contains the fixed part of the message and the variable parts. The variable parts are encoded in {} with an argument number, for example {0}, {1},

and so on.

Each argument number should match with an index of an element in an `Object` array containing argument values.

3. Create an `Object` array for variable arguments. For each variable part in the template, a value is required to be replaced. These values are initialized in an `Object` array. The elements in the `Object` array can be constructed using the constructors. If an element in the array requires translation, it should be fetched from the Resource Bundle with the `getString()` method.
4. Create a `MessageFormat` instance and set the desired locale. The locale is important because the message might contain date and numeric values, which are required to be translated.
5. Apply and format the pattern. To do so, fetch the pattern string from the Resource Bundle with the `getString()` method.

The `MessageFormat` class has a method `applyPattern()` to apply the pattern to the `MessageFormat` instance. Once the pattern is applied to the `MessageFormat` instance, invoke the `format()` method.

Code Snippet 11 when executed will display the message in Danish using `MessageFormatter` class.

Code Snippet 11:

```
import java.text.MessageFormat;
import java.util.Date;
import java.util.Locale;
import java.util.ResourceBundle;
public class MessageFormatterInternationalApplication {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        String template = "At {2,time,short} on {2,date,long}, we detected
{1,number,integer} virus on the disk {0}";
        MessageFormat formatter = new MessageFormat("");
        String language;
        String country;
```

```

if (args.length != 2) {
    language = new String("en");
    country = new String("US");
} else {
    language = new String(args[0]);
    country = new String(args[1]);
}

Locale currentLocale;
currentLocale = new Locale(language, country);
formatter.setLocale(currentLocale);

ResourceBundle messages = ResourceBundle.getBundle("messageformatterinternationalapplication/MessageFormatBundle", currentLocale);
Object[] messageArguments = {messages.getString("disk"), new Integer(7), new Date()};
formatter.applyPattern(messages.getString("template"));
String output = formatter.format(messageArguments);
System.out.println(output);
}
}

```

The following are the three properties file required by the code.

→ MessageFormatBundle.properties

```

disk = D:
template = At {2,time,short} on {2,date,long}, we detected
           {1,number,integer} virus on the disk {0}

```

→ MessageFormatBundle_fr_FR.properties

```

disk = D:
template = À {2,time,short} {2,date,long}, nous avons
           détecté le virus {1,number,integer} sur le disque {0}

```

```
→ MessageFormatBundle _ de _ DE.properties  
disk = D:  
template = Um {2,time,short} an {2,date,long}, ermittelten  
wir Virus {1,number,integer} auf der Scheibe {0}
```

Figure 12.4 displays the output for Code Snippet 11.



```
Output - MessageFormatterInternationalApplication (run) ✘ Inspector  
run:  
Um 14:05 an 11. November 2013, ermittelten wir Virus 7 auf der Scheibe D:  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 12.4: Output - Message Displayed in Danish

12.9 Check Your Progress

1. Which of the following option is true for internationalization?

(A)	GUI component labels are hard coded.	(C)	A language is represented using the alpha-2 or alpha-3 ISO 639 code.
(B)	New languages are supported after recompilation.	(D)	Different executables runs worldwide.

2. The Unicode standard _____ and _____.

(A)	Supports over one million characters.
(B)	Supports only high surrogates.
(C)	Includes the hexadecimal value prefixed with the string U+.
(D)	Uses the <code>codePointCount</code> method to check the validity of a parameter.

(A)	A, B	(C)	B, D
(B)	C, D	(D)	A, C

3. Which of the following option represents the default properties file?

(A)	System.out.println	(C)	MessagesBundle.properties
(B)	MessagesBundle	(D)	System.out

4. A _____ is an identifier for a particular combination of language and region.

(A)	Language	(C)	Local
(B)	util.Locale	(D)	Locale

5. Which of the following option specifies the next step to be followed after defining a locale?

(A)	Invoke the <code>getDisplayLanguage()</code> method	(C)	Fetch the translated messages
(B)	Pass it to <code> ResourceBundle</code> objects	(D)	Run the program from the command line

12.9.1 Answers

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

Summary

- In the internationalization process, the input and output operations of an application are specific to different locations and user preferences.
- Internationalization is commonly referred to as i18n.
- In localization, an application is adapted to a specific region or language.
- A locale represents a particular language and country. Localization is commonly referred as l10n.
- A language is represented using the alpha-2 or alpha-3 ISO 639 code in the internationalization and localization process.
- Unicode is a computing industry standard to uniquely encode characters for various languages in the world using hexadecimal values.
- No recompilation is required for localization.

ASK to Learn



What
Why
Where
Questions
When
How



Post your questions in the **ASK to LEARN** section
and we'll get back to you.

Session - 13

Advanced Concurrency and Parallelism

Welcome to the Session, Advanced Concurrency and Parallelism.

This session discusses advanced concurrency and parallelism features provided by Java 8.

In this Session, you will learn to:

- ➔ Explain the enhancements of `java.util.concurrent` package
- ➔ Describe atomic operations with the new set of classes of the `java.util.concurrent.atomic` package
- ➔ Explain the `StampedLock` class to implement locks
- ➔ Explain the new features of `ForkJoinPool`
- ➔ Define parallel streams
- ➔ Describe parallel sorting of arrays
- ➔ Identify recursive actions of the fork/join framework



13.1 Introduction

One major enhancement in Java 8 is the support for parallelization. By parallelizing workload, programmers can make applications run faster by efficiently using multi-core processors. However, parallelizing any type of workload comes with challenges. The primary challenge arises in the partitioning step. Ideally, a programmer would want to partition the workload in such a manner that every work piece completes execution exactly within the same amount of time. However, while writing code, programmers have to guess where the partition should be. Often, this results in some parts of the program taking longer to be processed than some other parts. As a result, the completed part needs to wait for the uncompleted ones, thereby defeating the whole purpose of parallelism. Such challenges are addressed by work stealing, which is a scheduling strategy for multithreaded programs.

Parallelism ensures that if threads of some parts of the program finish their jobs earlier than others, they take up work of the lagging ones to finish the overall job faster. However, work stealing itself comes with few challenges. Particularly, data integrity must be ensured so that different threads during the work stealing process does not perform dirty reads and writes. Although this challenge can be addressed using synchronization techniques, doing so might further slowdown the processing as synchronization comes with performance overheads. Therefore, work-stealing should be performed with minimal synchronization.

The Fork-Join framework introduced in Java 7 meets the requirements of work stealing through recursive job partitioning along with a double-ended queue (deque) structure for holding the tasks. At the API level, the `ForkJoinTask.join()` method allows a thread to avoid blocking itself and instead, ask the pool for any work it should do. In addition to efficient work stealing algorithm of the Fork-Join framework, Java 8 provides several enhancements in concurrency and parallelism features.

13.2 New Enhancements in the `java.util.concurrent` Package

Java 8 has made several enhancements in the `java.util.concurrent` package. `CompletableFuture`, `AsynchronousCompletionTask` is a new interface in the `java.util.concurrent` package that acts as a marker interface to identify asynchronous tasks that `async` methods generate. The marker interface is useful to monitor, debug, and track asynchronous operations. Another new interface added to the `java.util.concurrent` package is `CompletionStage<T>`. This interface represents a stage in an asynchronous computation process. Once the computation terminates, the stage completes. However, at times, the stage may in turn start one or more dependent stages represented by `CompletionStage<T>`.

Java 8 also introduces a new exception, `CompletionException` in the `java.util.concurrent` package. A `CompletionException` is thrown when an error or other exception is encountered in the course of completing a result or task.

In addition, Java 8 comes with the following new classes in the `java.util.concurrent` package:

- ➔ `CompletableFuture`
- ➔ `CountedCompleter`
- ➔ `ConcurrentHashMap.KeySetView`

13.2.1 CompletableFuture Class

The `CompletableFuture` class implements the `CompletionStage` and the `Future` interface to simplify coordination of asynchronous operations. An implementation of the already existing `Future` interface represents the result of an asynchronous computation. The `get()` method of `Future` returns the result of a computation once the computation completes, explicitly cancelled, or an exception gets thrown. This is a limitation in asynchronous programming that the `CompletableFuture` class is designed to address. The methods of the `CompletableFuture` class can run asynchronously and do not stop program execution.

Methods

Table 13.1 explains the important methods available in the `CompletableFuture` class.

Method	Description
<code>supplyAsync()</code>	Accepts a <code>Supplier</code> object that contains code to be executed asynchronously. This method after asynchronously executing the code returns a new <code>CompletableFuture</code> object on which other methods can be applied.
<code>thenApply()</code>	Returns a new <code>CompletableFuture</code> object that is executed with the result of the completed stage, provided the current stage completes normally.
<code>join()</code>	Returns the result when the current asynchronous computation completes, or throws an exception of type <code>CompletionException</code> .
<code>thenAccept()</code>	Accepts a <code>Consumer</code> object. On the completion of the current stage, this method wraps the result with the <code>Consumer</code> object and returns a new <code>CompletableFuture</code> object.
<code>whenComplete()</code>	Uses <code>BiConsumer</code> as an argument. Once the calling completion stage completes, <code>whenComplete()</code> method applies completion stage result on <code>BiConsumer</code> . <code>BiConsumer</code> accepts the result as the first argument and error if any as the second argument.
<code>getNow()</code>	Sets the value passed to it as the result if the calling completion stage is not completed.

Table 13.1: Methods of the `CompletableFuture` Class

13.2.2 CountedCompleter Class

The `CountedCompleter` class extends `ForkJoinTask` to represent a completion action performed when triggered, provided there are no pending actions. The `compute()` method of the `CountedCompleter` class does the main computation and typically invokes the `tryComplete()` method once before returning. The `tryComplete()` method checks if the pending count is nonzero, and if so, decrements the count. Otherwise, the `tryComplete()` method invokes the `onCompletion(CountedCompleter)` method and attempts to complete this task's completer, and if successful, marks this task as complete.

Optionally, the `CountedCompleter` class may override following methods:

- `onCompletion(CountedCompleter)`: To perform some action upon normal completion.
- `onExceptionalCompletion(Throwable, CountedCompleter)`: To perform some action when an exception is thrown.

A `CountedCompleter` class is declared as `CountedCompleter<Void>` when the class does not generate results. Such a class returns null as a result value. For such a class, the `getRawResult()` method needs to be overridden to provide a result from `join()`, `invoke()`, and related methods.

Code Snippet 1 shows the implementation of the `CountedCompleter` class.

Code Snippet 1:

```
package com.training.demo.countedcompleter;

import java.util.ArrayList;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.CountedCompleter;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
public class CountedCompleterDemo {
  static class NumberComputator extends CountedCompleter<Void> {
    final ConcurrentLinkedQueue<String> concurrentLinkedQueue;
    final int start;
    final int end;
    NumberComputator(ConcurrentLinkedQueue<String>
concurrentlinkedQueue, int start, int end) {
      this(concurrentlinkedQueue, start, end, null);
    }
    NumberComputator(ConcurrentLinkedQueue<String>
concurrentlinkedQueue, int start, int end,
NumberComputator parent) {
      super(parent);
      this.concurrentlinkedQueue=concurrentLinkedQueue;
      this.start=start;
      this.end=end;
    }
  }
}
```

```

}

@Override
public void compute() {
  if (end - start < 5) {
    String s = Thread.currentThread().getName();
    for (int i = start; i < end; i++) {
      concurrentLinkedQueue.add(String.format("Iteration
number: %d performed by thread %s", i, s));
    }
    propagateCompletion();
  } else {
    int mid = (end + start) / 2;
    NumberComputator subTaskA = new
    NumberComputator(concurrentlinkedQueue, start,
    mid, this);
    NumberComputator subTaskB = new
    NumberComputator(concurrentlinkedQueue, mid, end,
    this);
    setPendingCount(1);
    subTaskA.fork();
    subTaskB.compute();
  }
}

public static void main(String[] args) throws
ExecutionException, InterruptedException {
  ConcurrentLinkedQueue<String> linkedQueue = new
  ConcurrentLinkedQueue<>();
  NumberComputator numberComputator = new
}

```

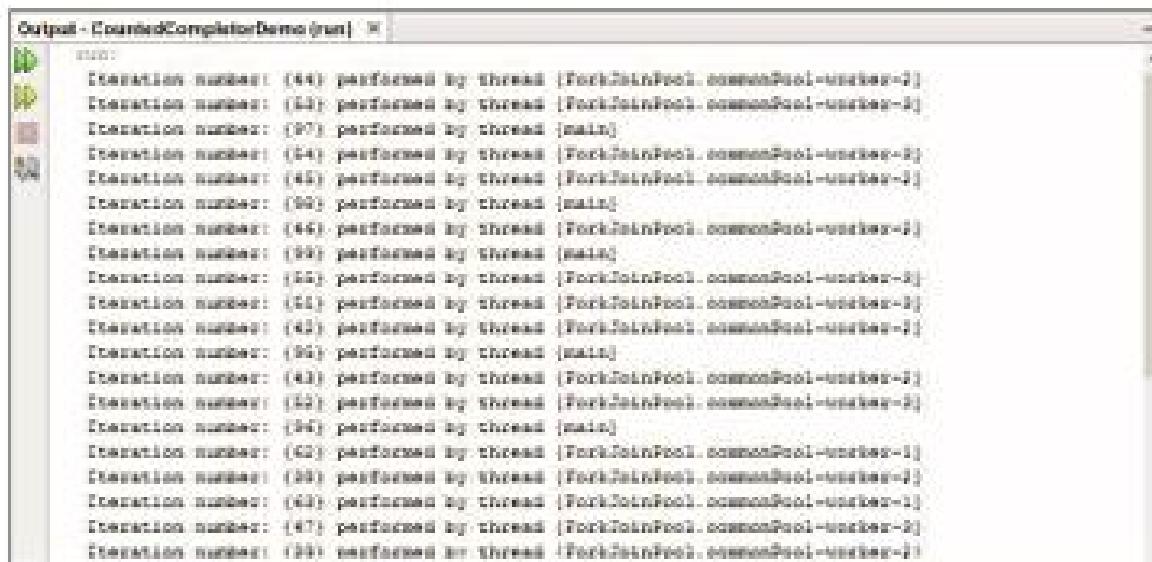
```
ForkJoinPool.commonPool().invoke(numberComputator);

ArrayList<String> list = new
ArrayList<>(linkedQueue);
for (String listItem : list) {
System.out.println(" " + listItem);
}
}
}
}
```

The code creates a `CountedCompleterDemo` class with a static inner `NumberComputator` class. The `NumberComputator` class has two overloaded constructors. The first constructor accepts following parameters: a `ConcurrentLinkedQueue` and the start and end index to perform iterations. The first overloaded constructor, in turn, calls the second one that has an additional parameter, a reference to itself. The `if` block in the overridden `compute()` method loops through the start and end index if their difference is less than five and finally calls `propagateCompletion()` to mark that the current task is complete. Otherwise, two `NumberComputator` sub-tasks are created. The `setPendingCount(1)` method with the argument 1 indicates that only the `subTaskA` sub-task is forked. The `compute()` method called on `subTaskB` makes `subTaskB` execute synchronously.

Then, the `main()` method invokes an initialized `NumberComputator` using a `ForkJoinPool` and outputs the result to console.

Figure 13.1 displays one of the possible outputs on executing the `CountedCompleterDemo` class.



```
Output - CountedCompleterDemo.java [run] * 
[1]: Iteration number: (14) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (15) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (16) performed by thread [main]
Iteration number: (17) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (18) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (19) performed by thread [main]
Iteration number: (20) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (21) performed by thread [main]
Iteration number: (22) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (23) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (24) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (25) performed by thread [main]
Iteration number: (26) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (27) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (28) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (29) performed by thread [ForkJoinPool.commonPool-worker-2]
Iteration number: (30) performed by thread [ForkJoinPool.commonPool-worker-2]
```

Figure 13.1: `CountedCompleterDemo` – Output

13.2.3 ConcurrentHashMap.KeySetView Class

The `ConcurrentHashMap.KeySetView` provides a convenient view of the keys contained in a `ConcurrentHashMap`. `ConcurrentHashMap.KeySetView` implements the `Set` interface, and therefore, you access the `ConcurrentHashMap` keys as a `Set` object. The `Set` object and `ConcurrentHashMap` object maintains a bi-directional relationship. Therefore, updating the `Set` object updates `ConcurrentHashMap` and vice-versa.

Note: You cannot directly create a `ConcurrentHashMap.KeySetView` instance. It is used when you call `keySet()`, `keySet(V)`, `newKeySet()`, and `newKeySet(int)` on a Map implementation, such as `ConcurrentHashMap`.

Code Snippet 2 shows creation of a `KeySetViewDemo` class to access the keys of a `ConcurrentHashMap` as a `Set`.

Code Snippet 2:

```
package com.training.demo.keysview;

import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

public class KeySetViewDemo {
  public static void main(String[] args) {
    Map<String, String> map =
      new ConcurrentHashMap<>();
    map.put("Java", "Java");
    map.put("Java EE", "Java EE");
    map.put("Spring", "Spring");
    Set keySet = map.keySet();
    System.out.println(keySet);
  }
}
```

The code creates a `Map` and initializes it with key value pairs. The call to `keySet()` uses `ConcurrentHashMap.KeySetView` to return a `Set` of keys, which the code prints to the output.

Figure 13.2 displays the output of the KeySetViewDemo class.

```
Output - KeySetViewDemo [run] X
sum:
[Java EE, Java, Spring]
BUILD SUCCESSFUL (total time: 2 seconds)
```

Figure 13.2: KeySetViewDemo – Output

13.3 Atomic Operations and Locks

In Java applications, one challenge related to scalability is to maintain a single count or sum that is simultaneously updated by multiple threads. Java 8 addresses the challenge through a small set of new classes in the `java.util.concurrent.atomic` package. These classes are designed to employ contention-reduction techniques to provide throughput improvements as compared to `Atomic` variables. The new classes are as follows:

- ➔ `LongAccumulator`: Maintains a long running value updated using a supplied function
- ➔ `LongAdder`: Maintains an initially zero long sum
- ➔ `DoubleAccumulator`: Maintains a double running value updated using a supplied function
- ➔ `DoubleAdder`: Maintains an initially zero double sum

Code Snippet 3 shows the use of the `LongAdder` and `DoubleAdder` atomic operation classes.

Code Snippet 3:

```
package com.training.demo.atomicoperation;

import java.util.concurrent.atomic.DoubleAdder;
import java.util.concurrent.atomic.LongAdder;
public class AtomicOperationClassDemo {
    private final LongAdder longAdder;
    private final DoubleAdder doubleAdder;
    public AtomicOperationClassDemo(LongAdder longAdder,
        DoubleAdder doubleAdder) {
        this.longAdder = longAdder;
    }
}
```

```

this.doubleAdder = doubleAdder;
}

public void incrementLong() {
    longAdder.increment();
}

public long getLongCounter() {
    return longAdder.longValue();
}

public void addDouble(int doubleValue) {
    doubleAdder.add(doubleValue);
}

public double getSumAsDouble() {
    return doubleAdder.doubleValue();
}

public static void main(String[] args) {
    AtomicOperationClassDemo atomicOperationClassDemo1 = new
    AtomicOperationClassDemo(new LongAdder(), new
    DoubleAdder());
    System.out.println("-----Long Counter-----");
    for (int i = 0; i < 10; i++) {
        atomicOperationClassDemo1.incrementLong();
        System.out.println("Long Counter " +
        atomicOperationClassDemo1.getLongCounter());
    }
    System.out.println("-----Double Sum-----");
    for (int j = 0; j < 10; j++) {
        atomicOperationClassDemo1.addDouble(j);
        System.out.println("Double Sum " +
        atomicOperationClassDemo1.getSumAsDouble());
    }
}

```

```

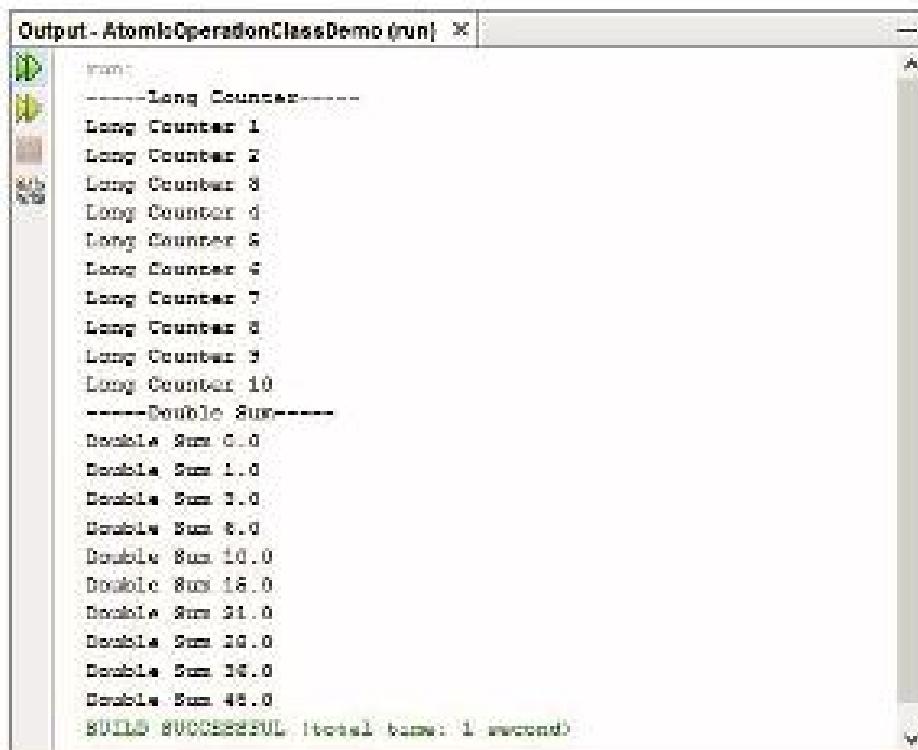
}
}
}

```

The code initializes a LongAdder and a DoubleAdder in the constructor. The incrementLong() method increments the current LongAdder value by 1 through a call to LongAdder.increment(). The getLongCounter() method returns the current LongAdder value. The addDouble() method adds the double value passed to it to the current value of DoubleAdder through a call to DoubleAdder.add(). The getSumAsDouble() method returns the current DoubleAdder value.

Then, the main() method instantiates AtomicOperationClassDemo with the LongAdder and DoubleAdder objects. The first loop calls the incrementLong() method followed by the getLongCounter() method. The second loop calls the addDouble() method followed by the getSumAsDouble() method. The code also prints the result out to the console from both the for loops.

Figure 13.3 displays the output of the AtomicOperationClassDemo class.



```

Output - AtomicOperationClassDemo [run] X
run:
-----
Long Counter
Long Counter 1
Long Counter 2
Long Counter 3
Long Counter 4
Long Counter 5
Long Counter 6
Long Counter 7
Long Counter 8
Long Counter 9
Long Counter 10
-----
Double Sum
Double Sum 0.0
Double Sum 1.0
Double Sum 2.0
Double Sum 3.0
Double Sum 4.0
Double Sum 5.0
Double Sum 6.0
Double Sum 7.0
Double Sum 8.0
Double Sum 9.0
Double Sum 10.0
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 13.3: AtomicOperationClassDemo – Output

Another major feature that Java 8 introduces to concurrent programming is the StampedLock class of the java.util.concurrent.locks package.

Prior to Java 8, `ReadWriteLock` was the most common way to implement locks. However, `ReadWriteLock` has several shortcomings. The primary shortcoming being that it suffered from starvation (for example, starvation can occur when many threads acquire read locks and only few threads acquire write locks). Also through `ReadWriteLock`, it is not possible to upgrade a read lock into a write lock. Another limitation is the lack of support for optimistic reads. The new `StampedLock` class of Java 8 addresses all these shortcomings. Instead of the usual locking mechanism, the `StampedLock` class returns a long number, also known as stamp, whenever a lock is granted. This stamp can be used either to release a lock or to check if the lock is still valid. Additionally, stamped locks support a new lock mode called optimistic locking.

The `StampedLock` class implements lock with three modes for controlling read/write access. These modes are as follows:

- ➔ **Writing:** Achieved through the `writeLock()` method that exclusively acquires the lock, blocking if necessary until available. The `writeLock()` method returns a stamp that can be used in the `unlockWrite(long)` method to release the lock. Writing mode is also supported by the timed and untimed versions of `tryWriteLock()`. When a thread holds the lock in write mode, no read locks can be obtained, and all optimistic read validations fail.
- ➔ **Reading:** Achieved through the `readLock()` method that non-exclusively acquires the lock, blocking if necessary until available. The `readLock()` method returns a stamp that can be used in the `unlockRead(long)` method to release the lock. Reading mode is also supported by the timed and untimed versions of `tryReadLock()`.
- ➔ **Optimistic Reading:** Achieved through the `tryOptimisticRead()` and `validate(long)` methods. The `tryOptimisticRead()` method returns a non-zero stamp only if the lock is not currently held in write mode. The `validate(long)` returns true if the lock has not been acquired by some other thread in write mode since obtaining the given stamp. As the optimistic reading mode can be easily taken over by a writer at any time, this mode should be carefully used. It is recommended to use the optimistic reading mode to improve throughput of short read-only code segments that does not have contentions.

Code Snippet 4 shows the use of the `StampedLock` class in the reading, writing, and optimistic reading mode.

Code Snippet 4:

```
package com.training.demo.stampedlock;
import java.util.concurrent.locks.StampedLock;
public class StampedLockDemo {
    private final StampedLock stampedLock = new StampedLock();
    private double balance;
    public StampedLockDemo(double balance) {
        this.balance = balance;
        System.out.println("Available balance: "+balance);
    }
}
```

```

 1 public void deposit(double amount) {
 2     System.out.println("\nAbout to deposit $: " + amount);
 3     long stamp = stampedLock.writeLock();
 4     System.out.println("Applied read lock");
 5     try {
 6         balance += amount;
 7         System.out.println("Available balance: " + balance);
 8     } finally {
 9         stampedLock.unlockWrite(stamp);
10         System.out.println("Unlocked write lock");
11     }
12 }
13
14 public void withdraw(double amount) {
15     System.out.println("\nAbout to withdraw $: " + amount);
16     long stamp = stampedLock.writeLock();
17     System.out.println("Applied write lock");
18     try {
19         balance -= amount;
20         System.out.println("Available balance: " + balance);
21     } finally {
22         stampedLock.unlockWrite(stamp);
23         System.out.println("Unlocked write lock");
24     }
25 }
26
27 public double checkBalance() {
28     System.out.println("\nAbout to check balance");
29     long stamp = stampedLock.readLock();
30     System.out.println("Applied read lock");
31     try {
32
  
```

```

System.out.println("Available balance: "+balance);
return balance;
} finally {
    stampedLock.unlockRead(stamp);
    System.out.println("Unlocked read lock");
}
}

public double checkBalanceOptimisticRead() {
    System.out.println("\nAbout to check balance with
optimistic read lock");
    long stamp = stampedLock.tryOptimisticRead();
    System.out.println("Applied non-blocking optimistic read
lock");
    double balance = this.balance;
    if (!stampedLock.validate(stamp)) {
        System.out.println("Stamp have changed. Applying full-
blown read lock.");
        stamp = stampedLock.readLock();
        try {
            balance = this.balance;
        } finally {
            stampedLock.unlockRead(stamp);
            System.out.println("Unlocked read lock");
        }
    }
    System.out.println("Available balance: "+balance);
    return balance;
}
public static void main(String[] args) {
}

```

```

StampedLockDemo stampedLockDemo=new
StampedLockDemo(4000.00);
stampedLockDemo.withdraw(1000.00);
stampedLockDemo.deposit(5000.00);
stampedLockDemo.checkBalance();
stampedLockDemo.checkBalanceOptimisticRead();
}
}

```

The `deposit()` method in the code acquires a write lock through a call to `StampedLock.writeLock()` before updating the balance field. The finally block inside the `deposit()` method releases the lock.

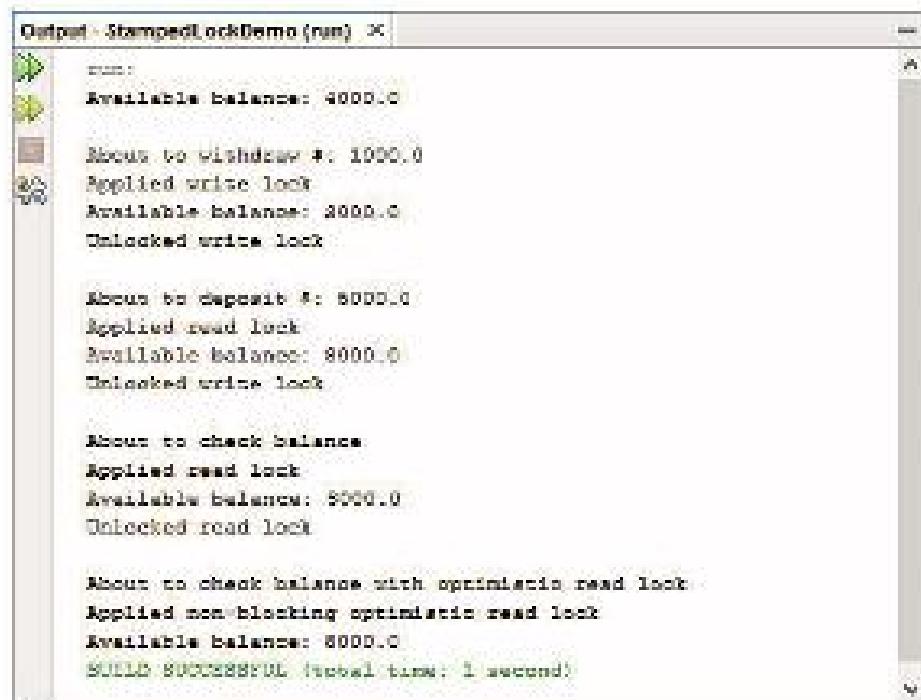
The `withdraw()` method also acquires a write lock through a call to `StampedLock.writeLock()` before updating the balance field, and releases the lock in the finally block. Then, the `checkBalance()` method acquires a read lock through a call to `StampedLock.readLock()` to read the balance and releases the lock in the finally block.

Next, the `checkBalanceOptimisticRead()` method acquires an optimistic read method through a call to `StampedLock.tryOptimisticRead()`.

After obtaining the current balance, the code checks whether or not the stamp is still valid through a call to the `StampedLock.validate()`. If the stamp is valid, the current balance is returned. Else, a full read lock is acquired before reading the balance again. The finally block releases the lock with a call to `StampedLock.unlockRead()`.

The `main()` method creates a `StampedLockDemo` object initialized with a long amount. Finally, the `withdraw()`, `deposit()`, `checkBalance()`, and `checkBalanceOptimisticRead()` methods are called in sequence.

Figure 13.4 displays the output of the StampedLockDemo class.



```

Output - StampedLockDemo (run) X
=====
run:
Available balance: 4000.0

Access to withdraw #: 1000.0
Applied write lock
Available balance: 3000.0
Unlocked write lock

Access to deposit #: 5000.0
Applied read lock
Available balance: 8000.0
Unlocked write lock

Access to check balance
Applied read lock
Available balance: 8000.0
Unlocked read lock

Access to check balance with optimistic read lock
Applied non-blocking optimistic read lock
Available balance: 8000.0
BUILD SUCCESSFUL (total time: 1 second)
  
```

Figure 13.4: StampedLockDemo – Output

13.4 More Features of the Fork-Join Framework

Some additional features of the Fork-Join framework include the new features added to the `ForkJoinPool` class in Java 8, stream parallelization, and parallelization of array sorting.

13.4.1 New `ForkJoinPool` Features

Java 8 introduces the common thread pool feature that allows any `ForkJoinTask` that is not explicitly submitted to a specified thread pool to use a common thread. The use of common thread pool greatly helps the application to reduce resource usage.

The new methods of the `ForkJoinPool` class are as follows:

- ➔ `commonPool()`: A static method that returns a common thread pool.
- ➔ `getCommonPoolParallelism()`: A new method that returns the targeted parallelism level of the common thread pool.

13.4.2 Stream Parallelization

Java 8 introduces the new Stream API that operates on collections and arrays to produce pipelined data that can be used to perform operations, such as filter, sort, and iterate through data. To support faster processing of large streams, traditional multi-thread programming approaches can be taken. However, by default, streams are not thread-safe and therefore using multiple threads to work on streams should be carefully considered to avoid any threading issues. To address the limitations of the multi-thread programming approaches to work with streams, Java 8 provides support for parallel computation of streams. With parallel computation, streams can now be accessed faster without the risk of threading issues.

Code Snippet 5 shows the use of parallel stream to iterate through the elements of an ArrayList.

Code Snippet 5:

```
package com.training.demo.parallelstream;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
public class ParallelStreamDemo {
    public static void main(String[] args) {
        List<String> items = new ArrayList<String>();
        items.add("one");
        items.add("two");
        items.add("three");
        items.add("four");
        Stream parallelStream = items.parallelStream();
        parallelStream.forEach(System.out::println);
    }
}
```

The code creates a List and initializes it with string elements. The parallelStream() method returns a parallel stream object of type Stream. The foreach() loop iterates through the parallel stream and prints out the elements.

Figure 13.5 displays the output of the ParallelStreamDemo class.



```
Output - ParallelStreamDemo (run) [x]
100
7333
4000
2000
1000
500
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 13.5: ParallelStreamDemo – Output

13.4.3 Arrays Sort Parallelism

Java 8 introduces a new `parallelSort()` method in the `Arrays` class that allows parallel sorting of array elements.

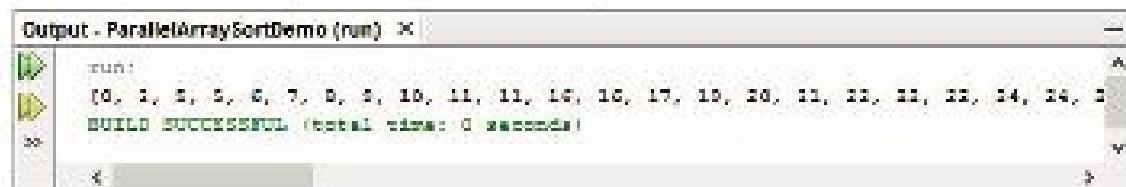
Code Snippet 6 shows how to sort arrays in parallel.

Code Snippet 6:

```
package com.training.demo.parallelarraysort;
import java.util.Arrays;
public class ParallelArraySortDemo {
  public static void main(String[] args) {
    int[] intArray = new int[100];
    for(int i=0; i<intArray.length; i++) {
      intArray[i] = (int) (Math.random() * 100);
    }
    Arrays.parallelSort(intArray);
    System.out.println(Arrays.toString(intArray));
  }
}
```

The code creates an `int` array of size 100 and fills it with random numbers starting from 0 to 100. The `Arrays.parallelSort()` method sorts the array in parallel before the code prints the array elements to the console.

Figure 13.6 displays the output of the `ParallelArraySortDemo` class.



```
Output - ParallelArraySortDemo (run) ×
run:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 13.6: `ParallelArraySortDemo` – Output

13.4.4 Recursive Action

The `ForkJoinPool` class, which is a part of the Fork-Join framework, extends the `AbstractExecutorService` class to implement the main work-stealing algorithm to execute `ForkJoinTask` processes. `RecursiveTask` and `RecursiveAction` that are implementations of `ForkJoinTask` are similar in how they function. Both `RecursiveTask` and `RecursiveAction` extend `ForkJoinTask` to represent tasks that run within a `ForkJoinPool`. The difference is that while `RecursiveTask` returns a result while `RecursiveAction` does not.

Code Snippet 7 shows the use of Fork-Join functionality with `RecursiveAction`.

Code Snippet 7:

```
package com.training.demo.recursiveaction;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
public class RecursiveActionDemo extends RecursiveAction {
private long assignedWork = 0;
public RecursiveActionDemo(long assignedWork) {
this.assignedWork = assignedWork;
}
private List<RecursiveActionDemo> createSubtasks() {
List<RecursiveActionDemo> subtaskList = new
ArrayList<>();
RecursiveActionDemo subtask1 = new
RecursiveActionDemo(this.assignedWork / 2);
RecursiveActionDemo subtask2 = new
RecursiveActionDemo(this.assignedWork / 2);
```

```

subtaskList.add(subtask1);
subtaskList.add(subtask2);
return subtaskList;
}

@Override
protected void compute() {
if (this.assignedWork > 50) {
System.out.println("Splitting assignedWork : " +
Thread.currentThread() + " computing:::" +
this.assignedWork);
List<RecursiveActionDemo> subtaskList = new
ArrayList<>();
subtaskList.addAll(createSubtasks());
for (RecursiveAction subtask : subtaskList) {
subtask.fork();
}
} else {
System.out.println("Main thread" +
Thread.currentThread() + " computing:::" +
this.assignedWork);
}
}

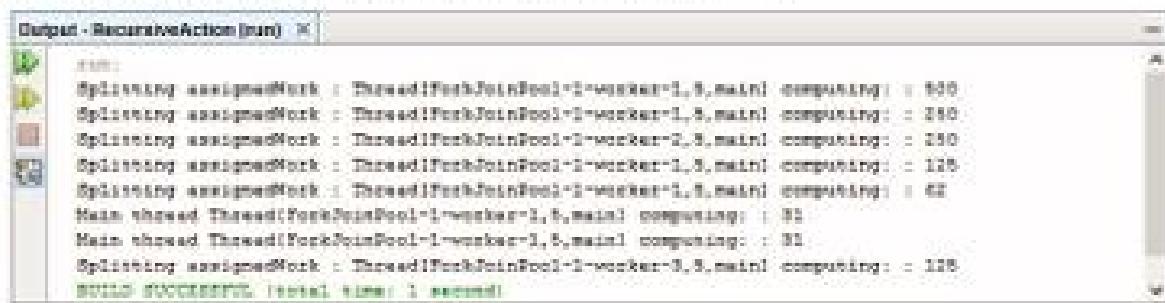
public static void main(String[] args) {
RecursiveActionDemo recursiveActionDemo = new
RecursiveActionDemo(500);
final ForkJoinPool forkJoinPool = new ForkJoinPool(4);
forkJoinPool.invoke(recursiveActionDemo);
}
}

```

The constructor contains code to create a RecursiveActionDemo initialized with the assignedWork long value passed to it when the constructor is invoked. The `createSubtasks()` method creates two subtasks and returns them as a List object to the caller.

The overridden `compute()` method calls the `createSubtasks()` method if the assignedWork is above 50 and then calls the `fork()` method of the subtasks to distribute work. If the workload is below 50, then the work is carried out by `RecursiveActionDemo` itself. The `main()` method creates a `RecursiveActionDemo` initialized with the value 500 and calls the `invoke()` method on a `ForkJoinPool` object to start the recursive action.

Figure 13.7 displays the output of the `RecursiveActionDemo` class.



```
Output - RecursiveAction.java
[1]: Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: > 500
[2]: Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: > 250
[3]: Splitting assignedWork : Thread[ForkJoinPool-1-worker-2,5,main] computing: > 250
[4]: Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: > 125
[5]: Splitting assignedWork : Thread[ForkJoinPool-1-worker-1,5,main] computing: > 62
Main thread Thread[ForkJoinPool-1-worker-1,5,main] computing: > 62
Main thread Thread[ForkJoinPool-1-worker-1,5,main] computing: > 62
Splitting assignedWork : Thread[ForkJoinPool-1-worker-3,5,main] computing: > 125
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 13.7: `RecursiveActionDemo` – Output

13.5 Check Your Progress

1. Which of the following classes implements the CompletionStage and the Future interfaces to simplify coordination of asynchronous operations?

(A)	CompletableFuture	(C)	ConcurrentHashMap.KeySetView
(B)	CountedCompleter	(D)	CompletableFuture.AynchronousCompletionTask

2. Which of the following statements is true regarding the ConcurrentHashMap.KeySetView class?

(A)	Provides access to the key/value pairs of a ConcurrentHashMap.	(C)	Allows adding keys to a ConcurrentHashMap asynchronously.
(B)	Provides access to the keys present in a ConcurrentHashMap.	(D)	Allows updating existing keys in a ConcurrentHashMap asynchronously.

3. Which of the following methods of the StampedLock class attempts to perform an optimistic read?

(A)	tryRead()	(C)	tryOptimisticRead()
(B)	optimisticRead()	(D)	optimisticReadLock()

4. What is the use of the getCommonPoolParallelism() method of the ForkJoinPool class?

(A)	Returns a common thread pool	(C)	Returns the targeted parallelism level of a thread
(B)	Returns a thread from a common pool that can be executed in parallel with the main thread	(D)	Returns the targeted parallelism level of the common thread pool

5. Which method of the Arrays allows parallel sorting of array elements?

(A)	parallelSort()	(C)	parallelArraySort()
(B)	sortParallel()	(D)	sortParallelArray()

13.5.1 Answers

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

Summary

- ➊ The `CompletableFuture` class simplifies coordination of asynchronous operations.
- ➋ The `CountedCompleter` class represents a completion action performed when triggered, provided there are no remaining pending actions.
- ➌ The `LongAccumulator`, `LongAdder`, `DoubleAccumulator`, and `DoubleAdder` classes provide better throughput improvements as compared to `Atomic` variables.
- ➍ The `StampedLock` class implements lock to control read/write access.
- ➎ Parallel computation of streams enables working with streams faster without the risk of threading issues.
- ➏ The new `parallelSort()` method in the `Arrays` class allows parallel sorting of array elements.
- ➐ `RecursiveTask`, similar to `RecursiveAction` extends `ForkJoinTask` to represent tasks that run within a `ForkJoinPool`.

Session - 14

Java Class Design and Advanced Class Design

Welcome to the Session, Java Class Design and Advanced Class Design.

This session covers the important points related to class design in Java such as access control modifiers, use of packages, static and inner classes, and so on.

In this Session, you will learn to:

- Explain Java access modifiers
- Explain advanced OOP concepts in Java
- Describe packages
- Define abstract class
- Explain the use of static and final keywords
- Identify different types of inner classes
- Explain advanced types



14.1 Access Control

Access control determines how a class and its member variables and methods are accessible and used by other classes and objects.

Java provides access modifiers to control access at the following levels:

- **Class level:** Applies to the class.
- **Member level:** Applies to the member variables and methods of the class.

14.1.1 Access Modifiers

An access modifier controls the access of class members and variables by other objects. Various access modifiers in Java are as follows:

- **public:** Can be applied to classes, member variables, and methods. Public variables and methods are accessible from within the same class, the same package in which the class is created, and also from a different package. This modifier is applied using the `public` keyword.
- **protected:** Can be applied to member variables and methods. Protected variables and methods are accessible only to the class in which they are declared and its subclasses. This modifier is applied using the `protected` keyword.
- **private:** Can be applied to member variables and methods. Private member variables and methods are accessible only to the class in which they are declared. This modifier is applied using the `private` keyword.

Besides these, there is friendly or package access, which is automatically applied to classes, member variables, and methods when no other access modifiers are specified. This modifier does not have any keyword. A class, variable, or method with friendly or package access is accessible only to the classes and other classes within the same package.

Access Modifier Map

Table 14.1 shows the access levels of the different access modifiers.

Access Modifier	Within Class	Within Package	Subclass Outside Package	Global
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
Friendly or package	Yes	Yes	No	No
private	Yes	No	No	No

Table 14.1: Access Modifier Map

14.1.2 Access Control Best Practices

Some of the best practices that should be followed when applying access control to Java classes, variables, and methods are as follows:

- Declare member variables and methods as private for information hiding, also known as encapsulation.
- Declare only those methods required to create objects as public.
- Do not declare member variables as public, except for constants. If access is required to member variables, use a corresponding public getter method.
- Declare variables and methods as protected if there is chance that a subclass might need them in the future. Otherwise, use the more restrictive friendly or private access modifier.

14.2 Advanced OOP Concepts in Java

Java is an object-oriented programming language. In order to design Java classes and interfaces in a better manner, a developer needs to be well-versed with OOP concepts.

OOP is a software development methodology based on modeling a real-world system. An object is the core concept involved in OOP. It is the representation of a real world entity. OOP can be considered as a programming approach involving object collections and their inter-relationships.

14.2.1 Multi-level Inheritance

Inheritance enables a class to inherit members of a class such as its variables and methods. Inheritance supports reusability of an existing class in a new class. Besides inheriting variables and classes from a super class, a subclass can additionally add new features to it. A subclass inherits from a superclass using the `extends` keyword. A Java class can extend from only one superclass. However, a class can extend from multiple interfaces.

Note: A subclass cannot inherit private variables and methods of the superclass. A subclass outside the package of the superclass can only access public and protected variables and methods of the superclass.

Java supports multi-level inheritance. One class can inherit from a parent class, which itself inherits from another parent class. Such inheritance hierarchy can extend to any level. However, irrespective of the level in a multi-level inheritance, the `Object` class always remains on the top of the hierarchy.

Code Snippet 1 shows an example of multi-level inheritance hierarchy.

Code Snippet 1:

```
package com.classdesign.demo;

public class Movie {
    String language="English";
    String type="Full length movie";
    void getMovie() {
        System.out.println("Language "+language);
        System.out.println("Type: "+type);
    }
}
```

The class creates two variables and a `getMovie()` method to print the variable values.

Code Snippet 2 shows an `ActionMovie` class that extends `Movie` forming an inheritance hierarchy.

Code Snippet 2:

```
package com.classdesign.demo;

public class ActionMovie extends Movie {
    String title="War Lord";
    String duration="1:50";
    String genre="Action";
    void getActionMovie() {
        System.out.println("Genre: "+genre);
        System.out.println("Title: "+title);
        System.out.println("Duration: "+duration);
    }
}
```

The class extends from `Movie`, declares three variables, and defines a `getActionMovie()` method that prints the variables.

Code Snippet 3 shows an `AnimatedActionMovie` class that extends `Movie` further forming an inheritance hierarchy.

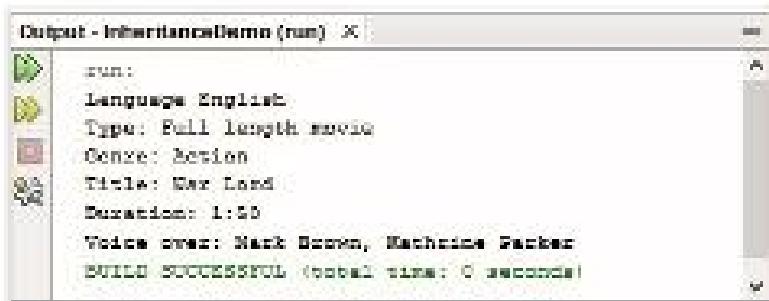
Code Snippet 3:

```
package com.classdesign.demo;

public class AnimatedActionMovie extends ActionMovie {
    String voiceOver = "Mark Brown, Kathrine Parker";
    void getAnimatedActionMovie() {
        System.out.println("Voice over: " + voiceOver);
    }
    public static void main(String[] args) {
        AnimatedActionMovie aActionMovie = new
        AnimatedActionMovie();
        aActionMovie.getMovie();
        aActionMovie.getActionMovie();
        aActionMovie.getAnimatedActionMovie();
    }
}
```

The `AnimatedActionMovie` class extends `ActionMovie` forming a multi-level inheritance hierarchy. The `AnimatedActionMovie` class contains a `voiceOver` variable and a `getVoiceOver()` method that returns the variable. The `main()` method creates an `AnimatedActionMovie` object and calls the `getMovie()` and `getActionMovie()` methods of the `Movie` and `ActionMovie` superclasses. The `main()` method also calls the `getAnimatedActionMovie()` method of its class.

Figure 14.1 displays the output of the `Movie` class.



```
Output - InheritanceDemo (run) X
run:
Language English
Type: Full length movie
Genre: Action
Title: War Land
Duration: 1:50
Voice over: Mark Brown, Kathrine Parker
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 14.1: ActionMovie – Output

14.2.2 Method Overloading

In object-oriented programming, every method has a signature. This comprises the number of parameters passed to the method, the data types of parameters, and the order in which the parameters are written. While declaring a method, the signature of the method is written in parentheses next to the method name.

No class is allowed to contain two methods with the same name and signature. However, it is possible for a class to have two methods having the same name but different signatures. The concept of declaring more than one method with the same method name, but different signatures is called method overloading.

In method overloading where multiple methods exist with the same name but different method signatures, a call to the correct method is resolved at compile time through static polymorphism.

Code Snippet 4 overloads the `add()` method to calculate the square of the `int` and `float` values passed as parameters.

Code Snippet 4:

```
package com.classdesign.demo;

public class MethodOverloadingDemo {
    static int add(int intNum1, int Num2) {
        return intNum1 + Num2;
    }

    static float add(float floatNum1, float floatNum2) {
        return floatNum1 + floatNum2;
    }

    public static void main(String[] args) {
        System.out.println("Sum of int value: " + add(5, 15));
        System.out.println("Sum of float value: " + add(5.95f,
        15.3f));
    }
}
```

The code creates two methods with the same name, but with different parameters declared in the class. The two `add()` methods take in two parameters of `int` and `float` types respectively. Within the `main()` method, depending on the type of value passed, the appropriate method is invoked through static polymorphism. The sum of the specified numbers is then printed to the console.

Figure 14.2 displays the output that displays the usage of static polymorphism through method overloading.

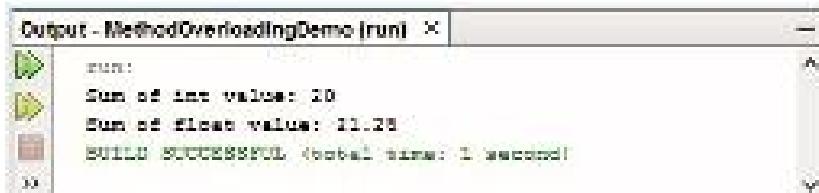


Figure 14.2: MethodOverloadingDemo – Output

Method overloading is also used extensively in the Java API. One example is the built-in `abs()` method of the `Math` class present in the `java.lang` package. This method returns the absolute value of the numeric parameter passed to it. There are several overloaded versions of the `abs()` method, such as `abs(int num)`, `abs(float num)`, and `abs(double num)`. Based on the parameter passed to the `abs()` method at runtime, Java determines the correct overloaded `abs()` method to invoke.

14.2.3 Method Overriding

In an inheritance hierarchy, a subclass can override the methods of the superclass. This feature, known as method overriding, is defined as creating a method in the subclass that has the same return type and signature as a method defined in the superclass. Method overriding is a form of dynamic polymorphism because the decision to call a particular implementation over other implementations is dynamically taken at runtime. The call decision is taken dynamically at runtime based on the object from which the operation is called.

Code Snippet 5 shows an example of method overriding to implement dynamic polymorphism.

Code Snippet 5:

```
package com.classdesign.demo;

class Animal {
    void getMessage() {
        System.out.println("Message from Animal.");
    }
}

class Dog extends Animal {
    @Override
    void getMessage() {
        System.out.println("Bow-wow! Message from Dog.");
    }
}
```

```
class Cat extends Animal {
    @Override
    void getMessage() {
        System.out.println("Meow! Message from Cat.");
    }
}

public class MethodOverridingDemo {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();
        animal1.getMessage();
        animal2.getMessage();
    }
}
```

The code creates an inheritance hierarchy with `Animal` as the base class containing the `getMessage()` method. Two classes, `Dog` and `Cat`, inherit `Animal` and override the `getMessage()` method to provide their own implementations.

`MethodOverridingDemo` is the main class whose `main()` method demonstrates dynamic polymorphism. The `main()` method creates a `Dog` and a `Cat` instance and assigns them to the `Animal` types, `animal1` and `animal2`. Polymorphism ensures that calls to `getMessage()` on the `animal` types are dispatched at runtime to the actual instances.

Figure 14.3 displays the output on executing the `MethodOverridingDemo` class.

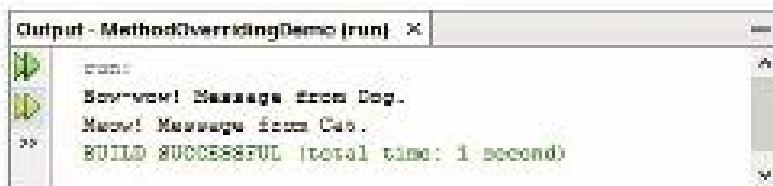


Figure 14.3: MethodOverridingDemo – Output

14.2.4 `super()` Constructor Call and `super` Keyword

In a Java inheritance hierarchy, a subclass can invoke the constructor of its super class using the `super()` constructor call. An overridden method of a class can also call its superclass method using the `super` keyword.

The syntax of the `super()` constructor call is as follows:

Syntax:

```
super();
```

or

```
super(parameter_list);
```

In the syntax, `super()` calls the superclass no-argument constructor and `super(parameter_list)` calls the superclass constructor with a matching parameter list that `parameter_list` in the syntax represents.

Note: `super()` call must be the first statement inside the constructor making the call.

The syntax of the `super` keyword is as follows:

Syntax:

```
super.<method_name>;
```

In the syntax, `method_name` is the overridden method name of the super class.

Code Snippet 6 shows the use of the `super()` constructor call and the `super` keyword.

Code Snippet 6:

```
package com.classdesign.demo;

class SuperClass {
    public SuperClass() {
        System.out.println("Message from SuperClass default
constructor.");
    }
    void print() {
        System.out.println("Message from SuperClass.print().");
    }
}

public class SuperCallDemo extends SuperClass {
    public SuperCallDemo() {
        super();
        System.out.println("Message from SuperCallDemo default
constructor.");
    }
}
```

```

@Override
void print() {
System.out.println("Message from SuperCallDemo.print()");
super.print();
}

public static void main(String[] args) {
SuperClass obj = new SuperCallDemo();
obj.print();
}

```

The code creates a `SuperClass` class with a default constructor and a `print()` method. The `SuperCallDemo` class extends `SuperClass` in order to inherit from it. The constructor of `SuperCallDemo` calls `super()` to invoke the default constructor of `SuperClass`. The overridden `print()` method of `SuperCallDemo` uses the `super` keyword to invoke the `print()` method of `SuperClass`.

Figure 14.4 displays the output on executing the `SuperCallDemo` class.

Figure 14.4: SuperCallDemo – Output

14.3 Package and Import Statements

Consider Sydney, which is a city in Australia as well as in Canada. You can easily distinguish between the two cities by associating them with their respective countries. Similarly, when working on a huge project, there may be situations where classes have identical names. This may result in name conflicts. This problem can be solved by having the classes in different packages. By doing this, classes can have identical names without any resultant name clashes.

In Java, a package is used to group classes and interfaces logically and prevent name clashes between those with identical names. Packages reduce any complexities when the same program is required in another application.

Although it is not mandatory to use packages for the Java classes or interfaces of an application, it is highly recommended to arrange classes and interfaces in packages to avoid naming conflicts as the application grows.

All the built-in Java classes and interfaces, based on their functionalities, are bundled into packages.

Some of these packages are as follows:

- ➔ `java.lang`: Bundles the classes that are fundamental to the design of the Java programming language.
- ➔ `java.io`: Bundles the classes to perform input output operations.
- ➔ `java.collections`: Bundles the classes and interfaces that are part of the Java collection framework.
- ➔ `java.net`: Bundles the classes to implement networking applications.
- ➔ `java.time`: Bundles the classes to work with date and time.

14.3.1 package Keyword

A package is declared with the `package` keyword. A package declaration must be the first statement in a Java class. Rules to declare a package are as follows:

- ➔ A package must be declared with class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- ➔ The name of the package must match the directory structure where the corresponding bytecode resides.

Code Snippet 7 shows the use of a user-defined package for a Java class.

Code Snippet 7:

```
package com.classdesign.demoA;
public class ClassA {
    public void getMessage () {
        System.out.println("Message from ClassA in
com.classdesign.demoA package");
    }
}
```

The code declares a package using a hierarchical naming structure where each part of the hierarchy is separated by the dot operator. The hierarchical naming structure can be related with the hierarchical directory structure of the file system.

Code Snippet 8 shows a Java class with the same name as the class in Code Snippet 6, but in a different package.

Code Snippet 8:

```
package com.classdesign.demoB;
public class ClassA {
    public void getMessage() {
        System.out.println("Message from ClassA in
com.classdesign.demoB package ");
    }
}
```

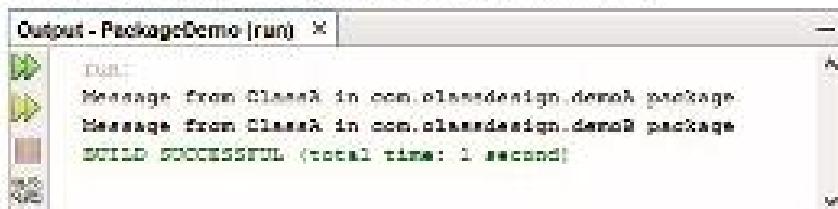
Code Snippet 9 shows a Java class using the two ClassA classes that are part of different packages.

Code Snippet 9:

```
package com.classdesign.demo;
public class PackageDemo {
    public static void main(String[] args) {
        com.classdesign.demoA.ClassA obj1 = new
        com.classdesign.demoA.ClassA();
        com.classdesign.demoB.ClassA obj2 = new
        com.classdesign.demoB.ClassA();
        obj1.getMessage();
        obj2.getMessage();
    }
}
```

The code uses the full qualified name, which is the class name along with the package name, to instantiate both the ClassA classes and calls their doMessage () methods.

Figure 14.5 displays the output on executing the PackageDemo class.



```
Output - PackageDemo [run] ×
[Run]
Message from ClassA in com.classdesign.demoA package
Message from ClassA in com.classdesign.demoA package
Message from ClassB in com.classdesign.demoB package
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 14.5: PackageDemo – Output

14.3.2 Importing Types and Packages

It is not convenient to type the fully qualified name each time a class needs to be used. To overcome this issue, Java enables importing a class once and thereafter, you can use the class with only the class name. This feature is called a single-type import.

Code Snippet 10 shows importing the ClassA class of the com.classdesign.demoA package with a single-type import.

Code Snippet 10:

```
package com.classdesign.demo;
import com.classdesign.demoA.ClassA;
public class PackageDemo {
  public static void main(String[] args) {
    ClassA obj1 = new ClassA();
    obj1.getMessage();
  }
}
```

The code imports the com.classdesign.demoA.ClassA class with an import statement that follows the package declaration. The main() method then instantiates ClassA by referring it only with the name.

Note: Java does not allow importing classes with the same name in different packages through single-type import. In such situations, you need to use the fully qualified name, as shown in Code Snippet 8.

Java also supports importing an entire package with a single import statement that automatically imports all the classes and interfaces that are part of the package.

Code Snippet 11 shows importing the entire com.classdesign.demoA package and using the ClassA class of that package.

Code Snippet 11:

```
package com.classdesign.demo;
import com.classdesign.demoA.*;
public class PackageDemo {
  public static void main (String[] args) {
    ClassA obj1 = new ClassA();
    obj1.getMessage ();
  }
}
```

Similar to single-type importing, Java does not allow importing two complete packages with the wildcard (*) symbol if they have classes with the same name. In such situations, you need to use the fully qualified name, as shown in Code Snippet 8.

Note: You do not need to explicitly import the java.lang package as the compiler does it by default for all classes.

14.4 Abstract Class

Java allows designing a class specifically to be used only as a base class by declaring it an abstract class. Such class can be referred to as an incomplete base class, as it cannot be instantiated, but it can only be sub-classed.

An abstract class contains one or more methods declared with the `abstract` keyword. An abstract method does not contain implementation. An abstract class is used to declare classes that only define common properties and behavior of other classes.

Note: A class that extends an abstract class should provide implementation of all the abstract methods or else declare itself as abstract.

Code Snippet 12 shows how to create and use an abstract class.

Code Snippet 12:

```
package com.classdesign.demo;
abstract class TestAbstract {
  abstract void getMessage ();
}
public class AbstractClassDemo extends TestAbstract {
```

```

void getMessage() {
    System.out.println("Message from abstract class.");
}

public static void main(String[] args) {
    TestAbstract abstractClass = new AbstractClassDemo();
    abstractClass.getMessage();
}
}
  
```

The code creates an abstract class named `TestAbstract` containing a `getMessage()` abstract method. The `AbstractClassDemo` class extends `TestAbstract` and implements the `getMessage()` method. The `main()` method creates a `AbstractClassDemo` instance and calls the `getMessage()` method.

Figure 14.6 displays the output on executing the `AbstractClassDemo` class.



Figure 14.6: `AbstractClassDemo` – Output

14.5 static and final Keywords

In Java, `static` and `final` are two important keywords that are extensively used while writing Java code. The `static` keyword is primarily used for memory management and can be applied to the following:

- ➔ Member variables
 - ➔ Method
 - ➔ Code block
 - ➔ Nested class
- The `final` keyword is used to restrict access in an inheritance hierarchy and can be applied to:
- ➔ Class
 - ➔ Variable
 - ➔ Method

14.5.1 Static Variable

Regular member variables, also known as instance variables or fields that are declared in a class, are associated with instances of the class. This means each instance of the class has its own copy of the variable. For better memory management, Java provides static variables. Such a variable is declared using the `static` keyword.

Unlike an instance variable that belongs to instance of a class, a static variable belongs to the class. A single copy of a static variable is shared by all instances of the class. Therefore, even after an instance of a class is destroyed, the static variable remains available to other instances of that class. A static variable is initialized only once, at the start of the execution, and the initialization happens before the initialization of any instance variables.

As a static variable belongs to a class, it can be accessed directly by the class name without the need of instantiating the class.

Code Snippet 13 shows how to create and use a static variable.

Code Snippet 13:

```
package com.classdesign.demo;
public class StaticDemo {
    static String message = "Hello! Message from static variable";
    public static void main(String[] args) {
        String msg = StaticDemo.message;
    }
}
```

The code creates a static variable of type `String` and accesses it from the `main()` method. Observe that no new instance is created using the `new` keyword to access the static variable.

The Java class library has many classes with static fields. For example, the `Math` class defines PI as:

```
public static final double PI
```

14.5.2 Static Methods

The `static` keyword can also be applied to methods of a class. Such a method, known as static method, belongs to the class and not to the any instance of the class. A static method can only access other static variables and call other static methods. Also, a static method cannot refer to the `this` and `super` keywords. Similar to static variables, a static method can be accessed directly by the class name.

Note: As the `main()` method of a class acts as the entry point of a Java program, the JVM needs to call it ahead of creating any instance. Therefore, the `main()` method is always declared as static.

Code Snippet 14 shows how to create and use a static method.

Code Snippet 14:

```
package com.classdesign.demo;
public class StaticDemo {
    private static String message = "Hello! Message from static
variable";
    public static String getMessage() {
        return StaticDemo.message;
    }
    public static void main(String[] args) {
        String msg = StaticDemo.getMessage();
        System.out.println(msg);
    }
}
```

The code creates a private static variable of type String and returns it from the static `getMessage()` method. The `main()` method, which itself is a static method, calls the static `getMessage()` method.

14.5.3 Static Blocks

A static block, also known as static initialization block, is a normal block of code enclosed in curly braces, {}, and marked with the `static` keyword. There can be multiple static blocks inside the class body. For multiple static blocks, the blocks are called in the order that they appear in the class body. The JVM executes the code of static blocks when it loads the class. When there are multiple static blocks, the JVM joins them into one single static block before executing it. Code inside a static block can refer static variables and methods.

Note: As a static block cannot be accessed outside the class, it is functionally equivalent to a private static class.

Code Snippet 15 shows how to create and use a static block.

Code Snippet 15:

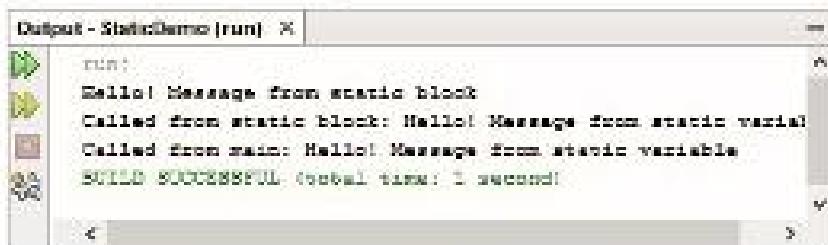
```
package com.classdesign.demo;
public class StaticDemo {
    private static String message = "Hello! Message from static
variable";
    public static String getMessage() {
        return StaticDemo.message;
    }
    public static void main(String[] args) {
```

```

String msg = StaticDemo.getMessage();
System.out.println("Called from main: "+msg);
}
static {
System.out.println("Hello! Message from static
block");
System.out.println("Called from static block: " +
StaticDemo.getMessage());
}
}
  
```

The code creates a private static variable of type `String` and returns it from the `static getMessage()` method. The `main()` method, which itself is a static method calls the `static getMessage()` method. The static block at the end of the class prints out a message and the output returned by a call to the `getMessage()` method.

Figure 14.7 displays the output on executing the `StaticDemo` class.



```

Output - StaticDemo [run] X
run:
Hello! Message from static block
Called from static block: Hello! Message from static variable
Called from main: Hello! Message from static variable
BUILD SUCCESSFUL. Total time: 1 second!
  
```

Figure 14.7: StaticDemo – Output

In the output, observe that the static block is executed ahead of the `main()` method.

14.5.4 Static Imports

A language enhancement introduced in Java 5 is static import. Prior to static imports, a static variable or method was required to be accessed by prefixing the class name separated by a dot (.) symbol. Starting from Java 5, static import allows importing static variables and methods of a class and use them, as they are declared in the same class. Static imports can greatly reduce code size by allowing to use static variables and methods of another class without prefixing the class name.

Code Snippet 16 shows the use of static import.

Code Snippet 16:

```
package com.classdesign.demo;

import static java.lang.Integer.MAX_VALUE;

public class StaticImportDemo {
  public static void main(String[] args) {
    /*Accessing static member with static import*/
    System.out.println(MAX_VALUE);
    /*Accessing static member without static import*/
    System.out.println(Integer.MAX_VALUE);
  }
}
```

The code performs a static import of the MAX_VALUE static variable of Integer. The main() method prints out the value of MAX_VALUE both with and without static imports.

Figure 14.8 displays the output on executing the StaticImportDemo class.

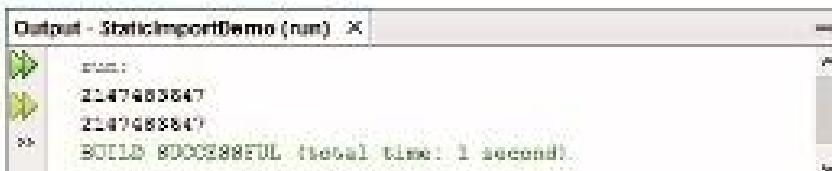


Figure 14.8: StaticImportDemo – Output

Although static imports reduce code size, many programmers prefer using the traditional class name prefix instead of static imports for increased readability. For example, both the Integer and Long classes contain the static MAX_VALUE variable. With static import, although it is possible to refer the variable directly as MAX_VALUE instead of Integer.MAX_VALUE, it is unclear of which class the variable belongs to. Therefore, it is recommended to use static imports only when frequent access to static members from one or two classes are required.

14.5.5 Final Class

In an inheritance hierarchy, a class can be prevented from being subclassed by marking it with the final keyword. Such a class is known as a final class. Typically, a class is declared as final to confer security and efficiency benefits. Another reason to declare a class as final is to standardize the behavior of a class by preventing it from being extended. This is because if a class is extended it can be made to behave differently. Hence, to avoid this, a class can be declared final.

Several Java classes, such as the System and String classes of the java.lang package, are declared as final.

The syntax for declaring a final class is as follows:

Syntax:

```
<access_modifier> final <class_name>
```

Code Snippet 17 shows a final class.

Code Snippet 17:

```
package com.classdesign.demo;
final class FinalClass{
}
```

The code creates a final class. If any other class attempts to extend FinalClass, the compiler will report a 'cannot inherit from final FinalClass' error.

14.5.6 Final Variables and Methods

Similar to classes, variables and methods can also be declared as final. A variable declared as final cannot be assigned a different value. In Java, a constant that is used to map an exact and unchanging value to a variable name is declared as final.

A method declared as final cannot be overridden in a subclass.

The syntax for declaring a final variable is as follows:

Syntax:

```
<access_modifier> final <variable_name> = <value>;
```

The syntax for declaring a final method is as follows:

Syntax:

```
<access_modifier> final <return_type> <method_name>(<parameter_optional>)
{ }
```

Code Snippet 18 shows a final variable and a final method.

Code Snippet 18:

```
package com.classdesign.demo;
class FinalMemberDemo {
final int initialCount=10;
final void getMessage() {
System.out.println("Message from final method");
}
}
```

The code creates a `initialCount` final variable and a `getMessage()` final method. For an attempt to assign a new value to `initialCount`, the compiler will report a 'cannot assign a value to final variable `initialCount`' error.

For an attempt to override the `getMessage()` method in a subclass, the compiler will report a 'cannot override `getMessage()` in `FinalMemberDemo` overridden method is final' error.

14.6 Inner Classes

You can also define a class within another class or interface. Such a class defined within the body of another class or interface is known as inner or nested class. Java supports nesting of classes. The different types of inner classes are as follows:

- Member inner class
- Anonymous inner class
- Local inner class
- Static nested class

14.6.1 Member Inner Class

A non-static class defined inside a class but outside a method is known as a member inner class.

Syntax for declaring a member inner class is as follows:

Syntax:

```
class <outer_class>{
    //code
    class <inner_class>{
        //code
    }
}
```

When you compile a class containing an inner class, the compiler creates two class files. To instantiate the inner class, the instance of the outer class is required. The outer class instance holds the inner class instance.

Code Snippet 19 shows the use of a member inner class.

Code Snippet 19:

```
package com.classdesign.Demo;

public class Outer {
    private String message = "Hello from outer class.";

    class Inner {
        void getMessage() {
            System.out.println(message);
            System.out.println("Hello from inner class.");
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Inner inner = outer.new Inner();
        inner.getMessage();
    }
}
```

The code creates a top-level class with a member inner class. Observe that the member inner class is accessing the private variable of the enclosing outer class. The `main()` method creates an outer class instance and uses it to create an inner class instance. The `getMessage()` method is then called on the inner class instance.

Figure 14.9 displays the output on executing the nested classes.



The screenshot shows the Eclipse IDE's Output window titled "Output - InnerClassDemo [run]". It contains the following text:
 12:21: Hello from outer class.
 12:21: Hello from inner class.
 >> BUILD SUCCESSFUL (total time: 1 second)

Figure 14.9: Member Inner Class – Output

14.6.2 Anonymous Inner Class

An inner class without a class name is known as an anonymous inner class. An anonymous inner class is declared and instantiated at the same time. Inner classes are typically used to override the method of a concrete class, abstract class, or interface.

The syntax of an anonymous inner class is as follows:

Syntax:

```
<extended_type> <variable_name> = new < extended_type > () {
  <access_modifier> <return_type> <> () {
    .....
    .....
  }
};
```

In the syntax, `extended_name` is the name of the parent class or interface whose method is overridden by the inner class.

Code Snippet 20 shows the use of an anonymous inner class.

Code Snippet 20:

```
package com.classdesign.demo;

abstract class Greet {
  abstract void getGreeting();
}

public class AnonymousInnerClass {
  public static void main(String[] args) {
    Greet g = new Greet() {
      void getGreeting() {
        System.out.println("Hello from anonymous inner
class.");
      }
    };
    g.getGreeting();
  }
}
```

The code creates an abstract `Greet` class with an abstract `getGreeting()` method. The `main()` method of the class, `AnonymousInnerClass` uses an anonymous inner class to instantiate a `Greet` object by overriding the `getGreeting()` method. Finally, the `getGreeting()` method is invoked on the `Greet` object named `g`.

Figure 14.10 displays the output of AnonymousInnerClass.



```
Output - AnonymousInnerClass [run] ×
run:
Hello from anonymous inner class.
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 14.10: Anonymous Inner Class – Output

14.6.3 Local Inner Class

A class defined inside a method is known as a local inner class. Similar to local variables defined within a method, a local inner class is scoped within the method. Once the method exits, the local inner class instance stops to exist. Also, a local inner class can be instantiated only from within the method where it is declared.

The syntax of an anonymous inner class is as follows:

Syntax:

```
<access_modifier> <return_type> <method_name> () {
  class <local_inner_name> {
    //Code
  }
}
```

In the syntax, `method_name` is the method that contains the local inner class.

Code Snippet 21 shows the use of a local inner class.

Code Snippet 21:

```
package com.classdesign.demo;
class LocalInnerClassDemo{
void display() {
String msg="Hello";
class Local{
void getMessage(){ System.out.println("Message from
local inner class: "+msg);}
}
Local l=new Local();
l.getMessage();
```

```

public static void main(String args[]) {
    LocalInnerClassDemo obj = new LocalInnerClassDemo();
    obj.display();
}

```

The code creates a local inner class named Local inside the display() method. The display() method instantiates the local inner class and invokes the getMessage() method on it.

In the display() method, observe that the local inner class refers to the msg local variable. Prior to Java 8, a local inner class could only access local variables that are declared as final. The ability to access non-final local variables has been introduced in Java 8.

Figure 14.11 displays the output of the local inner class.

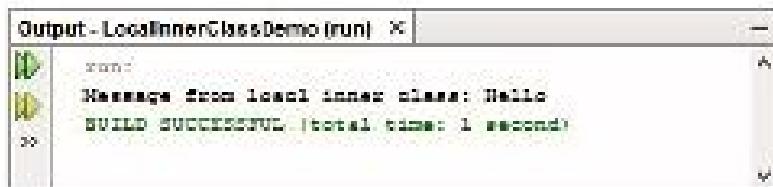


Figure 14.11: Local Inner Class – Output

14.6.4 Static Inner Class

An inner class that is a static member of the outer class is known as a static inner class. A static inner class is similar to a member inner class. The difference is that by being a static member of the outer class, a static inner class is created before an instance of the outer class. Also, a static inner class can be accessed directly with the class name of the outer class.

The syntax for declaring a static inner class is as follows:

Syntax:

```

class <outer_class>{
    //code
    static class <inner_class>{
        //code
    }
}

```

Code Snippet 22 shows the use of a static inner class.

Code Snippet 22:

```
package com.classdesign.demo;

public class Outer {
    static String msg="Hello";
    static class StaticInner {
        static void display() {
            System.out.print("Message from static inner class:
"+msg+"\n");
        }
    }
    public static void main(String[] args) {
        Outer.StaticInner.display();
    }
}
```

The code creates a static inner class named `StaticInner` within the `Outer` class. The `StaticInner` class contains a static `display()` method. The `main()` method calls the `display()` method of `StaticInner` without creating any instance of both the `Outer` and `StaticInner` classes.

Figure 14.12 displays the output of the static inner class.

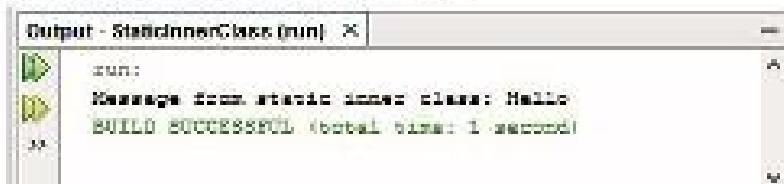


Figure 14.12: Static Inner Class – Output

14.7 Advanced Types

In addition to the regular types, such as top-level, abstract, and inner classes, Java supports advanced types, such as the `enum` type and immutable classes.

14.7.1 enum Type

A Java `enum` is a special type that defines a fixed set of constants. Similar to a Java class, an `enum` can contain constructors, variables, and methods. Internally, the compiler treats an `enum` as an advance type of class holding a set of constants. While compiling an `enum`, the compiler automatically adds some

special methods to the enum. One such method is the static values () method. This method returns the values of the enums in the order they are declared as an array and is useful to iterate over the values of an enum. Also, the compiler generates subclasses for each entry present in the enum. Therefore, an enum cannot be declared as final.

An enum is defined using the enum keyword. All enums implicitly extends the Enum class. Therefore, as Java does not support multi-inheritance, an enum cannot extend from another class.

The constant values of an enum are typed in uppercase letters. In addition, these values are implicitly static and final. Also, if an enum is used as a member of a class, it is implicitly defined as static.

Note: It is common for programmers to confuse between enum and the Enumeration interface because of the similarity in names. However, they are completely different. While enum defines a collection of constants, Enumeration is part of the java.util package allows iterating through the elements of a collection.

14.7.2 Immutable Classes

A class whose object's state cannot be changed once instantiated is called an immutable class. An immutable class is inherently thread safe. Their state cannot change once instantiated. Therefore, when working with immutable objects in multithreaded applications, synchronization overhead is not required. In addition, immutable objects can be easily shared or cached without the need to copy or clone them.

Some of the steps to consider for creating an immutable class are as follows:

- The class should be declared final to disable method overriding. Alternatively, the class constructor should be declared private and instances should be constructed through factory methods.
- The class should not contain setter methods to modify fields or objects referred to by fields.
- All fields should be declared private and final.
- Mutable objects that the instance fields refer to should not change.

Code Snippet 23 shows an immutable class.

Code Snippet 23:

```
package com.classdesign.demo;

public final class ImmutableClassDemo {
    private final String fName;
    private final String lName;
    public ImmutableClassDemo(final String fName, final String
        lName) {
        this.fName = fName;
```

```
this.lName = lName;  
}  
  
public String getFName () {  
    return fName;  
}  
  
public String getLName () {  
    return lName;  
}  
}
```

The code creates an immutable class through the following design:

- The class is declared as final.
- The instance variables are declared as private and final.
- The constructor parameters are declared as final to ensure that its value does not change.
- No setter methods, such as `setFName()` and `setLName()` methods are provided.

14.8 Check Your Progress

1. Which of the following access specifiers is used for the main() method?

(A) public	(C) Friendly or package
(B) protected	(D) private

2. Which of the following options correctly inherits from a superclass?

(A) class SubClass inherits SuperClass {}
(B) class SubClass inherits class SuperClass {}
(C) class SubClass extends SuperClass {}
(D) class SubClass extends class SuperClass {}

3. Which of the following statement is true regarding method overriding?

(A)	A method overriding another method have the same name, but differs in the parameter list.
(B)	A method overriding another method have different names, but the same parameter list.
(C)	A method overriding another method have the same name and parameter types, but different return types.
(D)	A method overriding another method have the same name, parameter types, and return types.

4. Which of the following code snippets correctly uses the `super()` constructor call?

(A)	<pre>public class SuperCallDemo extends SuperClass { public SuperCallDemo() { super(); } @Override void print() { } }</pre>
(B)	<pre>public class SuperCallDemo extends SuperClass { public SuperCallDemo() { super(); } @Override void print() { super(); } }</pre>
(C)	<pre>public class SuperCallDemo extends SuperClass { public SuperCallDemo() { } @Override void print() { super(); } }</pre>
(D)	<pre>public class SuperCallDemo extends SuperClass { super(); public SuperCallDemo() { } @Override void print() { } }</pre>

5. What will be the output of the following code snippet?

```
public class StaticDemo {  
    private static String message = "Hello!";  
    public static String getMessage(){  
        message="Hello! Hello!";  
        return StaticDemo.message;  
    }  
    public static void main(String[] args) {  
        message="Hello! Hello! Hello!";  
        String msg = StaticDemo.getMessage();  
        System.out.println(msg);  
    }  
    static {  
        System.out.println(StaticDemo.getMessage());  
    }  
}
```

(A)	Hello! Hello! Hello!	(C)	Hello! Hello! Hello! Hello! Hello!
(B)	Hello! Hello! Hello! Hello!	(D)	Hello! Hello! Hello!

14.8.1 Answers

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

Summary

- An access modifier controls the access of class members and variables by other objects.
- Inheritance enables a class to inherit variables and methods from another class.
- Polymorphism is the ability of different object types to respond to the same message, each one in its own way.
- In Java, a package is used to group classes and interfaces logically and prevent name clashes between those with identical names.
- An abstract class contains one or more methods declared with the `abstract` keyword.
- An inner class without a class name is known as an anonymous inner class.

Session - 15

Effective Programming with Lambda

Welcome to the Session, **Effective Programming with Lambda**.

This session describes lambda expressions in detail.

In this Session, you will learn to:

- Explain lambdas
- Identify the built-in functional interfaces
- Explain code refactoring for readability using lambdas
- Describe debugging of lambda



15.1 Lambda Usage

A lambda expression, or simply lambda, introduced in Java 8 is an unnamed block of code that facilitates functional programming. Lambdas simplify writing code and passing it around the application to be executed later.

15.1.1 Lambda Types

Based on how lambdas are written, lambdas can be categorized into two types. The most basic type of lambdas are object lambdas. An object lambda implements a functional interface and stored in a variable to be used later. Another type of lambdas are inline lambdas that are passed inline as parameters to methods.

Irrespective of how lambdas are written, lambdas can use inferred types, which allows passing parameters to the expression body without specifying the parameter types. Also, the expression body of a lambda can contain multiple statements and have a return type.

Code Snippet 1 creates the different types of lambdas and shows how to use them.

Code Snippet 1:

```
package lambdaexpressiondemo;

import java.util.Arrays;
@FunctionalInterface
interface FunctionalA {
    int doWork(int a, int b);
}

public class LambdaExpressionDemo {
    public static void main(String[] args) {
        /*Lambda 1: Using basic lambda */
        FunctionalA functionalA1 = (int num1, int num2) -> num1 +
            num2;
        System.out.println("5+5= " + functionalA1.doWork(5, 5));
        /*Lambda 2: Using lambda with inferred types */
        FunctionalA functionalA2 = (num1, num2) -> num1 + num2;
        System.out.println("5+10= " + functionalA2.doWork(5, 10));
        /*Lambda 3: Using lambda with expression body containing
         return statement */
        FunctionalA functionalA3 = (num1, num2) -> {
            int sum = num1 + num2;
            System.out.println("Sum= " + sum);
            return sum;
        };
        System.out.println("Sum= " + functionalA3.doWork(5, 10));
    }
}
```

```

        return num1 + num2;
    }

    System.out.println("5+15= " + functionalA3.doWork(5, 15));

/*Lambda 4: Using lambda with expression body containing
multiple statements*/
FunctionalA functionalA4 = (num1, num2) -> {
    int sum = num1 + num2;
    int result = sum * 10;
    return result;
}

System.out.println("(5+10)*10= " + functionalA4.doWork(5,
10));

/*Lambda 5: Passing lambda as method parameter to
Arrays.sort() method*/
String[] words = new String[]{"Hi", "Hello", "HelloWorld",
"Hi"};

System.out.println("Original array= " +
Arrays.toString(words));
Arrays.sort(words,
(first, second) -> Integer.compare(first.length(),
second.length()));

System.out.println("Sorted array by length using lambda= "
+ Arrays.toString(words));
}
}

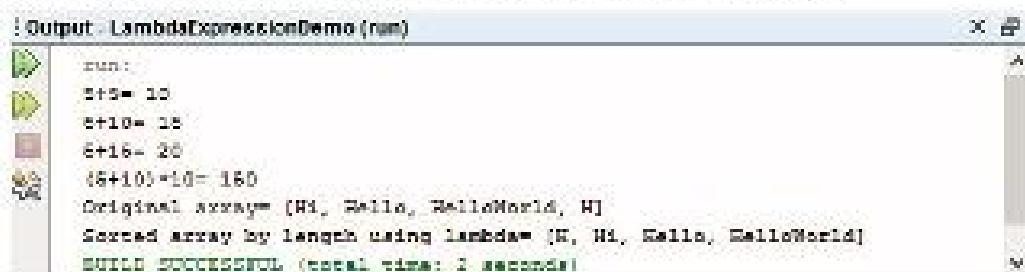
```

The code creates a functional interface, named FunctionalA. The code then uses several lambdas in the LambdaExpressionDemo class based on the FunctionalA functional interface. The lambdas are as follows:

- **Lambda 1:** Uses basic lambda syntax to assign the value of a lambda expression to a variable of type FunctionalA.

- **Lambda 2:** Performs the same function as Lambda 1 but without specifying the parameter types. The compiler infers the types by looking at the parameter types declared in the corresponding abstract method of the Functional API functional interface.
- **Lambda 3:** Performs the same function as Lambda 1 but with an explicit return statement enclosed within curly braces {}.
- **Lambda 4:** Uses multiple statements in the lambda expression body before returning the final value to the caller.
- **Lambda 5:** Passes a lambda as the second parameter to the Arrays.sort(T[] a, Comparator<? super T> c) method that sorts the specified array of objects according to the order induced by the specified comparator.

Figure 15.1 displays the output of the LambdaExpressionDemo class.



```
! Output - LambdaExpressionDemo (run)
run:
  s+5= 10
  s+10= 15
  s+15= 20
  (s+10)*10= 150
Original array= [Hi, Hello, HelloWorld, H]
Sorted array by length using lambda= [H, Hi, Hello, HelloWorld]
Build SUCCESSFUL (total time: 1 seconds)
```

Figure 15.1: LambdaExpressionDemo - Output

15.1.2 Built-in Functional Interfaces

Java 8 comes with a large number of built-in functional interfaces as part of the new `java.util.function` package.

Some of the functional interfaces are listed in table 15.1.

Interface	Abstract Method	Description
Predicate<T>	boolean test (T t)	Represents an operation that checks a condition and returns a boolean value as result
Consumer<T>	void accept (T t)	Represents an operation that takes an argument but returns nothing
Function<T, R>	R apply (T t)	Represents an operation that takes an argument and returns some result to the caller
Supplier<T>	T get()	Represents an operation that does not take any argument but returns a value to the caller

Table 15.1: Core Functional Interfaces

Code Snippet 2 shows the use of the common built-in functional interfaces.

Code Snippet 2:

```
package lambdaexpressiondemo;

import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;
public class FunctionalInterfacesDemo {

    static void testPredicate() {
        Predicate<String> result = arg -> {arg.equals("Hello Lambda")};
        String testStr = "Hello Lambda";
        System.out.println(result.test(testStr));
    }

    static void testConsumer() {
        Consumer<String> result = str -> System.out.println(str.toUpperCase());
        result.accept("hello lambda");
    }

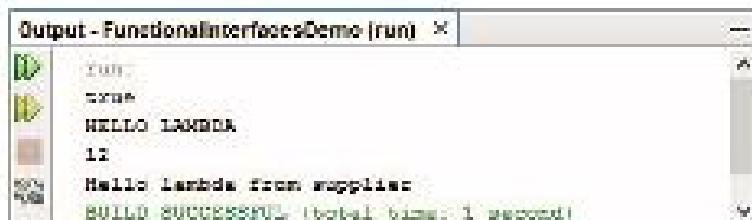
    static void testFunction() {
        Function<String, Integer> result = str -> str.length();
        System.out.println(result.apply("Hello Lambda"));
    }

    static void testProducer() {
        Supplier result = () -> {return "Hello lambda from supplier";};
        System.out.println(result.get());
    }

    public static void main(String[] args) {
        testPredicate();
        testConsumer();
        testFunction();
        testProducer();
    }
}
```

The `testPredicate()` method creates a lambda as the implementation of the boolean `test(T t)` function of the `Predicate` functional interface. The lambda checks whether the parameter passed to it is `Hello Lambda` and accordingly returns a Boolean value. Next, the `testConsumer()` method creates a lambda as the implementation of the void `accept(T t)` function of the `Consumer` functional interface. This lambda accepts a string as a parameter, converts the string to upper case, and prints the string out to the console. The `testFunction()` method creates a lambda for the `R apply(T t)` function of the `Function` functional interface. This lambda accepts a string, calculates its length, and returns the calculated length as an int value. Finally, the `testProducer()` uses lambda for the `T get()` method of `Supplier`. This lambda simply returns back a string to the caller.

Figure 15.2 displays the output of the `FunctionalInterfacesDemo` class.



```
Output - FunctionalInterfacesDemo [run] ×
[1]  true
[2]  HELLO LAMBDA
[3]  12
[4]  Hello Lambda from supplier
[5]  BUILD SUCCESSFUL (total time: 1 second)
```

Figure 15.2: FunctionalInterfacesDemo - Output

15.1.3 Primitive Versions of Functional Interfaces

The functional interfaces `Predicate<T>`, `Consumer<T>`, `Function<T, R>`, and `Supplier<T>` are generic and therefore, operate on reference type objects. Primitive values, such as `int`, `long`, `float`, `double` cannot be used with them. Therefore, Java 8 provides primitive versions for such functional interfaces. For example, `IntPredicate`, `LongPredicate`, and `DoublePredicate` are primitive versions of the `Predicate` interface. Similarly, `IntConsumer`, `LongConsumer`, and `DoubleConsumer` are primitive versions of the `Consumer` interface.

Code Snippet 3 shows the use of the primitive versions of the `Predicate` and `Consumer` functional interfaces.

Code Snippet 3:

```
package lambdaexpressiondemo;
import java.util.function.IntPredicate;
import java.util.function.LongConsumer;
public class PrimitiveFunctionalInterfacesDemo {
    static void testIntPredicate() {
        IntPredicate result = arg -> (arg == 10);
        System.out.println("IntPredicate.test() result: " + result.test(11));
    }
}
```

```

static void testLongConsumer() {
    LongConsumer result = val ->
        System.out.println("LongConsumer.accept() result: "+val*val);
    result.accept(1000000);
}

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

The code uses the IntPredicate primitive version of the Predicate interface in the testIntPredicate() method to test if the supplied int parameter is equal to 10. The LongConsumer primitive version of the Consumer interface squares the supplied long parameter in the testLongConsumer() method.

Figure 15.3 displays the output of the PrimitiveFunctionalInterfacesDemo class.



Figure 15.3: PrimitiveFunctionalInterfacesDemo - Output

15.1.4 Binary Versions of Functional Interfaces

The abstract methods of the Predicate, Consumer, and Function functional interfaces accept one argument. Java 8 provides equivalent binary versions of such functional interfaces that can accept two parameters. The binary functional version interfaces are prefixed with Bi, such as BiPredicate, BiConsumer, and BiFunction.

Note: As the abstract T get() method of Supplier does not take any argument, there is no binary equivalent of Supplier.

Code Snippet 4 shows the use of the binary versions of the Predicate and Consumer functional interfaces.

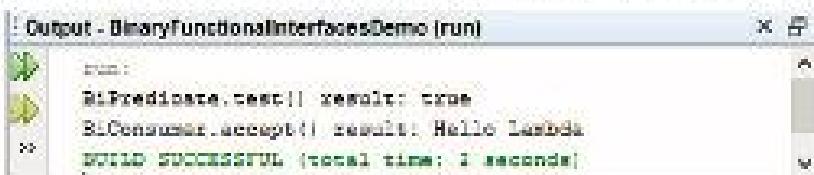
Code Snippet 4:

```
package lambdaexpressiondemo;

import java.util.function.BiPredicate;
import java.util.function.BiConsumer;
public class BinaryFunctionalInterfacesDemo {
  static void testBiPredicate() {
    BiPredicate<Integer, Integer> result = (arg1, arg2) -> arg1
      < arg2;
    System.out.println("BiPredicate.test() result: "
      "+result.test(5, 10));
  }
  static void testBiConsumer() {
    BiConsumer<String, String> result = (arg1, arg2) ->
    System.out.println("BiConsumer.accept() result: "+arg1+arg2);
    result.accept("Hello ", "Lambda");
  }
  public static void main(String[] args) {
    testBiPredicate();
    testBiConsumer();
  }
}
```

The code uses the BiPredicate binary version of the Predicate interface in the testBiPredicate() method to test if the first Integer parameter is less than the second one. In the testBiConsumer() method, the BiConsumer primitive version of the Consumer interface adds and prints the two string parameters.

Figure 15.4 displays the output of the BinaryFunctionalInterfacesDemo class.



```
Output - BinaryFunctionalInterfacesDemo [run]
x ×
run:
BiPredicate.test() result: true
BiConsumer.accept() result: Hello Lambda
BUILD SUCCESSFUL (total time: 1 seconds)
```

Figure 15.4: BinaryFunctionalInterfacesDemo - Output

15.1.5 UnaryOperator Interface

The `java.util.function` package contains a `UnaryOperator` functional interface that is a specialized version of the `Function` interface. `UnaryOperator` can be used on a single operand when the types of the operand and result are the same.

Code Snippet 5 shows the use of `UnaryOperator`.

Code Snippet 5:

```
package lambdaexpressiondemo;

import java.util.function.UnaryOperator;

public class UnaryOperatorDemo {

    public static void main(String[] args) {
        UnaryOperator<String> result = (x) -> x.toUpperCase();
        System.out.println("Output converted into uppercase:");
        System.out.println(result.apply("Hello Lambda"));
    }
}
```

The code uses a `UnaryOperator` as the assignment target for a lambda. The lambda expression accepts a string parameter, converts it into upper case, and returns the result as a string. The `apply()` method that `UnaryOperator` inherits from `Function` executes the lambda.

Figure 15.5 displays the output of the `UnaryOperatorDemo` class.



Figure 15.5: UnaryOperatorDemo - Output

15.2 Refactoring for Improved Readability

Lambda introduced in Java 8 can greatly increase readability of code. Java programmers can use lambdas to express problems in many situations in a shorter and more readable way than it was possible before. The introduction of lambdas does not break code. Existing code can run as it is with new code containing lambdas running alongside. However, developers might want to refactor their existing code to use the more convenient lambdas. Typically, refactoring will be done to remove existing boilerplate code and make the existing code more concise.

15.2.1 Refactoring Runnable Code

In multithread applications, one way to implement a new thread is to create a `Runnable` object and call its `run()` object. Prior to Java 8, it was achieved through an anonymous class.

Code Snippet 6 shows the use of anonymous class to run a piece of code in a different thread with `Runnable`.

Code Snippet 6:

```
package lambdaexpressiondemo;

public class MultiThreadedAnonymousDemo {
  public static void main(String[] args) {
    Runnable r1 = new Runnable() {
      @Override
      public void run() {
        System.out.println("Hello from anonymous");
      }
    };
    r1.run();
  }
}
```

`Runnable` contains a single abstract method, `void run()` and so is as a functional interface. For increased readability, a lambda can be used to represent the code to execute. The lambda can then be assigned to a `Runnable` instance variable to be executed later through a call to the `run()` method.

Code Snippet 7 shows the use of lambda to write a `Runnable` instance and run it.

Code Snippet 7:

```
package lambdaexpressiondemo;

public class MultiThreadedLambdaDemo {
  public static void main(String[] args) {
    Runnable r1 = () -> {
      System.out.println("Hello from lambda");
    };
    r1.run();
  }
}
```

15.2.2 Refactoring Comparison Code

The Comparator interface enables comparing the elements of a collection that need to be sorted. Comparator is a functional interface that contains the single int compare(T o1, T o2) method. When a collection or array needs to be sorted, a Comparator object is passed to the Collections.sort() or Arrays.sort() method. Prior to Java 8, a Comparator can be passed to the sort() method as an anonymous class.

Code Snippet 8 applies a Comparator using an anonymous inner class to sort a List of objects.

Code Snippet 8:

```
...
Collections.sort(employeeList, new Comparator<Employee>() {
    @Override
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getLastName().compareTo(emp2.getLastName());
    }
});
System.out.println("==> Sorted Employees by last name in ascending
order ==>");
for (Employee emp : employeeList) {
    System.out.println(emp.getFirstName() + " " + emp.getLastName());
}
...
}
```

For greater readability, you can use a lambda instead of the inner class to pass the comparison code to the sort() method.

Code Snippet 9 lists the complete code that applies a Comparator using a lambda to sort a List of Employee objects.

Code Snippet 9:

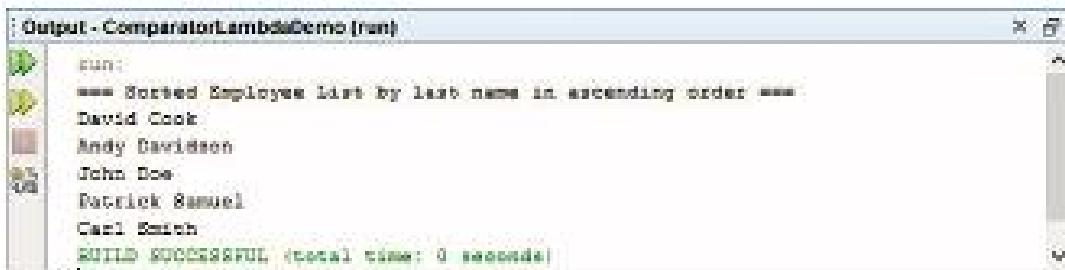
```
package lambdaexpressiondemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
class Employee {
    private String firstName;
```

```

private String lastName;
public Employee(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
public String getFirstName() {
    return firstName;
}
public String getLastName() {
    return lastName;
}
public class ComparatorLambdaDemo {
    public static void main(String[] args) {
        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(new Employee("Patrick", "Samuel"));
        employeeList.add(new Employee("John", "Doe"));
        employeeList.add(new Employee("Andy", "Davidson"));
        employeeList.add(new Employee("Carl", "Smith"));
        employeeList.add(new Employee("David", "Cook"));
        Comparator<Employee> sortedEmployee = (Employee empl,
            Employee emp2) -> empl.getLastName()
            .compareTo(emp2.getLastName());
        System.out.println("----Sorted Employee List by last name in ascending order
----");
        Collections.sort(employeeList, sortedEmployee);
        for (Employee emp : employeeList) {
            System.out.println(emp.getFirstName() + " " +
                emp.getLastName());
        }
    }
}
  
```

The code uses a lambda to compare two Employee objects based on the lastName field and stores it in a Comparator variable. The sort() method sorts the Employee list with the Comparator passed to it.

Figure 15.6 displays the output of the ComparatorLambdaDemo class.



```
Output - ComparatorLambdaDemo [run]
run:
*** Sorted Employee list by last name in ascending order ***
David Cook
Andy Davidsen
John Doe
Patrick Samuel
Carl Smith
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 15.6: ComparatorLambdaDemo - Output

15.2.3 Refactoring Concurrency Code

Callable and Future are extensively used in multithread Java applications to implement asynchronous processing. Callable is similar to Runnable in that both can be used to create a task which can be executed by threads in parallel. However, unlike Runnable that cannot return a value, Callable can return a value. In addition, the call() method of Callable can throw checked exception, which is not possible for the run() method of Runnable.

Callable is a functional interface in the `java.util.concurrent` package. The Callable interface have a single `V call()` abstract method. When a Callable is passed to a thread pool maintained by ExecutorService, the pool selects a thread and execute the Callable. On execution, the thread returns a Future object that holds the result of computation once done. The `get()` method of Future returns the computation result or block if the computation is not complete.

Code Snippet 10 shows the use of an anonymous class to run a piece of code in a different thread with Callable and Future.

Code Snippet 10:

```
...
ExecutorService executor = Executors.newFixedThreadPool(5);

Callable callable = new Callable() {
  @Override
  public String call() {
    try {
      Thread.sleep(10);
      return Thread.currentThread().getName();
    }
  }
}
```

```

    catch(InterruptedException ie) {
        ie.printStackTrace();
    }
    return Thread.currentThread().getName();
}
}

Future<String> future = executor.submit(callable);
...
  
```

Code Snippet 11 lists the complete code refactored to construct a Callable using lambda instead of an anonymous and to submit the Callable to an ExecutorService.

Code Snippet 11:

```

package lambdaexpressiondemo;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableLambdaDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        List<Future<String>> list = new ArrayList<>();
        Callable callable = () -> {
            try {
                Thread.sleep(10);
                return Thread.currentThread().getName();
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
            return Thread.currentThread().getName();
        };
        ...
      
```

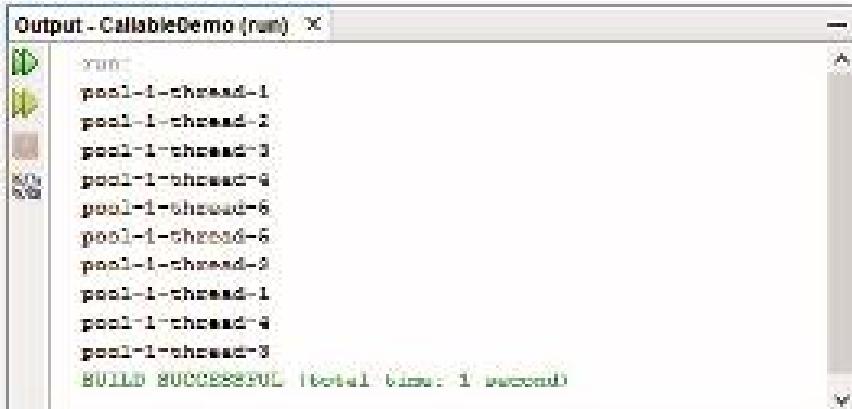
```
};

for (int i=0;i<10; i++) {
    Future<String> future = executor.submit(callable);
    list.add(future);
}

for (Future<String> future : list) {
    try {
        System.out.println(future.get());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

executor.shutdown();
}
```

Figure 15.7 displays the output of the CallableLambdaDemo class.



The screenshot shows the 'Output' window of an IDE during the execution of the 'CallableLambdaDemo' class. The window title is 'Output - CallableDemo (run)'. The output content is as follows:

```
pool-1-thread-1
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 15.7: CallableLambdaDemo - Output

15.3 Debugging Lambdas

You can debug lambdas like any other piece of Java code.

Code Snippet 12 shows a Java class with lambda that you can debug in NetBeans.

Code Snippet 12:

```
package lambdaexpressiondemo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Employee {
    private String firstName;
    private String lastName;
    public Employee(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
}

public class ComparatorLambdaDemo {
    public static void main(String[] args) {
        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(new Employee("Patrick", "Samuel"));
        employeeList.add(new Employee("John", "Doe"));
        employeeList.add(new Employee("Andy", "Davidson"));
        Comparator<Employee> sortedEmployee = (Employee empl,
```

```

Employee emp2) -> empl.getLastname() .compareTo(emp2.getLastname());
System.out.println("Sorted Employee by last name in ascending order");
Collections.sort(employeeList,sortedEmployee);
for (Employee emp : employeeList) {
  System.out.println(emp.getFirstName() + " " +
    emp.getLastName());
}
}
}
}
  
```

The code uses a lambda to compare two `Employee` objects based on the `lastName` field and stores it in a `Comparator` variable.

To test the lambda used in the code:

1. Open the `ComparatorLambdaDemo` class in NetBeans.
2. In the code editor, double-click the line number of the statement that uses lambda to set a breakpoint.
3. In the code editor, double-click the line number of the statement containing the `for` loop to set a breakpoint.

Figure 15.8 displays the breakpoints set on the `ComparatorLambdaDemo` class.



Figure 15.8: Breakpoints to Debug Lambda

4. Select **Debug → Debug Project** from the main menu of NetBeans. The program execution stops in the first breakpoint.
5. Observe the first name and last name values in the **Variables** window displayed. At this point, the lambda is yet to perform the sorting.

Figure 15.9 displays the employee values before sorting by lambda.

Variables	Type	Value
list	java.util.List	[Employee@400940, Employee@400940, Employee@400940, Employee@400940, Employee@400940, Employee@400940]
list[0]	Employee	Andy
list[1]	Employee	David
list[2]	Employee	Tom
list[3]	Employee	Bob
list[4]	Employee	Tom
list[5]	Employee	David

Figure 15.9: Employee Values before Sorting

6. Select Debug → Continue from the main menu to continue debugging until the debugging thread hits the second breakpoint.
7. Check the Variables window to ensure that the lambda has correctly performed the sorting based on the last name.

Figure 15.10 displays the employee values after sorting by lambda.

Variables	Type	Value
list	java.util.List	[Employee@400940, Employee@400940, Employee@400940, Employee@400940, Employee@400940, Employee@400940]
list[0]	Employee	Andy
list[1]	Employee	David
list[2]	Employee	Tom
list[3]	Employee	Bob
list[4]	Employee	Tom
list[5]	Employee	David

Figure 15.10: Employee Values after Sorting

8. Select Debug → Finish Debugger Session to stop debugging.

15.4 Check Your Progress

1. Which of the following annotations mark a functional interface that has a single abstract method?

(A) <code>@Functional</code>	(C) <code>@AbstractInterface</code>
(B) <code>@FunctionalInterface</code>	(D) <code>@Abstract</code>

2. Consider the following code snippet:

```
interface FunctionalA {
    int doWork(int a, int b);
}
```

Which of the following lambdas correctly provides implementation of the functional interface?

(A)	FunctionalA functionalA1 = (int num1, int num2) -> num1 + num2; System.out.println(functionalA1.doWork(5, 10));
(B)	FunctionalA functionalA1 = () -> num1 + num2; System.out.println(functionalA1.doWork(5, 10));
(C)	FunctionalA functionalA1 = ((int num1, int num2) -> num1 + num2); System.out.println(functionalA1.doWork(5, 10));
(D)	FunctionalA functionalA1 = (num1, num2) -> num1 + num2; System.out.println(functionalA1.doWork());

3. Which of the following statements is true regarding Predicate<T>?

(A)	Represents an operation that does not take any argument but returns a value to the caller
(B)	Represents an operation that takes an argument and returns some result to the caller
(C)	Represents an operation that takes an argument but returns nothing
(D)	Represents an operation that checks a condition and returns a boolean value as result

4. Which of the following is a binary functional interface?

(A)	BiDouble	(C)	BiBoolean
(B)	BiPredicate	(D)	BiInteger

5. What will be the result on compiling and executing the following code snippet?

```
import java.util.function.UnaryOperator;
public class LambdaDemo {
    public static void main(String[] args) {
        UnaryOperator<String> result = () -> x.toUpperCase();
        result.apply("Hello");
    }
}
```

(A)	The code will display an output Hello	(C)	The code will report a compilation error
(B)	The code will display an output HELLO	(D)	There will be no output

6. Which of the following code snippets correctly uses a lambda to execute a Runnable?

(A)	<pre>public static void main(String[] args) { Runnable r1 = () -> { System.out.println("Hello"); }; r1.run(); }</pre>
(B)	<pre>public static void main(String[] args) { Runnable r1 = new Runnable() -> { System.out.println("Hello"); }; r1.run(); }</pre>
(C)	<pre>public static void main(String[] args) { Runnable r1 = (String msg) -> { System.out.println("Hello"); }; r1.run(); }</pre>
(D)	<pre>public static void main(String[] args) { Runnable r1 = ({String msg} -> System.out.println("Hello")); r1.run(); }</pre>

15.4.1 Answers

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

Summary

- A lambda expression is an unnamed block of code that facilitates functional programming.
- The `java.util.function` package introduced in Java 8 contains a large number of functional interfaces.
- Java 8 provides primitive versions for functional interfaces to operate on primitive values.
- Java 8 provides equivalent binary versions of some functional interfaces that can accept two parameters.
- `UnaryOperator` is used on a single operand when the types of the operand and result are the same.
- Java programmers can use lambdas to express problems in a shorter and more readable way.
- Lambda expressions can be debugged in NetBeans like any piece of Java code by setting breakpoints.

Session - 16

Java Data Structures

Welcome to the Session, **Java Data Structures**.

This session explores some of the legacy data structures in Java such as `BitSet`, `Dictionary`, and others.

In this Session, you will learn to:

- Explain the `Enumeration` interface
- Describe the `BitSet` class
- Describe the `Stack` classes
- Explain the `Dictionary` classes



16.1 Enumeration

Enumeration is an interface in the `java.util` package that defines methods to iterate through the elements of a collection. The methods of the Enumeration interface are as follows:

- ➔ `hasMoreElements()`: Checks whether or not the enumeration contains more elements.
- ➔ `nextElement()`: Returns the next element, if present, in the enumeration.

The exception associated with this interface is `NoSuchElementException`. This exception is raised if `nextElement()` is called when there are no more elements in the enumeration.

Code Snippet 1 uses an Enumeration to iterate through the elements of an array.

Code Snippet 1:

```
package com.datastructures.demo;

import java.lang.reflect.Array;
import java.util.Enumeration;
public class CustomEnumeration implements Enumeration {
private final int arraySize;
private int arrayCursor;
private final Object array;
public CustomEnumeration (Object obj) {
arraySize = Array.getLength (obj);
array = obj;
}
@Override
public boolean hasMoreElements () {
return (arrayCursor < arraySize);
}
@Override
public Object nextElement () {
return Array.get (array, arrayCursor++);
}
```

The code creates a `CustomEnumeration` class that implements the `Enumeration` interface. The class constructor accepts an `Object` and stores its size and the object itself to the `arraySize` and `array` variables respectively. The overridden `hasMoreElements()` method returns true if the `arrayCursor` variable is less than the `arraySize` variable. The `nextElement()` overridden method returns an `Object` that represents an element in the array with the index specified by the `arrayCursor` variable.

Code Snippet 2 shows a class that uses the custom enumeration defined in Code Snippet 1.

Code Snippet 2:

```
package com.datastructures.demo;

import java.util.Enumeration;

public class EnumerationDemo {

  public static void main(String[] args) {
    String[] strArray = new String[]{"One", "Two", "Three"};
    Enumeration customEnumeration = new
      CustomEnumeration(strArray);
    while (customEnumeration.hasMoreElements()) {
      System.out.println(customEnumeration.nextElement());
    }
  }
}
```

The `main()` method in the code creates a `CustomEnumeration` instance initialized with an array. The `hasMoreElements()` method in the while loop checks for more elements in the array. If `hasMoreElements()` method returns true, the `nextElement()` method prints out the array element.

Figure 16.1 displays the output of the `EnumerationDemo` class.



Figure 16.1: `EnumerationDemo` – Output

Though these codes used a very basic set of data with just three values, in practical scenarios, enumerations can be used for large volume data sets too. However, in recent versions of Java, `Enumeration` is considered as legacy and retained only for backward compatibility. It is recommended to use the `Iterator` interface instead of `Enumeration`.

16.2 BitSet Class

In computing, a bit is a smallest unit of data. Java applications often need to work with large volumes of bits and `BitSet` class is designed for that purpose. The `BitSet` class is a collection of bit values and can resize as needed. A `BitSet` indexes its bits with non-negative integer and each bit in a `BitSet` can be accessed with its index. The advantage of `BitSet` is that it uses only one bit to store a value.

Code Snippet 3 shows the use of `BitSet` class.

Code Snippet 3:

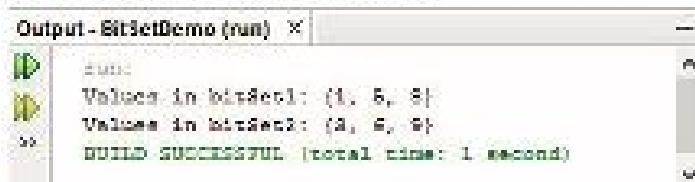
```
package com.datastructures.deno;

import java.util.BitSet;

public class BitSetDemo {
  public static void main(String[] args) {
    BitSet bitSet1 = new BitSet();
    BitSet bitSet2 = new BitSet();
    bitSet1.set(1);
    bitSet1.set(5);
    bitSet1.set(8);
    bitSet2.set(3);
    bitSet2.set(6);
    bitSet2.set(9);
    System.out.println("Values in bitSet1:
"+bitSet1+"\nValues in bitSet2: "+bitSet2);
  }
}
```

The code creates two `BitSet` objects and calls the `set()` method to initialize them. Finally, the `BitSet` objects are printed out.

Figure 16.2 displays the output of the BitSetDemo class.



```
Output - BitSetDemo (run) ×
Run
Values in bitSet1: {1, 5, 8}
Values in bitSet2: {3, 6, 9}
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 16.2: BitSetDemo - Output

As of recent versions of Java, this class is legacy and not much used. It is retained for backward compatibility.

16.3 Stack Class

A Stack is a collection of objects based on the Last-in First-out (LIFO) principle. In a Stack, the last element added, or pushed to the stack is the first element to be removed, or popped out of the stack. The Stack class extends the Vector class and in addition to the inherited methods of Vector, Stack defines five methods, as listed in table 16.1.

Method	Description
empty()	Checks whether or not the Stack is empty.
peek()	Returns the object at the top of the Stack without removing the object.
pop()	Returns the object at the top of the Stack after removing the object from the Stack.
push(E item)	Pushes an object onto the top of this Stack.
search(Object o)	Returns the position of an object from the top of the Stack. This method returns 1 for the object at the top of the Stack, 2 for the object below it, and so on. If an object is not found, this method returns -1.

Table 16.1: Methods of the Stack Class

Code Snippet 4 shows the use of the Stack class.

Code Snippet 4:

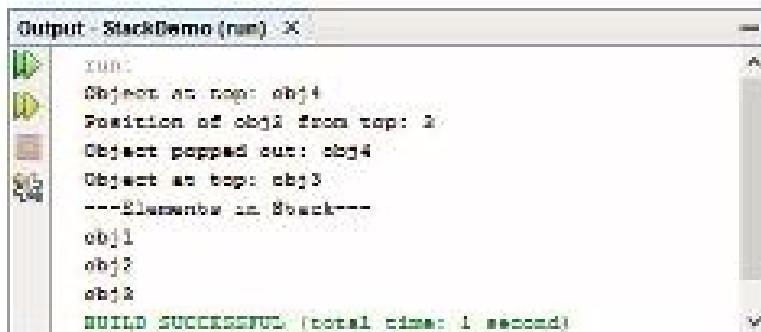
```
package com.datastructures.demo;
import java.util.Stack;
public class StackDemo {
private static Stack getInitializedStack() {
Stack stack = new Stack();
stack.push("obj1");
stack.push("obj2");
return stack;
}
```

```
stack.push("obj3");
stack.push("obj4");
return stack;
}

public static void main(String[] args) {
    Stack initializedStack = StackDemo.getInitializedStack();
    System.out.println("Object at top: " +
        initializedStack.peek());
    System.out.println("Position of obj2 from top: " +
        initializedStack.search("obj2"));
    System.out.println("Object popped out: " +
        initializedStack.pop());
    System.out.println("Object at top: " +
        initializedStack.peek());
    System.out.println("----Elements in Stack---");
    for (Object obj : initializedStack) {
        System.out.println(obj);
    }
}
```

The `getInitializedStack()` method of the code creates a `Stack` and pushes four objects to it. The `main()` method calls the `peek()` method to retrieve and print the object at the top of the `Stack`. The `search()` method obtains and prints the position of the `obj2` object from the top. The `pop()` method pops out the element at the top and the `peek()` method again obtains and prints the element currently at the top. Finally, the enhanced `for` loop prints each element present in the `Stack`.

Figure 16.3 displays the output of the StackDemo class.



```
Output - StackDemo (run) X
[Run]
Object at top: obj4
Position of obj3 from top: 3
Object popped out: obj4
Object at top: obj3
---Elements in Stack---
obj1
obj2
obj3
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 16.3: StackDemo - Output

16.4 Dictionary Classes

In Java collections, a dictionary is used to store key-value pairs. Every key and value in a dictionary is an object. The `Dictionary` abstract class of the `java.util` package is the super class of all dictionary implementation classes. Examples of dictionary implementation classes are `Hashtable` and `Properties`.

16.4.1 Hashtable Class

The `Hashtable` class implements a collection of key-value pairs that are organized based on the hash code of the key. A hash code is a signed number that identifies the key. Based on the hash code, a key-value pair, when added to a `Hashtable`, gets stored into a particular bucket. A `Hashtable` is significantly faster as compared to other dictionaries. When a lookup is performed for a key, the `Hashtable` searches for the key in only one particular bucket. As a result, the number of key comparisons significantly reduces.

When elements are added to a `Hashtable`, the `Hashtable` automatically resizes itself by increasing its capacity. When the `Hashtable` capacity reaches the capacity that you specified during its creation, the number of buckets is automatically increased. Internally, the number of buckets is increased to the smallest prime number, which is larger than twice the current number of buckets in the `Hashtable` class. For example, if the current number of buckets is 5 and the `Hashtable` reaches its initial capacity, the number of buckets automatically increases to 11.

Code Snippet 5 shows the use of the `Hashtable` class.

Code Snippet 5:

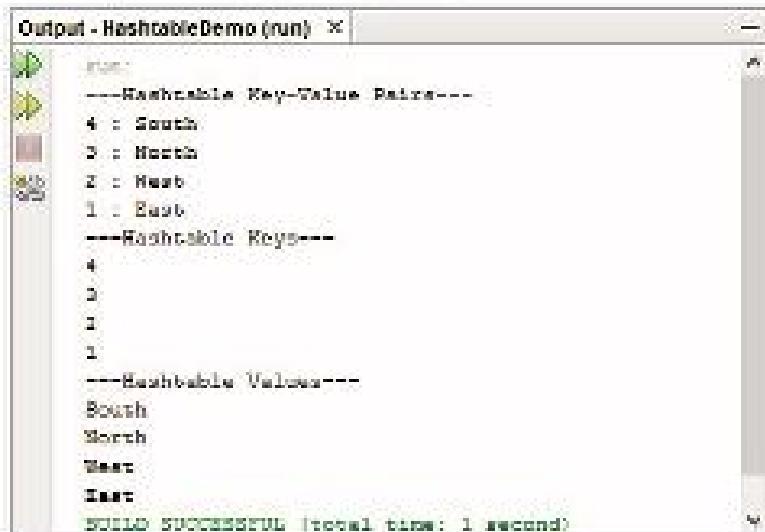
```
package com.datastructures.demo;
import java.util.Enumeration;
import java.util.Hashtable;
public class HashtableDemo {
    private static Hashtable initializeHashtable() {
```

```
Hashtable hTable = new Hashtable();
hTable.put("1", "East");
hTable.put("2", "West");
hTable.put("3", "North");
hTable.put("4", "South");
return hTable;
}

public static void main(String[] args) {
Hashtable initializedHtable =
HashtableDemo.initializeHashtable();
Enumeration e = initializedHtable.keys();
System.out.println("----Hashtable Key-Value Pairs----");
while (e.hasMoreElements()) {
String key = (String) e.nextElement();
System.out.println(key + " : " +
initializedHtable.get(key));
}
e = initializedHtable.keys();
System.out.println("----Hashtable Keys----");
while (e.hasMoreElements()) {
System.out.println(e.nextElement());
}
e = initializedHtable.elements();
System.out.println("----Hashtable Values----");
while (e.hasMoreElements()) {
System.out.println(e.nextElement());
}
}
```

The `getInitializedHashtable()` method of the code creates and initializes a `Hashtable` with key-value pairs. The `main()` method creates an `Enumeration` of the `Hashtable` with a call to the `keys()` method. The `get(key)` method retrieves the value of a particular key. The `elements()` method returns all the values of the `Hashtable` as an `Enumeration` object. The code uses the `Enumeration` object to print the key-value pairs, keys, and values of the `Hashtable`.

Figure 16.4 displays the output of the `HashtableDemo` class.



```
Output - HashtableDemo (run) X
-----
 4 : South
 3 : North
 2 : West
 1 : East
-----
 4
 3
 2
 1
-----
 South
 North
 West
 East
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 16.4: `HashtableDemo` - Output

16.4.2 Properties Class

The `Properties` class extends `Hashtable` to implement a collection of key-value pairs where both the types of the keys and values are `String`. Although being a `Hashtable` subclass, the `Properties` class inherits the `put()` method to add a key-value pair, you should avoid it. This is because, if you add a non-string key or value, the method will fail at runtime. Instead, you should use the `setProperty()` method to add key-value pairs and the `getProperty()` method to retrieve one.

Code Snippet 6 shows the use of the `Properties` class.

Code Snippet 6:

```
package com.datastructures.demo;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.Properties;
import java.util.Set;
public class PropertiesDemo {
```

```

private static Properties initializeProperties() {
    Properties properties = new Properties();
    properties.setProperty("1", "East");
    properties.setProperty("2", "West");
    properties.setProperty("3", "North");
    properties.setProperty("4", "South");
    return properties;
}

public static void main(String[] args) {
    Properties initializedProperties =
        PropertiesDemo.initializeProperties();
    Set set = initializedProperties.keySet();
    Iterator itr = set.iterator();
    while (itr.hasNext()) {
        String str = (String) itr.next();
        System.out.println("The value of "
            + str + " is " +
            initializedProperties.getProperty(str) + ".");
    }
}
    
```

The `getInitializedProperties()` method of the code creates a `Properties` object and initializes it with key-value pairs through calls to the `setProperty()` method. The `main()` method calls the `keySet()` method to retrieve a `Set` object containing a collection of keys. The `iterator()` method returns an `Iterator` object that is used to iterate through the key-value pairs. For each iteration, the key-value pairs are printed by the `Iterator.next()` and `Properties.getProperty()`.

Figure 16.5 displays the output of the `PropertiesDemo` class.



```

Output - PropertiesDemo [run] X
run:
The value of 1 is South.
The value of 2 is North.
The value of 3 is West.
The value of 4 is East.
BUILD SUCCESSFUL (total time: 0 seconds)
    
```

Figure 16.5: PropertiesDemo - Output

16.5 Check Your Progress

1. Which element of Enumeration returns the next element, if present, in the enumeration?

(A) <code>nextElement()</code>	(C) <code>iterate()</code>
(B) <code>next()</code>	(D) <code>getNext()</code>

2. Which of the following code snippets correctly adds an element to a BitSet?

(A) <code>bitSetObject.add(1);</code>	(C) <code>bitSetObject.push(1);</code>
(B) <code>bitSetObject.put(1);</code>	(D) <code>bitSetObject.set(1);</code>

3. Which method of the Stack class removes the object from the top of the stack?

(A) <code>push()</code>	(C) <code>pop()</code>
(B) <code>remove()</code>	(D) <code>peek()</code>

4. Which of the following Hashtable methods returns an Enumeration for the Hashtable?

(A) <code>elements()</code>	(C) <code>next()</code>
(B) <code>enumeration()</code>	(D) <code>getEnumeration()</code>

5. Which method adds a key-value pair to a Properties object?

(A) <code>addProperty()</code>	(C) <code>push()</code>
(B) <code>add()</code>	(D) <code>setProperty()</code>

16.5.1 Answers

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

Summary

- Java includes a few legacy data structures such as Enumeration, BitSet, and so on for backward compatibility.
- Enumeration interface is used to iterate through the elements of a collection.
- BitSet is a collection of bit values.
- Stack extends Vector to provide an implementation of a LIFO collection.
- Dictionary is used to store key-value pairs.
- Hashtable stores key-value pairs where keys are organized based on their hash code.
- Properties stores key-value pairs where both the types of the keys and values are String.

Session - 17

Java Logging API and ResourceBundle

Welcome to the Session, Java Logging API and ResourceBundle.

This session introduces the Log4J logging framework and the ResourceBundle class.

In this Session, you will learn to:

- Describe the Log4J architecture
- Identify Log4J configuration options
- Explain the file appender
- Explain the JDBC appender
- Identify the ResourceBundle class



17.1 Log4J Overview

Log4J is an open-source logging framework for Java applications. Log4J enables generating log messages from different parts of the application. Log messages allow debugging the application for errors and tracing the execution flow. Log4J assigns different level of importance, such as ERROR, WARN, INFO, and DEBUG to log messages. With Log4J, log messages can be routed to different types of destinations, such as console, file, and database.

The Log4J architecture is composed of three primary components. The components are as follows:

- Loggers
- Appenders
- Layouts

17.1.1 Loggers

A logger is the primary Log4J component that is responsible for logging messages. Developers can create their own application-specific loggers or use the Log4J root logger. All Log4J loggers inherit from the root logger. Therefore, whenever a piece of code logs some message, Log4J searches for an application-specific logger. If none are found, Log4J uses the root logger.

In an application, the root logger can be instantiated and retrieved by calling the `LoggerManager.getLogger()` method. Application loggers can be instantiated and retrieved by calling the `LoggerManager.getLogger(String loggerName)` method that accepts the name of the desired logger as a parameter. Although, any name can be passed to the `getLogger()` method as a String, it recommend to name the logger with the fully qualified name of the class that will perform logging.

Loggers are assigned log levels where TRACE is the lowest level. The levels move up from TRACE through DEBUG, INFO, WARN, and ERROR, until the highest FATAL level.

Figure 17.1 displays the Log4J log levels.

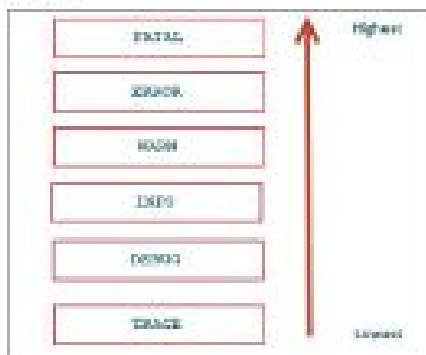


Figure 17.1: Log4J Log Levels

When a higher level is assigned to a logger, all log messages of that level and the levels below it are logged. For example, if the INFO level is assigned to a logger, then INFO, DEBUG, and TRACE messages are logged by the logger.

Note: In addition to the standard log levels, Log4J defines two special levels, named ON and OFF. The ON level turns on all levels while the OFF level turns off all levels.

17.1.2 Appenders

Loggers log messages to output destinations, such as console, file, and database. Such output destinations are known as appenders. Log4J provides a number of appender classes to log messages to various destinations. For example, `ConsoleAppender` logs messages to the console, `FileAppender` logs messages to a file, and `JDBCAppender` log messages to a relational database table.

Log4J also allows defining custom appenders. A custom appender extends from the `AppenderSkeleton` class that defines the common logging functionality. The core method of `AppenderSkeleton` that a custom appender should override is the `append()` method.

17.1.3 Layout

Layouts define how log messages are formatted in the output destination. Layouts are associated with appenders. Before an appender sends a log message to the output destination, it uses the associated layout to format the log message.

Log4J provides built-in layout classes, such as `PatternLayout`, `HtmlLayout`, `JsonLayout`, and `XmlLayout`. Log4J also supports custom layout that can be created by extending the abstract `AbstractStringLayout` class.

17.2 Log4J Configuration

Log4J is open source and can be freely downloaded. Once downloaded, the NetBeans project that will perform logging needs to be configured to include the Log4J JAR files.

17.2.1 Project Configuration

Steps to configure a NetBeans project to include the Log4J JAR files are as follows:

1. Download the Log4J binary file from the official Website, <https://logging.apache.org/log4j/2.0/download.html>.
2. Extract the compressed Log4J file into a suitable location.
3. Open NetBeans.
4. Create a Log4JDemo Java application project.

5. Select **Files → Project Properties (Log4JDemo)** from the main menu of NetBeans, as shown in figure 17.2.

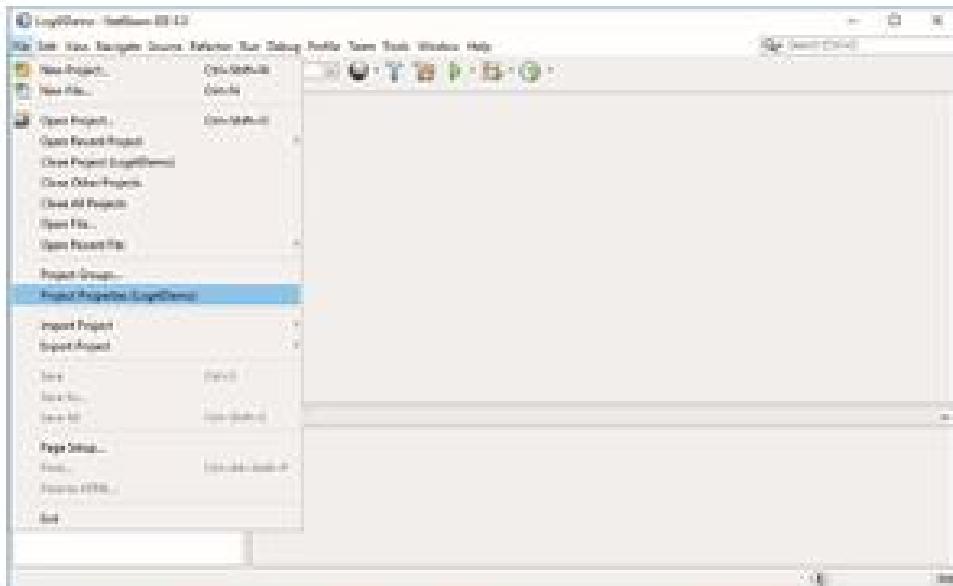


Figure 17.2: Project Properties (Log4JDemo) Menu Item

6. In the **Project Properties – Log4JDemo** dialog box that appears, select **Libraries**, and then click **Add JAR/Folder**, as shown in Figure 17.3.

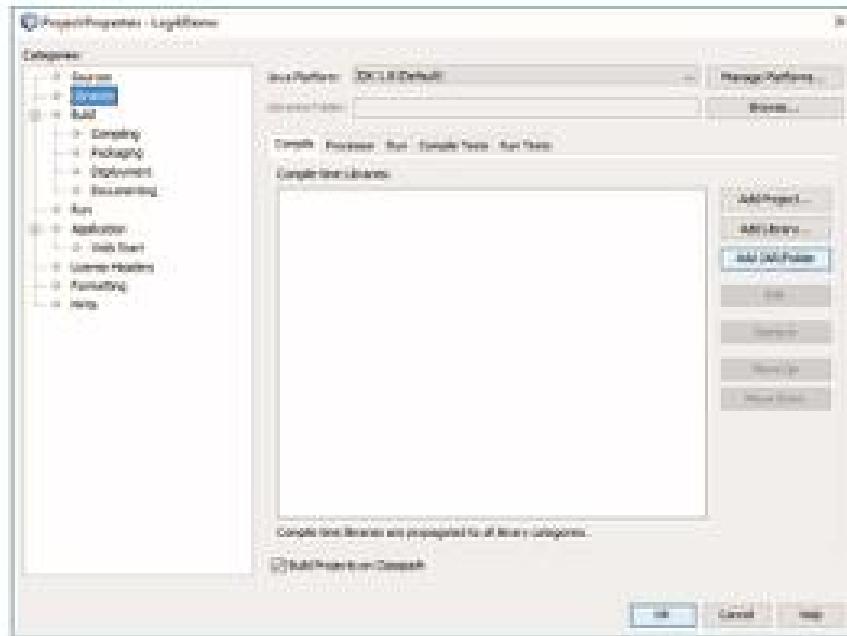


Figure 17.3: Project Properties – Log4JDemo Dialog Box

7. In the Add JAR/Folder dialog box that appear, browse to the downloaded Log4J directory, and select the log4j-api-2.x.x.jar and log4j-core-2.x.x.jar files by pressing the Ctrl button. Figure 17.4 displays the selected files.

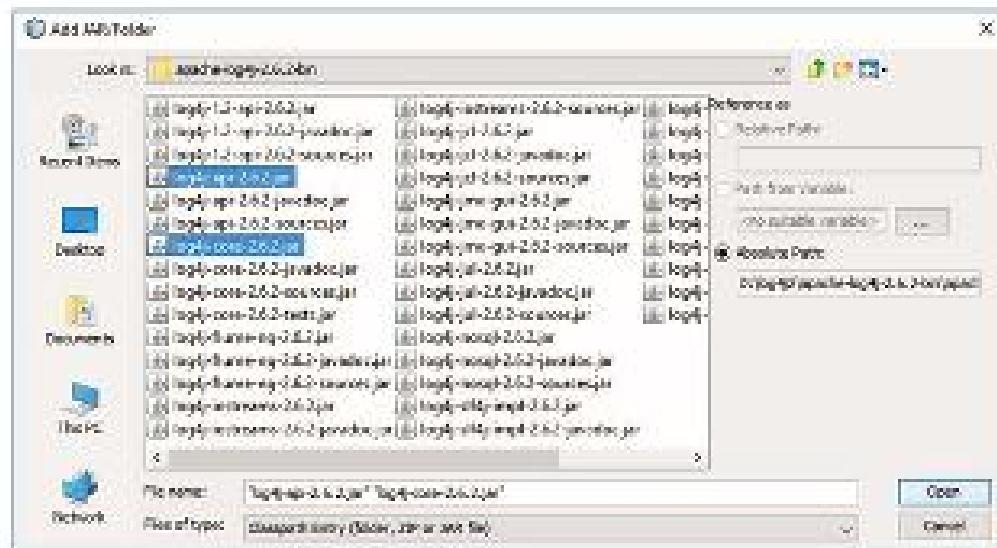


Figure 17.4: Log4J JAR Files

8. Click Open.
9. Click OK in the Project Properties – Log4JDemo dialog box. This adds the required Log4J JAR files to the project.

17.2.2 Logging Methods

For each of the log levels, Log4J defines a corresponding log method.

Methods

Table 17.1 lists the log methods of Log4J.

Method	Description
trace ()	Logs a method with the TRACE level.
debug ()	Logs a method with the DEBUG level.
info ()	Logs a method with the INFO level.
warn ()	Logs a method with the WARN level.
error ()	Logs a method with the ERROR level.
fatal ()	Logs a method with the FATAL level.
keySet ()	Returns a Set of all keys in the ResourceBundle

Table 17.1: Log Methods

Code Snippet 1 shows a `LoggerDemo` class that uses all the log methods.

Code Snippet 1:

```
package com.log4j.demo;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
public class LoggerDemo {

    private static Logger logger = LogManager.getLogger("LoggerDemo.class");
    public void performLogging() {
        logger.debug("This is a debug message");
        logger.info("This is an info message");
        logger.warn("This is a warn message");
        logger.error("This is an error message");
        logger.fatal("This is a fatal message");
    }
    public static void main(String[] args) {
        LoggerDemo logger = new LoggerDemo();
        logger.performLogging();
    }
}
```

The code calls the `getLogger()` method of `LogManager` passing the name of the class as parameter. The `getLogger()` method returns a `Logger` object for the class. The `performLogging()` method calls the log methods on the `Logger` object. The `main()` method calls the `performLogging()` method.

Figure 17.5 displays the output of the `LoggerDemo` class.



```
logger -log4jdemo.out =
[main]
DEBUG RootLogger is Log4J configuration file found. Using default configuration. Logging only occurs to the console.
[INFO:2019-09-19T11:10:00Z] [main] LoggerDemo.class - This is an INFO message
[WARN:2019-09-19T11:10:00Z] [main] LoggerDemo.class - This is a warn message
[ERROR:2019-09-19T11:10:00Z] [main] LoggerDemo.class - This is an error message
[FATAL:2019-09-19T11:10:00Z] [main] LoggerDemo.class - This is a fatal message
```

Figure 17.5: `LoggerDemo` - Output

The error message in the output is generated because no `Log4J` configuration file exists yet. As a result, `Log4J` uses the default configuration of the root logger. By default, root logger is configured with the `ERROR` log level. Therefore, only `ERROR` and `FATAL` messages got logged.

Log4J can be configured to use specific loggers, appenders, and layouts. The two most common configuration options are through properties and XML files.

17.2.3 Properties File Configuration

Log4J properties configuration file contains key-value pairs that specifies logging configuration options. By default, Log4j expects a properties file with the name `log4j2.properties` in the project classpath. In a NetBeans Java application project, the `log4j2.properties` file should be under the `src` directory. Code Snippet 2 shows a `log4j2.properties` configuration file.

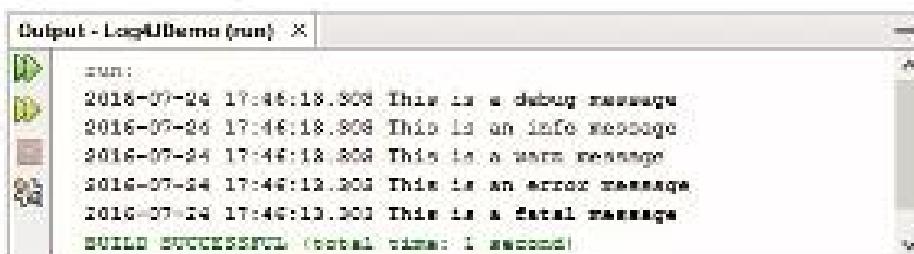
Code Snippet 2:

```
name = PropertiesConfig
appenders = consoleappender
appender.consoleappender.type = console
appender.consoleappender.name = STDOUT
appender.consoleappender.layout.type = PatternLayout
appender.consoleappender.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %msg%n
rootLogger.level = debug
rootLogger.appenders = stdout
rootLogger.appenders.stdout.ref = STDOUT
```

In the configuration code, the `name` and `appenders` properties specify the name of the configuration and the appender to use respectively. The properties starting with `appender` configures the appender to use. The `appender.consoleappender.type` property specifies `console` to use the Log4J `console` appender. The `appender.consoleappender.layout.type` and `appender.consoleappender.layout.pattern` properties specifies the pattern layout to use for the appender and the specific pattern to use. The pattern contains a date and time format represented by `%d` option. The `%msg` option outputs the application supplied message and the `\n` option specifies that a new line character should be used.

The `rootLogger.level` property configures the root logger with the `DEBUG` level. Finally, the `rootLogger.appenders` and `rootLogger.appenders.stdout.ref` properties associate the `console` appender with the root logger.

Once this is done, execute the `LoggerDemo` class given in Code Snippet 1. The root logger outputs all the log messages, as shown in figure 17.6.



```
Output - Log4jDemo (run) X
run:
2016-07-24 17:46:18,803 This is a debug message
2016-07-24 17:46:18,803 This is an info message
2016-07-24 17:46:18,803 This is a warn message
2016-07-24 17:46:18,803 This is an error message
2016-07-24 17:46:18,803 This is a fatal message
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 17.6: Logging Using `log4j2.properties` Configuration File

17.2.4 XML File Configuration

Log4J also support configurations through XML file. Similar to properties file, a `log4j2.xml` file must be present in the project classpath.

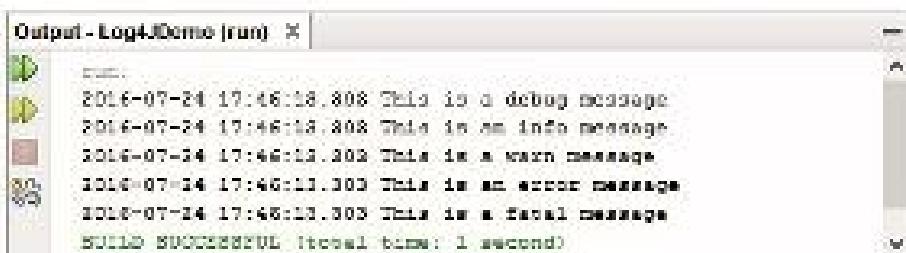
Code Snippet 3 shows a `log4j2.xml` configuration file.

Code Snippet 3:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration name="PropertiesConfig">
<Appenders>
  <Console name="consoleappender" target="STDOUT">
    <PatternLayout>
      <pattern>
        %d{yyyy-MM-dd HH:mm:ss.SSS} %msg%n
      </pattern>
    </PatternLayout>
  </Console>
</Appenders>
<Loggers>
  <Root level="DEBUG">
    <AppenderRef ref="consoleappender"/>
  </Root>
</Loggers>
</Configuration>
```

A Log4J XML configuration file contains the <Configuration> root element. The <Appenders> element contains a <Console> element to configure a console appender. The <PatternLayout> element specifies the pattern layout to use with the appender and the <pattern> element specifies the formatting pattern to use. The <Loggers> element contains the <Root> element to configure the root logger. The level attribute of the <Root> element assigns the DEBUG log level to the root logger. Finally, the ref attribute of the <AppenderRef> element assigns the console appender to the root logger. This XML configuration performs the same function as the properties configuration given in Code Snippet 2.

Figure 17.7 displays the output on executing the LoggerDemo class given in Code Snippet 1.



```
Output - Log4JDemo [run] X
2016-07-24 17:46:18.808 This is a debug message
2016-07-24 17:46:18.808 This is an info message
2016-07-24 17:46:18.808 This is a warn message
2016-07-24 17:46:18.808 This is an error message
2016-07-24 17:46:18.808 This is a fatal message
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 17.7: Logging Using log4j2.xml Configuration File

17.3 File Appender

The file appender is used to send log messages to a file. A file appender can be configured with the <File> element in an XML configuration file.

Code Snippet 4 shows the use of a file appender.

Code Snippet 4:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration name="PropertiesConfig">
<Appenders>
    <File name="fileappender" fileName="applogs/logfile.log">
        <PatternLayout>
            <pattern>
                %d{yyyy-MM-dd HH:mm:ss,SSS} %msg%n
            </pattern>
        </PatternLayout>
    </File>
</Appenders>
<Loggers>
```

```
<Root level="DEBUG">
  <AppenderRef ref="fileappender"/>
</Root>
</Loggers>
</Configuration>
```

The code on executing the `LoggerDemo` class given in [Code Snippet 1](#)

creates a `logfile.log` file containing log messages in the `applogs` directory. The `applogs` directory will be created in the project root directory.

[Figure 17.8](#) displays the content of the `logfile.log` file.

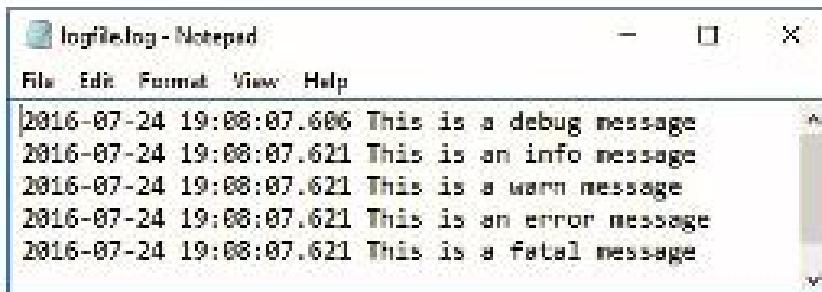


Figure 17.8: logfile.log File

17.4 JDBC Appender

The JDBC Appender supports storing log messages in a database table. The `<JDBC>` element configures a JDBC appender. To use a JDBC appender, you need the following mandatory information:

- The connection URL to the database
- The database table name to insert log messages
- The columns in the table to write log messages

To use the JDBC appender, a relational database server is required. One popularly used database for Java application is MySQL. You can download MySQL from

<http://dev.mysql.com/downloads/windows/installer/5.7.html>

Once MySQL is installed, you need to create a database and a table in it to store logging information.

Code Snippet 5 shows the statements to create a database and a table.

Code Snippet 5:

```
mysql>create database LOG4JLOG;
mysql>use LOG4JLOG;
mysql>CREATE TABLE applicationlog (ID varchar(100), LEVEL varchar(100), LOGGER
varchar(100), MESSAGE varchar(100));
```

17.4.1 Connection Factory

In order to use the JDBC appender, the application needs a connection to the database. You can create a connection factory using the Apache commons-dbcp package. This package relies on code in the commons-pool package to manage connection pool.

Note: The commons-dbcp package can be downloaded from https://commons.apache.org/proper/commons-dbcp/download_dbcp.cgi.

The commons-pool package can be downloaded from https://commons.apache.org/proper/commons-pool/download_pool.cgi.

Code Snippet 6 shows a MySqlConnectionFactory class that creates a connection to the LOG4JLOG database.

Code Snippet 6:

```
package com.log4j.demo;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;
import javax.sql.DataSource;
import org.apache.commons.dbcp.DriverManagerConnectionFactory;
import org.apache.commons.dbcp.PoolableConnection;
import org.apache.commons.dbcp.PoolableConnectionFactory;
import org.apache.commons.dbcp.PoolingDataSource;
import org.apache.commons.pool.impl.GenericObjectPool;
public class MySqlConnectionFactory {
  private static interface Singleton {
    final MySqlConnectionFactory INSTANCE = new
      MySqlConnectionFactory();
  }
  private final DataSource dataSource;
```

```

private MySqlConnectionFactory() {
    Properties properties = new Properties();
    properties.setProperty("user", "root");
    properties.setProperty("password", "root");
    GenericObjectPool pool = new GenericObjectPool();
    DriverManagerConnectionFactory connectionFactory = new
    DriverManagerConnectionFactory(
        "jdbc:mysql://127.0.0.1/log4jlog", properties);
    new PoolableConnectionFactory(connectionFactory, pool, null,
        "SELECT 1", 3, false, false,
        Connection.TRANSACTION_READ_COMMITTED);
    this.dataSource = new PooledDataSource(pool);
}

public static Connection getDatabaseConnection() throws SQLException {
    return Singleton.INSTANCE.dataSource.getConnection();
}
  
```

The code creates a `MySqlConnectionFactory` as a singleton. The class constructor uses a `Properties` object to set up the database user name and password credentials. The constructor then initializes a `DataSource` object from a `PoolableConnectionFactory` that it constructs. The `getDatabaseConnection()` static method is responsible for returning a `Connection` object.

17.4.2 Configuration File

The Log4J configuration file is responsible for configuring the JDBC appender to use the connection returned by the `MySqlConnectionFactory` class for inserting logging information.

Code Snippet 7 shows the `log4j2.xml` file to configure the JDBC appender.

Code Snippet 7:

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="error">
  <Appenders>
    <JDBC name="databaseAppender" tableName="LOG4JLOG_APP_LOG">
      <ConnectionFactory
          class="com.log4j.demo.MySqlConnectionFactory"
          method="getDatabaseConnection" />
    
```

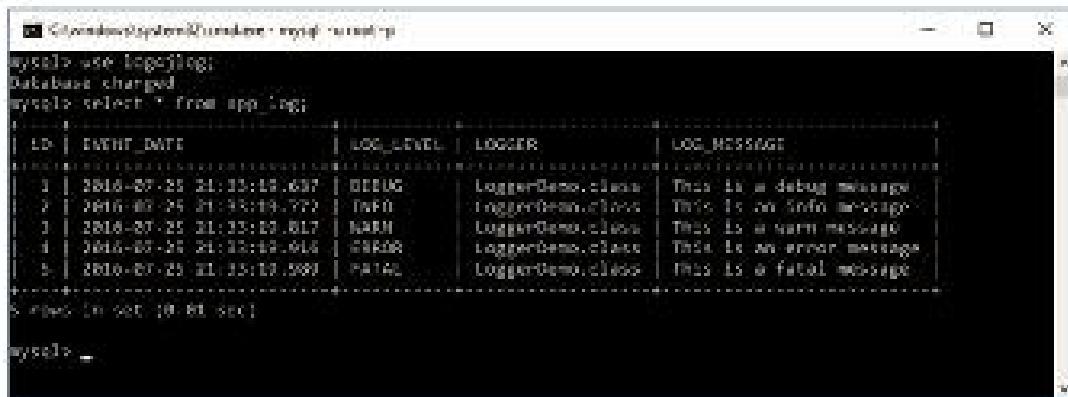
```

<Column name="EVENT_DATE" isEventTimestamp="true" />
<Column name="LOG_LEVEL" pattern="%level" />
<Column name="LOGGER" pattern="%logger" />
<Column name="LOG_MESSAGE" pattern="%message" />
</JDBC>
</Appenders>
<Loggers>
<Root level="DEBUG">
<AppenderRef ref="databaseAppender" />
</Root>
</Loggers>
</Configuration>

```

The code uses the `<JDBC>` element to configure the JDBC appender. The name and tableName attributes of the `<JDBC>` element specifies the appender name and the table name to which logging data will be inserted. The class and method attributes of the `<Appenders>` element specifies the connection factory class and the method that returns a connection. The `<Column>` maps the table columns with the logging data that the columns will hold. Finally, the `<Loggers>` element associates the JDBC appender with the root logger.

Figure 17.9 shows the output on executing the `LoggerDemo` class given in Code Snippet 1 at the mysql prompt.



The screenshot shows a terminal window running MySQL. The user has created a database named 'log4jlog' and selected it. They then ran a query to select all rows from the 'app_log' table. The table has four columns: ID, EVENT_DATE, LOG_LEVEL, and LOG_MESSAGE. The data shows five entries corresponding to the log messages printed by the `LoggerDemo` class in the code snippet. The log levels are DEBUG, INFO, WARN, ERROR, and FATAL respectively.

ID	EVENT_DATE	LOG_LEVEL	LOG_MESSAGE
1	2016-07-25 21:33:19.697	DEBUG	LoggerDemo.class: This is a debug message
2	2016-07-25 21:33:19.772	INFO	LoggerDemo.class: This is an info message
3	2016-07-25 21:33:19.817	WARN	LoggerDemo.class: This is a warn message
4	2016-07-25 21:33:19.918	ERROR	LoggerDemo.class: This is an error message
5	2016-07-25 21:33:19.999	FATAL	LoggerDemo.class: This is a fatal message

Figure 17.9: Logging Data in Database Table

17.5 ResourceBundle Class

The `ResourceBundle` class of the `java.util` package enables creating localized programs that can display the UI and output in different languages. An object of the `ResourceBundle` class represents locale-specific information. When a program needs a locale-specific resource, for example, a String, the program loads it from the `ResourceBundle` based on the current locale of the user.

The `PropertyResourceBundle` and `ListResourceBundle` classes extend `ResourceBundle`. The `PropertyResourceBundle` is a concrete class to represent locale-specific information stored as key-value pairs in properties file. On the other hand, `ListResourceBundle` is an abstract class to represent locale-specific information stored in list-based collections.

Java programs do not interact directly with the `PropertyResourceBundle` and `ListResourceBundle` classes. All interactions of programs go through the `ResourceBundle` class.

The `ResourceBundle` class has a single default controller. However, there are several methods that programs can use to construct a `ResourceBundle` and interact with local-specific resources.

Methods

Table 17.2 lists the key methods available in the `ResourceBundle` class.

Method	Description
<code>getBundle()</code>	Returns a <code>ResourceBundle</code> object for the default locale. Overloaded version of this method accepts a <code>Locale</code> object to return a <code>ResourceBundle</code> object for the specified locale.
<code>getLocale()</code>	Returns the current locale of the user.
<code>getObject(String key)</code>	Returns the object for the corresponding key from the resource bundle.
<code>clearCache()</code>	Clears the cache of all resource bundles loaded by the class loader.
<code>containsKey(String key)</code>	Checks whether or not the specified key exists in the resource bundle.
<code>getKeys()</code>	Returns an Enumeration of all keys in the resource bundle.
<code>keySet()</code>	Returns a Set of all keys in the <code>ResourceBundle</code> .

Table 17.2: Methods of the `ResourceBundle` Class

17.6 Check Your Progress

1. Which of the following classes implements the CompletionStage and the Future interfaces to simplify coordination of asynchronous operations?

(A) <code>LoggerManager.getLogger(String name)</code>	(C) <code>LoggerManager.getLogger()</code>
(B) <code>LoggerManager.getRootLogger()</code>	(D) <code>LoggerManager.getLogger(String name)</code>

2. What is the name of the Log4J XML configuration file?

(A) <code>log4j2.xml</code>	(C) <code>log.xml</code>
(B) <code>log4jConfig.xml</code>	(D) <code>log4jLog.xml</code>

3. Which XML element represents a file appender?

(A) <code><Appenders></code>	(C) <code><File></code>
(B) <code><Appender></code>	(D) <code><FileAppender></code>

4. Which of the following statements is true regarding the connection factory class used with a JDBC appender?

(A) Returns a pool of connections to the database	(C) Maps the columns of a database table with logging data
(B) Establishes a connection to the database	(D) Associates a JDBC appender to a logger for establishing connection to a database

5. Which method of the ResourceBundle class returns an Enumeration of all keys in the ResourceBundle?

(A) <code>keySet()</code>	(C) <code>getKeySet()</code>
(B) <code>getKeys()</code>	(D) <code>getEnumeratedKey()</code>

17.6.1 Answers

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

Summary

- The Log4J architecture is composed of loggers, appenders, and layouts.
- Properties and XML files are two most common approaches to specify Log4J configuration options.
- The file appender redirects logging data to a file.
- The JDBC appender redirects logging data to a database table.
- The ResourceBundle class enables creating localized programs based on user locales.

Session - 18

Java Documentation and Networking

Welcome to the Session, Java Documentation and Networking.

This session explores Java documentation and also describes networking in Java.

In this Session, you will learn to:

- Explain the Javadoc tool
- Describe the key classes of the java.net package
- Explain socket programming
- Explain URL processing



18.1 Javadoc Tool

The Javadoc tool enables creating HTML-based API documentation of Java source code. Javadoc relies on documentation tags present in the source code. This tool can be used to create documentation of packages, classes, interfaces, methods, and fields.

18.1.1 Documentation in Java

The Java API is a large collection of types where each type can have a large number of constructs, such as constructors, fields, and methods. In order to effectively use the Java API, developers need to know the purpose of the types and its constructs. The information is provided through API documentation. While developing applications and libraries, Java developers also need to provide documentation of the classes and their constructs. The aim is to describe everything that another developer would require to use the API. For example, an API documentation of a method typically provides information about what the method does, its access specifier, parameters, return type, and any exceptions thrown. Such information can help other developers understand what has been done and why it has been done, while browsing through the code.

In Java, API documentation is created using documentation tags. Based on the tags, the Javadoc tool generates API documentation in HTML.

The tool is automatically installed as part of the Java SDK. It can be found in the bin folder of the JDK path. IDE tools such as Eclipse and NetBeans also have support for Javadoc.

18.1.2 Javadoc Tags

The Javadoc tags can be primarily divided into class-level and method-level tags.

Class-level Tags

Table 18.1 explains the class-level tags.

Tag	Description
@author	Insert the author name of the class
{@code}	Insert text in code format
@see	Inserts a See Also heading with a link or text entry that points to closely related reference.
@since	Inserts a Since heading used to specify from when this class exists
@deprecated	Inserts a comment to indicate that this class is deprecated, and should no longer be used

Table 18.1: Class-level Tags

Code Snippet 1 shows the use of class-level tags.

Code Snippet 1:

```
/**
 * @author Carl Boynton
 * @author Andy Payne
 * @see Collection
 * @see Vector
 * @since JDK1.0
 */
public class MathDemo {
    /*Code implementation*/
}
```

Thus, the class MathDemo will have information indicating who are the authors of the code, which classes to see further, and since which version the class has been existing.

Method-level Tags

Table 18.2 explains the method-level tags.

Tag	Description
@param	Inserts a parameter that the method accepts
@return	Inserts the return type of the method
@throws	Inserts any exception that the method throws
@see	Inserts a See Also heading with a link or text points to closely related methods
@since	Inserts a Since heading with a text to specify from when this class exists
@deprecated	Inserts a comment to indicate that this method is deprecated, and should no longer be used

Table 18.2: Method-level Tags

Code Snippet 2 shows the use of the method-level tags.

Code Snippet 2:

```
/**
 * @param num1 This is the first parameter to addInt method
 * @param num2 This is the second parameter to addInt method
 * @return int This returns sum of numA and numB.
 * @see MathDemo#addLong(long,long)
 */
public int addInt(int num1, int num2) {
    return num1 + num2;
}
```

The `@see` annotation in the code specifies an `addLong(long, long)` method in the `MathDemo` class. This method must be defined in the `MathDemo` class failing which the `Javadoc` tool will report a compilation error.

In addition to `Javadoc` tags, Java API documentation can contain text for explaining a class or its constructs. Such text must be included as documentation comments.

Code Snippet 3 shows a complete example that demonstrates the use of `Javadoc` tags and documentation comments.

Code Snippet 3:

```
/*
 * The (@code MathDemo) class implements a calculation algorithm to
 * add two integers.
 * @author Carl Boynton
 * @author Andy Payne
 * @see Math
 * @since JDK8.0
 */
public class MathDemo {
    /**
     * Constructs a MathDemo instance.
     */
    public MathDemo() { }
    public long addLong(long num1, long num2) {
        return num1 + num2;
    }
    /**
     * This method is used to add two integers.
     * @param num1 This is the first parameter to addInt method
     * @param num2 This is the second parameter to addInt method
     * @return int This returns sum of numA and numB.
     */
    public int addInt(int num1, int num2) {
        return num1 + num2;
    }
    /**
     * This is the main method to use addInt method.
     * @param args Unused.
     * @exception java.io.IOException on input error.
     * @see java.io.IOException
     */
    public static void main(String[] args) throws java.io.IOException{
        MathDemo mathDemo = new MathDemo();
        System.out.println(mathDemo.addInt(5, 8));
    }
}
```

You can generate Java documentation for this code snippet using one of two ways:

at the command prompt using the Javadoc tool or using the IDE option.

Consider the first approach. You will type a command as follows at the command prompt:

```
javadoc MathDemo.java
```

It will result in an HTML file containing the Javadoc generated documentation.

Figure 18.1 displays the Javadoc generated documentation opened in the browser.

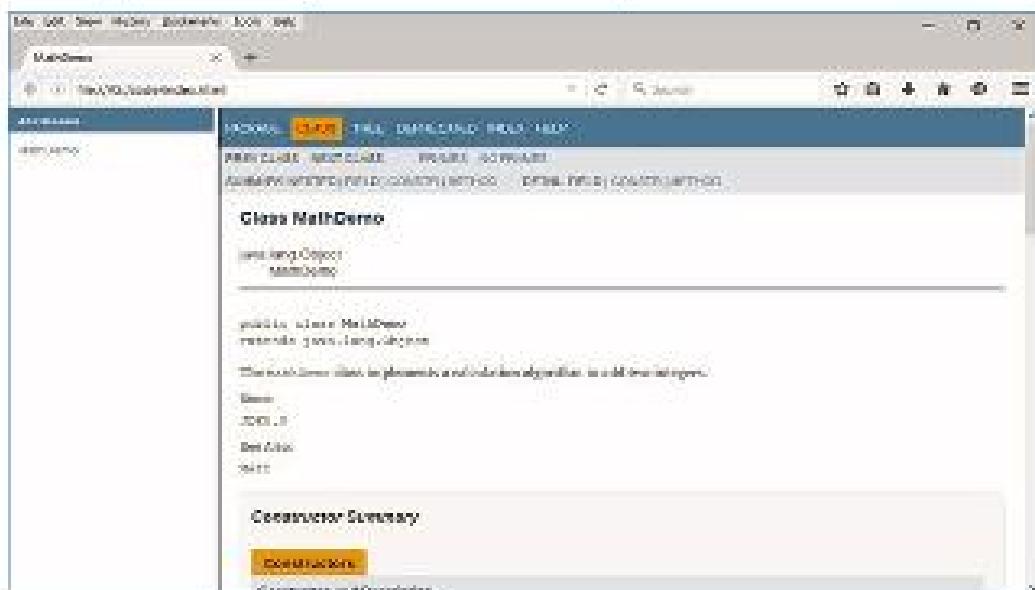


Figure 18.1: Javadoc Generated Documentation

The second option of generating Javadoc is to use the Javadoc generation features of an IDE, such as NetBeans. NetBeans enables automatically inserting Javadoc comments and tags, generating Javadoc, and viewing Javadoc documentation.

NetBeans assists in writing Javadoc through hints and the code-completion feature. Using the code-completion feature, basic Javadoc comments can be automatically generated in source files. For a Javadoc comment, typing `/**` and pressing the TAB and ENTER key automatically generates a Javadoc comment block. For a method, typing `@param` and `@return` tags depending on the parameter numbers, types, and return type of the method being documented. For other tags, a hint appears as a pop up as a Javadoc tag is typed. On clicking a tag or pressing the ENTER key, the tag is inserted in the source file.

Figure 18.2 shows the Javadoc tag code-completion pop-up.

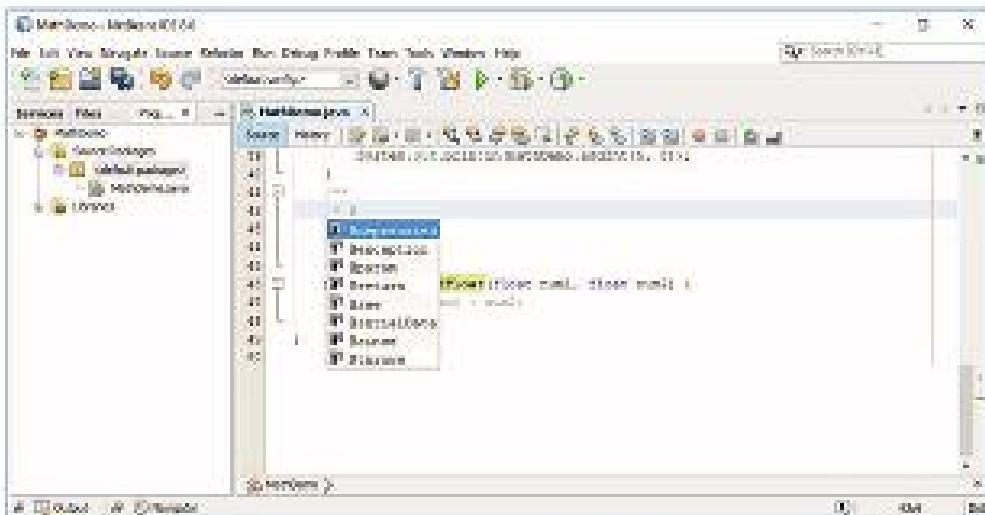


Figure 18.2: Code-completion of Javadoc Tag

Once you document a Java class, you can easily generate the documentation of the class using the integrated Javadoc tool of NetBeans. To generate Javadoc documentation, open the class in NetBeans. Then, select Run → Generate Javadoc from the main menu of NetBeans, as shown in figure 18.3.

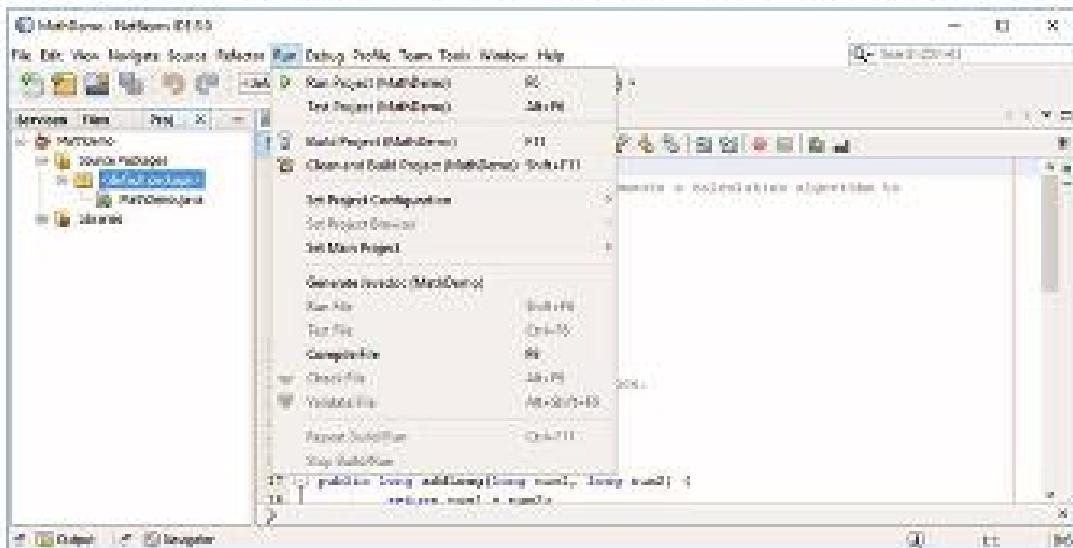


Figure 18.3: Javadoc Documentation Generation in NetBeans

NetBeans generates the Javadoc in the `dist/javadoc` folder of the project directory. Once Javadoc is generated it can be viewed on the browser. To view the documentation of a class, right-click the class in the code editor, and then select Show Javadoc from the contextual menu, as shown in Figure 18.4.

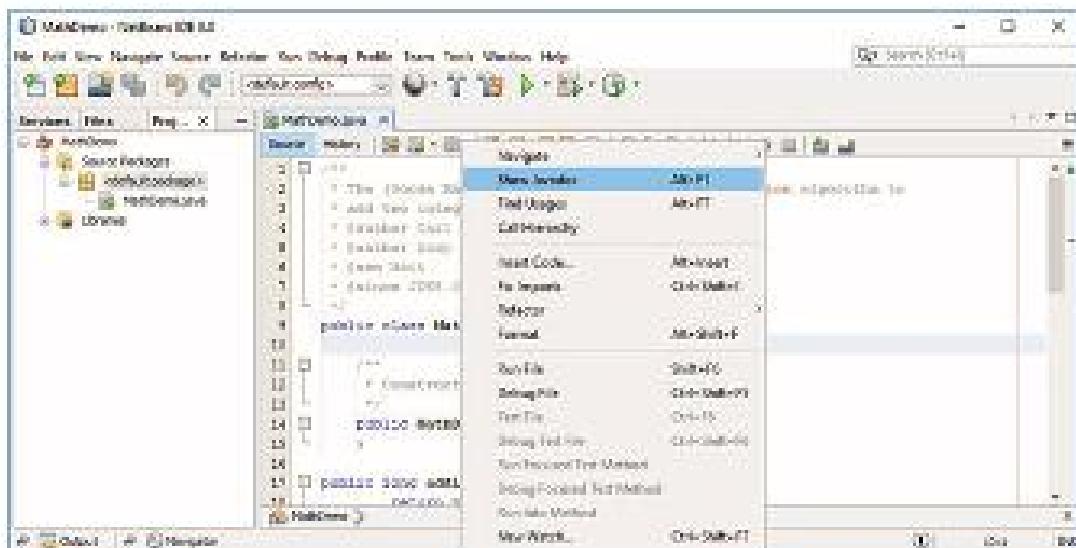


Figure 18.4: Viewing Javadoc in NetBeans

The browser displays the documentation.

Figure 18.5 displays the Javadoc generated documentation in the browser.

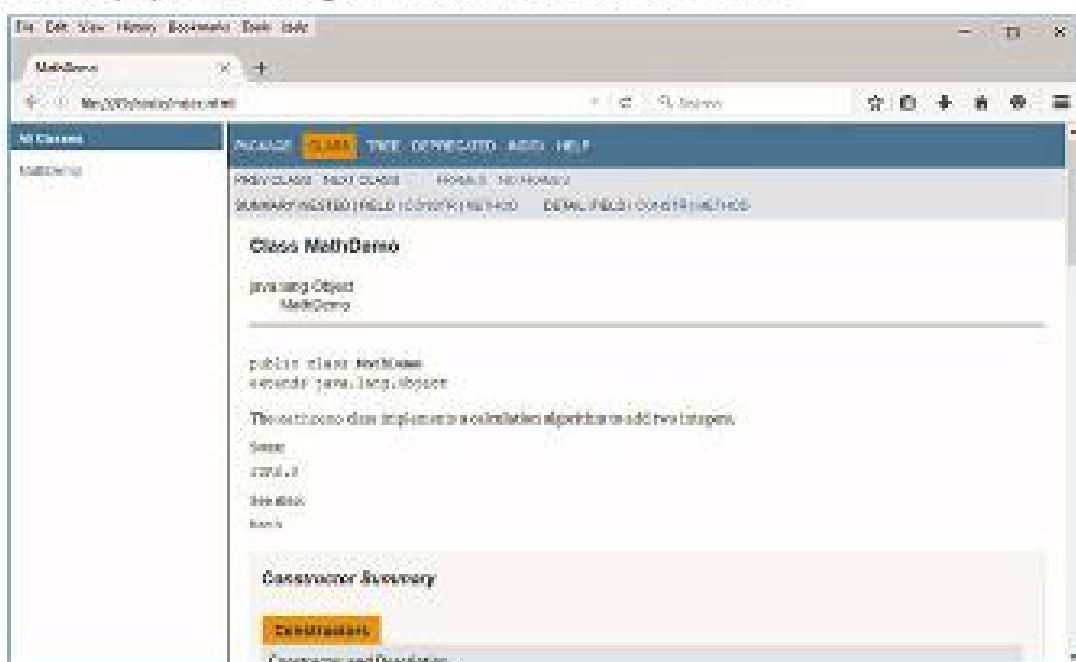


Figure 18.5: Javadoc Generated Documentation

18.2 java.net Package

The `java.net` package contains classes and interfaces for network programming. Using the `java.net` package, developers can create applications that communicate over networks in the client-server model. The package contains specialized classes to create transport layer client and server sockets and also perform communication over the Internet.

Some of the important classes of the `java.net` package are as follows:

- `DatagramPacket`
- `DatagramSocket`
- `Socket`
- `ServerSocket`
- `InetAddress`
- `URL`
- `URLConnection`

18.2.1 DatagramPacket Class

The `DatagramPacket` class represents datagram packets that are used for communication using the User Datagram Protocol (UDP) protocol. The UDP protocol is a lightweight transport-layer protocol used for connectionless delivery of data packets, also known as datagram packets. A datagram packet contains information about itself and its destination, based on which the packet is transferred from the source to destination. Datagram packets might be routed differently. In addition, UDP does not guarantee datagram packet delivery and that datagram packets will arrive in a specific order.

Note - A `DatagramPacket` object can have a maximum size of 65507 bytes.

Methods

Table 18.3 explains the methods of the `DatagramPacket` class.

Method	Description
<code>setData(byte[] buf)</code>	Sets the data of a packet as a <code>byte[]</code>
<code>setAddress(InetAddress iaddr)</code>	Sets the IP address of the computer to which a datagram packet needs to be sent
<code>setLength(int length)</code>	Sets the length of a packet as an <code>int</code> value
<code>getData()</code>	Returns the data of the packet as a <code>byte[]</code>
<code>getLength()</code>	Returns the length of data in a packet to be sent or in a packet that has been received
<code>getAddress()</code>	Returns the IP address of the computer to which a datagram packet is sent or the computer that sends a datagram packet

Table 18.3: Methods of the `DatagramPacket` Class

18.2.2 DatagramSocket Class

The `DatagramSocket` class is responsible for sending and receiving datagram packets as `DatagramPacket` objects.

Methods

Table 18.4 explains the methods of the `DatagramSocket` class.

Method	Description
<code>connect(InetAddress address, int port)</code>	Connects the socket to the IP address and port of a remote computer
<code>disconnect()</code>	Disconnects the socket
<code>send(DatagramPacket packet)</code>	Sends a <code>DatagramPacket</code> object to a destination.
<code>receive(DatagramPacket packet)</code>	Receives a <code>DatagramPacket</code> object.

Table 18.4: Methods of the `DatagramSocket` Class

18.2.3 Socket Class

The `Socket` class represents the socket used by both the client and server for communicating data. The `Socket` class is used for communication over the Transmission Control Protocol (TCP) protocol. Similar to UDP, TCP is a transport layer protocol. TCP is positioned above Internet Protocol (IP), and therefore, is also referred as TCP/IP.

However, unlike UDP, TCP maintains a connection between endpoints that `Socket` objects represents. In both the client and the server, `Socket` objects are used to send and receives TCP packets. Unlike UDP that does not guarantee packet deliveries and packet ordering, TCP guarantees both because both the client and server sockets remains connected. On account of the overhead of maintaining a connection, TCP is slower than UDP.

To transmit data to a server, a client creates an object of the `Socket` class. However, the server never instantiates a `Socket`. The server obtains a `Socket` object by calling the `accept()` method of the `ServerSocket` class.

A client can create a `Socket` to represent a connection to the server by invoking the public `Socket(String host, int port)` constructor of the `Socket` class. In the constructor, the first parameter specifies the IP address of the server and the second address specifies the port number that the server listens for messages.

Methods

Table 18.5 explains the key methods of the `Socket` class.

Method	Description
<code>connect(SocketAddress host, int timeout)</code>	Connects the client socket to the server socket. This method is required if a <code>Socket</code> object is created without initializing it with a connection to the server.
<code>getInputStream()</code>	Returns an <code>InputStream</code> object of the <code>Socket</code> . Both clients and servers use the <code>getInputStream()</code> method to receive data.
<code>getOutputStream()</code>	Returns an <code>OutputStream</code> object of the <code>Socket</code> . Both clients and servers use the <code>getOutputStream()</code> method to send data.
<code>close()</code>	Closes the <code>Socket</code> connection.

Table 18.5: Methods of the `Socket` Class

18.2.4 ServerSocket Class

The `ServerSocket` class is used by servers to listen for incoming connections from clients. Typically, a server invokes the `ServerSocket(String port)` constructor to instantiate a `ServerSocket` object. Once the server obtains a `ServerSocket` object, the server can call various methods of `ServerSocket` to accept connections and obtain information about the `ServerSocket` object.

Methods

Table 18.6 explains the key methods of the `ServerSocket` class.

Method	Description
<code>bind(SocketAddress endPoint)</code>	Binds a <code>ServerSocket</code> object to a specified IP address and port number that the <code>SocketAddress</code> parameter represents.
<code>accept()</code>	Lists for a connection to be made to this socket and accepts it. The <code>accept()</code> method blocks until either a client connects to the server on the specified port or the socket times out.
<code>getLocalPort()</code>	Returns the port number as an <code>int</code> value that a <code>ServerSocket</code> object is listening to.
<code>setSoTimeout(int timeout)</code>	Sets a timeout in milliseconds after which a <code>ServerSocket</code> object stops accepting client connections.
<code>isClosed()</code>	Returns a boolean value to indicate whether or not a <code>ServerSocket</code> object is closed.

Table 18.6: Methods of the `ServerSocket` Class

18.2.5 InetAddress Class

The `InetAddress` class represents an Internet address to perform a Domain Name System (DNS) look-up and reverse look-up. An Internet address, or IP address that the `InetAddress` class represents is a 32-bit number, often represented as a set of four 8-bit numbers separated by periods. For example, 127.0.0.1 is the IP address of a local computer.

Methods

Table 18.7 explains the important methods of the `InetAddress` class.

Method	Description
<code>getAddress()</code>	Returns the IP address of the <code>InetAddress</code> object as a <code>byte[]</code>
<code>getByName(String host)</code>	Returns the IP address of the host passed as parameter as an <code>InetAddress</code> object
<code>getHostName()</code>	Returns the host name of the <code>InetAddress</code> object
<code>getAllByName(String host)</code>	Returns an array of its IP addresses for the host passed as parameter
<code>isReachable(int timeout)</code>	Returns a boolean to indicate whether or not the IP address represented by <code>InetAddress</code> is reachable

Table 18.7: Methods of the `InetAddress` Class

18.2.6 URL Class

The `URL` class represents a Uniform Resource Locator (URL) that points to a resource on the Web. A URL can be a Web page, a document, an image, or any other file.

Methods

Table 18.8 explains the important methods of the `URL` class.

Method	Description
<code>getPath()</code>	Returns the path of the URL as a <code>String</code>
<code>getQuery()</code>	Returns the query part of the URL as a <code>String</code>
<code>getPort()</code>	Returns the port of the URL as an <code>int</code> value
<code>getDefaultPort()</code>	Returns the default port for the protocol of the URL as an <code>int</code> value
<code>getProtocol()</code>	Returns the protocol of the URL as a <code>String</code>
<code>getHost()</code>	Returns the host of the URL as a <code>String</code>
<code>getFile()</code>	Returns the filename of the URL as a <code>String</code>
<code>openConnection()</code>	Opens a connection to the URL and returns a <code>URLConnection</code> object

Table 18.8: Methods of the `URL` Class

18.2.7 URLConnection Class

The `openConnection()` method of the `URL` class returns an implementation of the `URLConnection` class. The `URLConnection` class is an abstract class that acts as the superclass of all classes that represent a communications link between an application and a URL.

Methods

Table 18.9 explains the important methods of the `URLConnection` class.

Method	Description
<code>getURL()</code>	Returns the URL that the <code>URLConnection</code> object is connected to as a <code>URL</code> object
<code>setDoInput(boolean input)</code>	Accepts a boolean value to indicate whether the <code>URLConnection</code> object will be used for input. The default value is true
<code>setDoOutput(boolean output)</code>	Accepts a boolean value to indicate whether the <code>URLConnection</code> object will be used for output. The default value is false
<code>getInputStream()</code>	Returns the input stream of the <code>URLConnection</code> as an <code>InputStream</code> object. This method is called to read from a URL
<code>getOutputStream()</code>	Returns the output stream of the <code>URLConnection</code> as a <code>OutputStream</code> object. This method is called to write to a URL
<code>getContent()</code>	Returns an Object of the contents of the <code>URLConnection</code>
<code>getContentEncoding()</code>	Returns the content-encoding header field of the of the <code>URLConnection</code> as a <code>String</code> object.
<code>getContentLength()</code>	Returns the content-length header field of the of the <code>URLConnection</code> as an int value
<code>getContentType()</code>	Returns the content-type header field of the of the <code>URLConnection</code> as a <code>String</code> object
<code>getLastModified()</code>	Returns the last-modified header field as an int value
<code>getExpiration()</code>	Returns the expires header field as a long value
<code>getIfModifiedSince()</code>	Returns the ifModifiedSince field of the <code>URLConnection</code> object as a long value

Table 18.9: Methods of the `URLConnection` Class

18.3 Socket Programming and URL Processing

The `Socket` and `ServerSocket` classes of the `java.net` package enables socket programming over TCP. Socket programming involves a server and a client.

18.3.1 Client/Server Program

In a client/server programs that communicates using TCP/IP, a server is created to listen for client connections. Then, a client is created to connect with the server and exchange data packages.

The steps to create a server class are as follows:

1. Create the server socket as a `ServerSocket` object
2. Wait for a client request
3. Create input and output streams to receive and send data respectively
4. Perform communication with the client
5. Close the socket

Code Snippet 5 shows the use of `ServerSocket` to create a server.

Code Snippet 5:

```
package com.io.demo;  
import java.io.DataInputStream;  
import java.io.DataOutputStream;  
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.net.SocketTimeoutException;  
  
public class Server extends Thread {  
    private ServerSocket serverSocket;  
  
    public Server(int port) throws IOException {  
        serverSocket = new ServerSocket(port);  
    }  
  
    public void run() {  
        while (true) {  
            try {  
                System.out.println("Listening for client message on  
                port " + serverSocket.getLocalPort());  
                Socket socket = serverSocket.accept();  
                DataInputStream in = new DataInputStream(  
                    socket.getInputStream());  
                DataOutputStream out = new DataOutputStream(socket.  
                    getOutputStream());  
                out.writeUTF("Hello from server.");  
            }  
        }  
    }  
}
```

```

        catch (SocketTimeoutException sTException) {
            sTException.printStackTrace();
        }
        catch (IOException ioException) {
            ioException.printStackTrace();
        } finally {
            try {
                serverSocket.close();
            }
        }
        catch (IOException ioException) {
            ioException.printStackTrace();
        }
    }
}

public static void main(String[] args) {
    try {
        Thread thread = new Server(6060);
        thread.start();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

The code creates a `Server` class that extends `Thread`. The constructor creates a `ServerSocket` object with the specified port passed as parameter. In the overridden `run()` method of the `Thread` class, the code calls the `accept()` method of `ServerSocket` to obtain a `Socket` object. The code then constructs a `DataOutputStream` object and uses it to write out a message. The `main()` method creates a `Server` object as a `Thread` and starts it.

Figure 18.6 displays the output of the server.

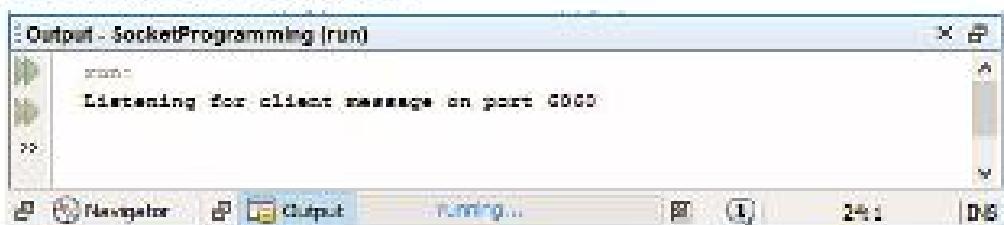


Figure 18.6: Output of the Server

18.3.2 Creating a Client

The steps to create a client class are as follows:

1. Create a socket as a `Socket` object
2. Create input and output streams to receive and send data respectively
3. Perform communication with the server
4. Close the socket

Code Snippet 6 shows the use of the `Socket` class to create a client.

Code Snippet 6:

```
package com.io.demo;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class Client {
    public static void main(String[] args) {
        try {
            Socket clientSocket = new Socket("localhost", 6060);
            InputStream inFromServer = clientSocket.getInputStream();
            DataInputStream in = new DataInputStream(inFromServer);
            System.out.println("Message received from server: " + in.readUTF());
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The code creates a `Client` class. In the `main()` method, the code creates a `Socket` object passing the IP address and port of the server. The code then constructs a `DataInputStream` from an `InputStream` object and uses it to write out the message received from the server.

Figure 18.7 displays the output of the client.

```
[Output]
SocketProgramming (run) <--> SocketProgramming (run) #2
run:
Message received from server: Hello from server.
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 18.7: Output of the Client

18.3.3 URL Processing

URL is an address of a resource in the Internet. Java provides the `URL` and `URLConnection` classes to write Java programs that communicate with a URL. Using both the `URL` and `URLConnection` classes, you can create a URL to a Web page can be created, establish a connection to the Web page, and retrieve the Web page content.

Code Snippet 7 shows the use of the `URL` and `URLConnection` classes.

Code Snippet 7:

```
package com.io.demo;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
public class Client {
    public static void main(String[] args) {
        try {
            Socket clientSocket = new Socket("localhost", 6060);
            InputStream inFromServer =
                clientSocket.getInputStream();
```

```
        DataInputStream in = new DataInputStream(inFromServer);

        System.out.println("Message received from server: " +
                           in.readUTF());

        clientSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

In the main() method, the code creates a URL object for the `http://www.google.com` URL. The `getConnection()` method returns a `URLConnection` object that is casted to a `HttpURLConnection` object. The `getInputStream()` method of `URLConnection` retrieves an `InputStream` that is used to create a `InputStreamReader` object. Then, the code creates a `BufferedReader` object of `InputStreamReader` and uses the `readLine()` method inside a `while` loop to retrieve each line of content from the `http://www.google.com` URL. Finally, the retrieved content is printed out.

Figure 18.8 displays the output of the URLProcessingDemo class.

```
Output - URLProcessingDemo (run)
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
[20]
[21]
[22]
[23]
[24]
[25]
[26]
[27]
[28]
[29]
[30]
[31]
[32]
[33]
[34]
[35]
[36]
[37]
[38]
[39]
[40]
[41]
[42]
[43]
[44]
[45]
[46]
[47]
[48]
[49]
[50]
[51]
[52]
[53]
[54]
[55]
[56]
[57]
[58]
[59]
[60]
[61]
[62]
[63]
[64]
[65]
[66]
[67]
[68]
[69]
[70]
[71]
[72]
[73]
[74]
[75]
[76]
[77]
[78]
[79]
[80]
[81]
[82]
[83]
[84]
[85]
[86]
[87]
[88]
[89]
[90]
[91]
[92]
[93]
[94]
[95]
[96]
[97]
[98]
[99]
[100]
[101]
[102]
[103]
[104]
[105]
[106]
[107]
[108]
[109]
[110]
[111]
[112]
[113]
[114]
[115]
[116]
[117]
[118]
[119]
[120]
[121]
[122]
[123]
[124]
[125]
[126]
[127]
[128]
[129]
[130]
[131]
[132]
[133]
[134]
[135]
[136]
[137]
[138]
[139]
[140]
[141]
[142]
[143]
[144]
[145]
[146]
[147]
[148]
[149]
[150]
[151]
[152]
[153]
[154]
[155]
[156]
[157]
[158]
[159]
[160]
[161]
[162]
[163]
[164]
[165]
[166]
[167]
[168]
[169]
[170]
[171]
[172]
[173]
[174]
[175]
[176]
[177]
[178]
[179]
[180]
[181]
[182]
[183]
[184]
[185]
[186]
[187]
[188]
[189]
[190]
[191]
[192]
[193]
[194]
[195]
[196]
[197]
[198]
[199]
[200]
[201]
[202]
[203]
[204]
[205]
[206]
[207]
[208]
[209]
[210]
[211]
[212]
[213]
[214]
[215]
[216]
[217]
[218]
[219]
[220]
[221]
[222]
[223]
[224]
[225]
[226]
[227]
[228]
[229]
[230]
[231]
[232]
[233]
[234]
[235]
[236]
[237]
[238]
[239]
[240]
[241]
[242]
[243]
[244]
[245]
[246]
[247]
[248]
[249]
[250]
```

Figure 18.8: Output of the WBLIPProcessingDemo Class

18.4 Check Your Progress

1. Which of the following options is a class-level Javadoc tag?

A.	<code>@param</code>
B.	<code>@throws</code>
C.	<code>@deprecated</code>
D.	<code>@return</code>

2. Which of the following statements is true regarding the TCP/IP?

A.	TCP/IP is a connectionless protocol.
B.	TCP/IP ensures packet delivery.
C.	TCP/IP is faster than UDP.
D.	TCP/IP has a maximum packet size of 65507 bytes.

3. Which of the following methods of the `ServerSocket` class waits for a client connection?

A.	<code>accept()</code>
B.	<code>await()</code>
C.	<code>connect()</code>
D.	<code>bind()</code>

4. Identify the correct method of the `Socket` class that a client uses to connect with a `ServerSocket`?

A.	<code>connect(InetAddress host, int timeout)</code>
B.	<code>connect(SocketAddress host, int timeout)</code>
C.	<code>connect(InetAddress host, int port, int timeout)</code>
D.	<code>connect(SocketAddress host, int timeout, int port)</code>

5. Which of the following code snippet correctly creates a `URLConnection` object?

A.	<code>URL url = new URL("http://example.com"); URLConnection connection = new URLConnection(url);</code>
B.	<code>URL url = new URL(); URLConnection urlConnection = url. openConnection("http://example.com");</code>
C.	<code>URL url = new URL("http://example.com"); URLConnection urlConnection = url.openConnection();</code>
D.	<code>URLConnection connection = new URLConnection("http:// example.com");</code>

18.4.1 Answers

1	C
2	B
3	A
4	B
5	C

Summary

- The Javadoc tool relies on documentation tags present in the source code to create API documentation.
- Javadoc can be generated using the Javadoc tool from the command line or the in-built Javadoc options of NetBeans.
- Classes and interfaces of the `java.net` package supports network programming.
- Socket programming over UDP is supported by the `DatagramPacket` and `DatagramSocket` classes.
- Socket programming over TCP is supported by the `Socket` and `ServerSocket` classes of the `java.net` package.
- URL processing can be done by the `URL` and `URLConnection` classes.