

M.Sc. (IT) Final Year
MIT - 10

ADVANCED JAVA



मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल
MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL

Reviewer Committee

1. Dr. Amit Kumar Mandle
Assistant Professor
IEHE, Bhopal
2. Dr. Romsha Sharma
Professor
Shri Sathya Sai College for Women, Bhopal
3. Dr. Sharad Gangale
Professor
RKDF University, Bhopal

Advisory Committee

1. Dr. Jayant Sonwalkar
Hon'ble Vice Chancellor
Madhya Pradesh Bhoj (Open) University, Bhopal
2. Dr. L.S. Solanki
Registrar
Madhya Pradesh Bhoj (Open) University, Bhopal
3. Dr. Kishor John
Director
Madhya Pradesh Bhoj (Open) University, Bhopal
4. Dr. Amit Kumar Mandle
Assistant Professor
IEHE, Bhopal
5. Dr. Romsha Shrama
Professor
Shri Sathya Sai College for Women, Bhopal
6. Dr. Sharad Gangale
Professor
RKDF University, Bhopal

COURSE WRITERS

Rohit Khurana, CEO, ITL Education Solutions Ltd., 2nd Floor, GD-ITL Tower, Netaji Subhash Place, Pitampura, New Delhi

Units (1.0-1.2.1, 1.2.4, 1.3-1.3.1, 1.3.3, 1.4, 1.4.1, 1.4.3, 1.4.4, 1.4.6, 1.4.7, 1.4.8-1.5.3, 1.5.5, 1.5.6-1.11, 2.0-2.2, 2.2.4, 2.2.6, 2.2.9, 2.3.1, 2.3.3-2.3.4, 2.3.7-2.3.8, 2.4, 2.6.1-2.6.4, 3.3-3.3.2, 3.3.3-3.3.5, 3.3.6-3.5.1, 3.5.2-3.5.3, 3.6.3-3.14, 4.4, 4.4.3-4.4.5, 4.5.3, 4.6-4.6.1, 4.6.4-4.6.5, 4.7-4.7.1, 5.2.1-5.2.2, 5.2.3, 5.4, 5.4.1-5.5, 5.5.1, 5.5.2, 5.6, 5.6.1-5.7, 5.7.1, 5.8-5.12)

Kalapatapu Kalyani, Asstt. Professor, Matrusri Institute of Post Graduate Studies, Affiliated to Osmania University & Recognized by AICTE, Hyderabad

Units (1.2.2-1.2.3, 1.4.2, 1.4.5, 1.5.4, 2.2.1-2.2.3, 2.2.5, 2.2.7, 2.6, 2.6.5-2.6.6, 3.0-3.2, 3.6)

Rashmi Kanta Das, Microsoft Certified Systems Engineer, Senior Faculty, Leading Coaching Institutes, Bhubaneswar, Odissa.

Units (1.2.5, 1.3.2, 2.2.8, 2.2.10, 2.3, 2.3.2, 2.3.5, 2.3.6, 2.3.9, 2.5, 2.6.7-2.11, 3.6.1, 4.0-4.3, 4.4.1-4.4.2, 4.5-4.5.2, 4.5.4, 4.6.2-4.6.3, 4.7.2-4.12, 5.0-5.2, 5.3, 5.7.2-5.12)

Copyright © Reserved, Madhya Pradesh Bhoj (Open) University, Bhopal

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Registrar, Madhya Pradesh Bhoj (Open) University, Bhopal

Information contained in this book has been published by VIKAS® Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Madhya Pradesh Bhoj (Open) University, Bhopal, Publisher and its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.

Published by Registrar, MP Bhoj (open) University, Bhopal in 2020



VIKAS® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.

E-28, Sector-8, Noida - 201301 (UP)

Phone: 0120-4078900 • Fax: 0120-4078999

Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44

• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

SYLLABI-BOOK MAPPING TABLE

Advanced Java

Syllabi	Mapping in Book
<p>UNIT-I The Genesis of Java, Introduction and Creation, Applets and Applications, Security, Bytecodes, Java Buzzwords, Simple, Multi-Threaded, Architecture Neutral, Java and JavaScript, New in JDK, An Overview of Java, What is an Object, Features of Object Oriented Programming, the First Simple Programme, Compiling, Data Types, Variables and Arrays, Data Types in Java, Literals, Characters, Variable Declaration, Symbolic Constants, Type Casting, Arrays, Vectors, Array Declaration Syntax, Operating in Java, Arithmetic Operators, Basic Assignment Operators, Relational Operators, Boolean Logical Operators, Ternary Operator, Operator Precedence, Control Statements, Java's Selection Statements, Switch, Nested Switch, Iteration Constructs, Continue, Return.</p>	<p>Unit 1: Overview of Java, Data Types and Variables, Arrays, Operators and Control Statements (Pages: 3-63)</p>
<p>UNIT-II Class an Introduction, What is a Class, What are Methods, Methods and Classes in Details, Methods Overloading, Constructor Overloading, Objects as Parameters, Returning Objects, Recursion, Access Control/Visibility, Understanding Static, Final, Nested and Inner Classes, the String Class, Command Line Arguments, Inheritance, Inheritance Basic, Member Access and Inheritance, Super Class Variable and Sub Class Object, Using Super to Call Superclass Constructors, Another Use of Super, Multilevel Hierarchy, Calling Constructor, Overriding Methods, Abstract Classes Method, Final and Inheritance, Object Class, Interfaces and Packages, Defining Interface, What is a Package, Class Path Variable, Access Protection, Important Packages, Exception Handling, Fundamentals of Exception Handling, Types of Exceptions, Uncaught Exceptions, Try and Catch Keywords, Throw, Throws and Finally, Nested Try Statements, Java Built-In Exceptions, User Defined Exceptions.</p>	<p>Unit 2: Class, Inheritance, Interfaces, Packages and Exception Handling (Pages: 65-154)</p>
<p>UNIT-III Multithreaded Programming, the Java Thread Model, Priorities, Synchronization, Messaging, Thread Class and Runnable Interface, Creation of Threads, Creating Multiple Threads, Synchronization and Deadlock, Suspending, Resuming and Stopping Threads, Applets and Input/Output, Input/Output Basics, Streams (Byte and Character), Reading From and Writing To Console, Reading and Writing Files, <code>PrintWriter</code> Class, Fundamentals of Applets, Transient and Volatile Modifier, <code>Strictfp</code>, Native Methods, Problems with Native Methods, Handling Strings, String Length, Operations on Strings, Extract Character Methods, String Comparison Methods, Searching and Modifying, Data Conversion and Value of () Methods, Changing Case of Characters, String Buffer, Exploring Java. Lang, Wrapper Classes and Simple Type Wrappers, Void, Abstract Process Class, Runtime Class and Memory Management, Other Programme Execution, System Class, Environment Properties, Using <code>Clone ()</code> and <code>Cloneable ()</code> Interfaces, Class, Class Loader, Math Class, Thread, Thread Group and Runnable Interface, Throwable Class, Security Manager, <code>java.lang.ref</code> and <code>java.lang.reflect</code> Packages, Java . Util - The Utility Classes, the Enumeration Interface, Vector, Stack, Dictionary, Hash Table, Properties, Using the <code>Store ()</code> and <code>Load ()</code>, String Tokenizer, Bitset Class, Date and Date Comparison, Time Zones, Random Class, Observe.</p>	<p>Unit 3: Multithreaded Programming, Applets, Handling String, <code>java.lang</code> and Utility Classes (Pages: 155-299)</p>

UNIT – IV

Input/Output Classes, File in Java, Directory, File Name, Filter Interface, Creating Directory, the Stream Classes, Input Stream and Output Stream, File Input Stream and File Output Stream, Byte Array Input Stream and Byte Array Output Stream, Filtered Byte Stream, Buffered Byte Stream, Print Stream, Random Access File, Stream Tokenizer, Stream Benefits, **Networking**, Basic of Networking, Proxy Server, Domain Naming Services, Networking Classes and Interfaces, InetAddress Class, TCP/IP Sockets, Datagram Packet, Network, **Applet Class**, Applet Basics, Applet Life Cycle, a Simple Banner Applet, Handling Events, `getDocumentBase()`, `getCodeBase()`, `showDocumentBase()`, Audio Clip and Applet Stub Interface, **AWT: Windows, Graphics and Text**, AWT Classes, Window Fundamentals, Working With Frame Windows, Frame Window in An Applet, Event Handling in a Frame Window, Window Program, Displaying Information while Working with Graphics and Color, Working with Fonts, Managing Text Output using Font Metrics, Exploring Text and Graphics, **AWT: Controls, Layouts and Menus**, Control Fundamentals, Layouts, Menus, Dialog Class, Other Controls.

Unit 4: Input/Output Classes,
Networking, AWT Graphics and
Text, Controls, Layouts and Menus
(Pages: 301-418)

UNIT - V

Images, File Formats, Image Fundamentals, ImageObserver, MediaTracker, **JDBC**, JDBC Introduction to Class and Methods, Register, Driver, Establish a Session, Execute a Query, ResultSet, Closing the Session, **Swings**, JAPPLET, **Java Beans**, What is a Java Bean? Advantages of Java Beans, Application Builder Tools, the Bean Developer Kit (BDK), JAR Files, Introspection, Developing a Simple Bean, Using Bound Properties, Using the Bean Info Interface, Constrained Properties, Persistence, Customisers, **The Basic Servlet API**, the Get Method, the POST Method, Mime Content Types, **Java and CORBA Connectivity**, the Compatibility Problem, an Overview of IDL and IIOP, Working with CORBA System, CORBA Servers, CORBA Clients, a Simple CORBA Service, Legacy Applications and CORBA.

Unit 5: Images, JDBC, Java Beans,
Servlet API and CORBA
Connectivity
(Pages: 419-521)

CONTENTS

INTRODUCTION	1-2
UNIT 1 OVERVIEW OF JAVA, DATA TYPES AND VARIABLES, ARRAYS, OPERATORS AND CONTROL STATEMENTS	3-63
1.0 Introduction	
1.1 Objectives	
1.2 Introduction and Creation of Java	
1.2.1 Java Applets	
1.2.2 Bytecodes	
1.2.3 Java Buzzwords	
1.2.4 Java and JavaScript	
1.2.5 Generics of Java	
1.3 Overview of Java	
1.3.1 What is Object?	
1.3.2 Features of Object Oriented Programming	
1.3.3 Java: Simple Program and Compiling	
1.4 Data Types in Java	
1.4.1 Literals	
1.4.2 Characters	
1.4.3 Variables Declaration	
1.4.4 Symbolic Constants	
1.4.5 Type Casting	
1.4.6 Arrays	
1.4.7 Array Declaration Syntax	
1.4.8 Vectors	
1.5 Operators in Java	
1.5.1 Arithmetic Operators	
1.5.2 Basic Assignment Operators	
1.5.3 Relational Operators	
1.5.4 Boolean Logical Operators	
1.5.5 Ternary Operators	
1.5.6 Operator Precedence	
1.6 Control Statements	
1.6.1 Nested Switch	
1.6.2 Iteration Constructs and Return	
1.7 Answers to ‘Check Your Progress’	
1.8 Summary	
1.9 Key Terms	
1.10 Self Assessment Questions and Exercises	
1.11 Further Reading	
UNIT 2 CLASS, INHERITANCE, INTERFACES, PACKAGES AND EXCEPTION HANDLING	65-154
2.0 Introduction	
2.1 Objectives	
2.2 Introduction to Class	
2.2.1 Method and Classes	
2.2.2 Method and Constructor Overloading	

- 2.2.3 Objects as Parameters
- 2.2.4 Returning Objects
- 2.2.5 Recursion
- 2.2.6 Access Control/ Visibility
- 2.2.7 Static and Final Classes
- 2.2.8 Nested and Inner Classes
- 2.2.9 String Class
- 2.2.10 Command Line Arguments
- 2.3 Inheritance
 - 2.3.1 Member Access
 - 2.3.2 Super Class Variable
 - 2.3.3 Subclass Object
 - 2.3.4 Using Super to Call Superclass Constructors
 - 2.3.5 Multilevel Hierarchy
 - 2.3.6 Calling Constructor
 - 2.3.7 Overriding Methods
 - 2.3.8 Abstract Classes Method
 - 2.3.9 Final Class in Inheritance
- 2.4 Interface
- 2.5 Packages
- 2.6 Fundamentals of Exception Handling
 - 2.6.1 Types of Exception
 - 2.6.2 Try and Catch Keyword
 - 2.6.3 Finally Keywords
 - 2.6.4 Throw and Throws
 - 2.6.5 Nested Try Statements
 - 2.6.6 Java Build-In Exceptions
 - 2.6.7 User Defined Exceptions
- 2.7 Answers to 'Check Your Progress'
- 2.8 Summary
- 2.9 Key Terms
- 2.10 Self Assessment Questions and Exercises
- 2.11 Further Reading

**UNIT 3 MULTITHREADED PROGRAMMING, APPLETS,
HANDLING STRING, `java.lang` AND UTILITY CLASSES**

155-299

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Multithreaded Programming
- 3.3 Java Thread Model
 - 3.3.1 Thread Priorities
 - 3.3.2 Synchronization Messaging
 - 3.3.3 Thread Class
 - 3.3.4 Runnable Interface
 - 3.3.5 Creating Multiple Threads
 - 3.3.6 Suspending, Resuming and Stopping Threads
- 3.4 Basic Input/Output
 - 3.4.1 Streams (Byte and Character)
 - 3.4.2 Reading From and Writing To Console
 - 3.4.3 Reading and Writing Files
 - 3.4.4 `PrintWriter` Class

- 3.5 Fundamentals of Applets
 - 3.5.1 Transient and Volatile Modifier
 - 3.5.2 Modifier Strictfp
 - 3.5.3 Native Interface
- 3.6 String Handling
 - 3.6.1 Operations on String and Extract Character Methods
 - 3.6.2 StringBuffer
- 3.7 Wrapper Classes
 - 3.7.1 Memory Management
 - 3.7.2 java.lang Environment Properties
 - 3.7.3 Security Manager and SecurityManager Class
- 3.8 Java Utility Class
- 3.9 Enumeration Interface
 - 3.9.1 Using Store () and Load ()
- 3.10 Answers to 'Check Your Progress'
- 3.11 Summary
- 3.12 Key Terms
- 3.13 Self Assessment Questions and Exercises
- 3.14 Further Reading

UNIT 4 INPUT/OUTPUT CLASSES, NETWORKING, AWT GRAPHICS AND TEXT, CONTROLS, LAYOUTS AND MENUS

301-418

- 4.0 Introduction
- 4.1 Objectives
- 4.2 File and File Name in Java
 - 4.2.1 Directory and Creating Directory
- 4.3 Stream Classes
- 4.4 Basic of Networking
 - 4.4.1 Proxy Server
 - 4.4.2 Domain Naming Services
 - 4.4.3 Networking Classes and Interfaces
 - 4.4.4 InetAddress Class
 - 4.4.5 Datagram Packet Network
- 4.5 Applet Basic
 - 4.5.1 Applet Life Cycle
 - 4.5.2 Simple Banner Applet
 - 4.5.3 Handling Events
 - 4.5.4 AudioClip
- 4.6 AWT Classes
 - 4.6.1 Window Fundamentals
 - 4.6.2 Working With Frame Windows
 - 4.6.3 Frame Window and Event Handling in a Frame Window
 - 4.6.4 Display Information While Working with Graphics and Color
 - 4.6.5 Working with Fonts
- 4.7 AWT Controls and Layout Managers
 - 4.7.1 AWT Menus
 - 4.7.2 Dialog Class
- 4.8 Answers to 'Check Your Progress'
- 4.9 Summary

- 4.10 Key Terms
- 4.11 Self Assessment Questions and Exercises
- 4.12 Further Reading

UNIT 5 IMAGES, JDBC, JAVA BEANS, SERVLET API AND CORBA CONNECTIVITY

419-521

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Images in Java
 - 5.2.1 File Formats
 - 5.2.2 Image Fundamentals
 - 5.2.3 ImageObserver and MediaTracker
- 5.3 JDBC: An Introduction
- 5.4 Swings
 - 5.4.1 Components of Swing
- 5.5 Java Beans
 - 5.5.1 What is Java Beans?
 - 5.5.2 JAR Files and Introspection
- 5.6 Basic Servlet API
 - 5.6.1 MIME Content Types
- 5.7 CORBA Connectivity in Java
 - 5.7.1 Working CORBA System
 - 5.7.2 Simple CORBA Service
- 5.8 Answers to 'Check Your Progress'
- 5.9 Summary
- 5.10 Key Terms
- 5.11 Self Assessment Questions and Exercises
- 5.12 Further Reading

INTRODUCTION

Java is a high-level, class-based, Object-Oriented Programming (OOP) language that is specifically designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers Write Once, Run Anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but Java has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities, such as reflection and runtime code modification that are typically not available in traditional compiled languages.

Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle) and released in 1995 as a core component of Sun Microsystems' Java platform. The original and reference implementation Java compilers, virtual machines, and class libraries were originally released by Sun under proprietary licenses. As of October 2021, Java 17 is the latest version. The Java 8, Java 11 and Java 17 are the current Long-Term Support (LTS) versions.

The Java platform consists of several programs, each of which provides a portion of its overall capabilities. For example, the Java compiler, which converts Java source code into Java bytecode, an intermediate language for the JVM, is provided as part of the Java Development Kit (JDK). The Java Runtime Environment (JRE), complementing the JVM with a Just-In-Time (JIT) compiler, converts intermediate bytecode into native machine code on the fly. The Java platform also includes an extensive set of libraries. The essential components in the platform are the Java language compiler, the libraries, and the runtime environment in which Java intermediate bytecode executes according to the rules laid out in the virtual machine specification.

Java is divided into two parts, i.e., Core Java (J2SE) and Advanced Java (JEE). The core Java part covers the fundamentals (data types, functions, operators, loops, thread, exception handling, etc.) of the Java programming language and is used to develop general purpose applications. Whereas Advanced Java covers the standard concepts, such as database connectivity, networking, Servlet, Web-Services, JSP (Java Server Pages), JDBC (Java DataBase Connectivity), RMI (Remote Method Invocation), Socket Programming, etc. It is a specialization in specific domain.

This book is divided into five units which explains the genesis and basic concepts of Java, class, methods, inheritance, multilevel hierarchy, overriding methods, interfaces and packages, exception handling, Java built-in exceptions, user defined exceptions, multithreaded programming, Java thread model, runnable interface, synchronization and deadlock, Applets and Input/Output (I/O), handling strings, exploring Java.Lang, packages, Java.Util class, I/O classes, networking, interfaces, Applet class, Applet life cycle, AWT classes and controls, JDBC, Java

NOTES

NOTES

Beans, Bean Developer Kit (BDK), JAR files, Basic Servlet API, Java and CORBA Connectivity. The book follows the Self-Instruction Mode or the SIM format wherein each unit begins with an 'Introduction' to the topic followed by an outline of the 'Objectives'. The content is presented in a simple and structured form interspersed with Answers to 'Check Your Progress' for better understanding. A list of 'Summary' along with a 'Key Terms' and a set of 'Self-Assessment Questions and Exercises' is provided at the end of the each unit for effective recapitulation.

UNIT 1 OVERVIEW OF JAVA, DATA TYPES AND VARIABLES, ARRAYS, OPERATORS AND CONTROL STATEMENTS

Overview of Java, Data Types and Variables, Arrays, Operators and Control Statements

NOTES

Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Introduction and Creation of Java
 - 1.2.1 Java Applets
 - 1.2.2 Bytecodes
 - 1.2.3 Java Buzzwords
 - 1.2.4 Java and JavaScript
 - 1.2.5 Generics of Java
- 1.3 Overview of Java
 - 1.3.1 What is Object?
 - 1.3.2 Features of Object Oriented Programming
 - 1.3.3 Java: Simple Program and Compiling
- 1.4 Data Types in Java
 - 1.4.1 Literals
 - 1.4.2 Characters
 - 1.4.3 Variables Declaration
 - 1.4.4 Symbolic Constants
 - 1.4.5 Type Casting
 - 1.4.6 Arrays
 - 1.4.7 Array Declaration Syntax
 - 1.4.8 Vectors
- 1.5 Operators in Java
 - 1.5.1 Arithmetic Operators
 - 1.5.2 Basic Assignment Operators
 - 1.5.3 Relational Operators
 - 1.5.4 Boolean Logical Operators
 - 1.5.5 Ternary Operators
 - 1.5.6 Operator Precedence
- 1.6 Control Statements
 - 1.6.1 Nested Switch
 - 1.6.2 Iteration Constructs and Return
- 1.7 Answers to ‘Check Your Progress’
- 1.8 Summary
- 1.9 Key Terms
- 1.10 Self Assessment Questions and Exercises
- 1.11 Further Reading

1.0 INTRODUCTION

Java, initially, named ‘Oak’ was developed by a team which was headed by James Gosling, at Sun Microsystems, USA in 1991. Java is a third generation programming language which implements the concepts of Object-Oriented Programming (OOP).

NOTES

It inherits most of the features of the existing languages, C and C++. Also it has new features to make it one of the simple object oriented languages that is easy to learn. One of the most important features of Java is that it is a language independent of a platform, that is, a program written for one system can be executed on any other system. This feature of Java makes it a popular language for developing Internet-based applications.

A data type determines the type of operations that can be performed on the data. Java provides various data types and each is represented differently within a computer's memory. Operators are symbols which perform operations on various data items known as operands. For example, in $a + b$, a and b are operands and $+$ is an operator. Note that to perform an operation, operators and operands are combined together forming an expression. Arrays are defined as a sequence of the same type of data elements of a fixed size. These data elements can be primitive or non-primitive data types. The elements of an array are stored in contiguous memory locations and each individual element can be accessed using one or more indices or subscripts.

By default, statements are executed in the same order in which they appear in the program and each statement is executed only once. However, the serial execution of statements makes a program inflexible and unsuitable for most practical applications. To make a program more flexible, control statements are used to alter the flow of control of the program. In Java, the control statements are broadly classified into three categories, namely conditional statements, iteration statements and jump statements. All these control statements are commonly used with logical tests or test conditions to alter the flow of control conditionally or unconditionally.

In this unit, you will study about the introduction and creation of Java, Java applets and applications, security, bytecodes, Java buzzwords, multi-threaded, architecture neutral, Java and JavaScript, genesis of Java, overview of Java, object, features of object oriented programming, first Java program and compiling, data types and literals, characters, variables declaration and symbolic constants, type casting, arrays, vectors, arrays declaration syntax, arithmetic operators, assignment operators, relational operators, Boolean logical operators, ternary operators, operator precedence, control statements, selection statements, switch and nested switch, iteration constructs, continue and return.

1.1 OBJECTIVES

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

- Discuss the introduction and creation of Java
- Understand the Java applets and applications
- Describe the security and bytecodes
- Explain the Java buzzwords and multi-threaded
- Define the architecture neutral, Java and JavaScript
- Elaborate the genesis of Java
- Understand the overview of Java and object

- Elaborate the features of object oriented programming
- Analyse the first Java program and compiling
- Explain the data types and literals
- Define the characters, variable declaration and symbolic constants
- Elaborate the type casting , arrays and vectors
- Discuss the array declaration syntax and arithmetic operators
- Describe the basic assignment operators and relational operators
- Define the Boolean logical operators and ternary operators
- Explain the operator precedence and control statements
- Analyse the selection statement, switch and nested switch
- Understand the iteration constructs, continue and return

NOTES

1.2 INTRODUCTION AND CREATION OF JAVA

Java, initially, named ‘Oak’ was developed by a team which was headed by James Gosling, at Sun Microsystems, USA in 1991.

History

One of the reasons for the development of Java was the need for a software that would not be dependent on any platform and would be portable enough to be embedded in electronic devices like remote controls and microwave ovens. Another reason that led to the growth of Java was the Internet and the media’s need for portable programs that would not be dependent on any platform. Gosling and other team members developed Web applets by using a new language that could run on all types of computers. In 1993, the first Web browser named ‘HotJava’ was developed to locate and run applet programs. This development made Java language popular on the Internet. By the year 1996, Java became a general-purpose, object-oriented programming language which was used for the Internet programming. Soon, Java became popular and many Web browsers like Internet Explorer and Netscape Navigator incorporated the ability to run Java applets.

Features

Java has become a popular language for Internet applications because of the following features:

- **It is Simple:** Java inherits the syntax of C/C++ and many of the OOPs features of C++. Thus, one can understand the concepts of object-oriented language and learn Java with minimum effort. Moreover, Java omits the complex and unreliable codes of C and C++. These codes include operator overloading, pointers and preprocessor header files. Java provides a short and convenient means to accomplish a given task.
- **It can be Easily Interpreted:** Unlike other languages Java uses a system with both a compiler and an interpreter for program execution. First, the compiler converts the program code to bytecode which in turn is converted

NOTES

to machine code on any machine, using the interpreter. The machine code thus generated can be executed irrespective of the system on which it is being executed.

- **It has Neutral Architecture (Independent of Platform):** This feature makes Java very special. Java programs can run on any platform, which means that they can run on different CPUs and operating system architectures. The bytecode produced by the Java compiler can run on any machine which has Java run-time environment.
- **It is an Object-Oriented Language:** Java is an object-oriented language as it bounds code and data together in the form of objects. The objects and classes contain the program code and data. The Java object model is easily extensible and classes can be used anywhere in the program in the form of packages.
- **It is Robust:** Java is a robust language because of two main reasons, first, it is a language typed strictly that checks the code at the time of compilation; second, it manages memory in an effective way. In C++, the programmer has to manually de-allocate the dynamic memory used by objects. Java does this automatically (with the help of a feature known as garbage collector).
- **It can be Distributed:** Since Java is not dependent on any platform, it is suitable for developing applications for networks. Java can handle TCP/IP protocols and hence applications developed in it can access remote objects on the Internet like any object on a local system.
- **It is Multithreaded:** Java supports multithreaded programming which allows us to write a program that can perform more than one task simultaneously. A user need not to wait for one program to finish a task, before starting the next task for example, a user can listen to an audio clip while downloading the applet. This feature helps to improve the performance of graphical applications.
- **Its Quality of Performance is High:** As stated earlier, a Java program is converted to bytecode which is then converted to machine code using an interpreter. Since bytecode is highly optimized, it enables the JVM to execute programs at a faster rate.
- **It is dynamic:** Java programs can link to new class libraries, objects, methods, etc., at the run-time. Java also provides the facility to include functions of other languages like C and C++. These are referred to as native methods. These methods are also linked dynamically at run-time.

Java versus C++

Both C++ and Java are object-oriented languages but they are very different from each other. Some of the features of C++ were deliberately removed and other new features were added to make Java more flexible and reliable. Some of the differences between Java and C++ are as follows:

- Java does not support multiple inheritances of classes directly.
- The concept of multithreading is supported by Java.

- The ‘Destructor’ function in Java is replaced by the ‘finalize’ method.
- The keyword ‘typedef’ is not supported by Java.
- Java does not support pointers, instead it uses implicit object references.
- The virtual keyword is not supported in Java.
- Java does not support the concept of global variables.
- Java supports exception handling in a different way than C++. It provides a final clause for a clean-up.
- Non-primitive data types are allocated memory by using the new operator.
- Java adds many features that are necessary for object-oriented programming.

NOTES

Java Virtual Machine

As discussed earlier, Java uses both compiler and interpreter. The source code written in Java is compiled to generate bytecode and then this bytecode is interpreted to machine instructions for a specific machine. The bytecode generated by the compiler is not machine specific. It is generated for a virtual machine known as JVM (Java Virtual Machine). It exists only inside the computer memory. This virtual machine is designed in such a way that it can be implemented on any existing processor and itself acts as a virtual processor chip. It hides the underlying operating system details from Java applications.

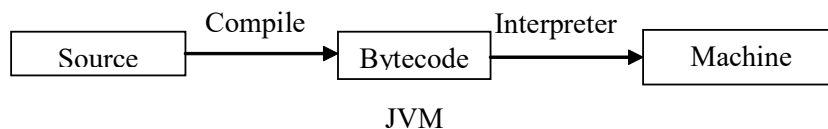


Fig. 1.1 Execution of a Java Program

Java Development Kit

The JDK (Java Development Kit) consists of various tools that are used to develop and execute Java programs. The tools included in JDK are listed in Table 1.1.

Table 1.1 Tools in Java Development Kit

Tool	Function
javac	Java compiler that converts source code to Java bytecode
java	Java interpreter that interprets class files generated by Java compiler and converts it to machine code
javadoc	Document generator which automatically generates documentation from source code
javah	Generates C headers and a stub generator used for writing native methods
javap	This class file disassembler enables to convert bytecode files to a program description
jdb	Java debugger which helps in tracking errors in the program
appletviewer	Used for running and debugging Java applets without a Web browser

NOTES

1.2.1 Java Applets

In recent years, Java has become a popular language for the programs that are required to run on different systems. Java's new innovation named 'applet' has completely changed Internet programming. Applets are tiny programs that are designed in such a way that they can be transmitted over the Internet. They can be downloaded on demand and executed automatically by the Java compatible web browser. They are used to handle user input, data supplied by the server and simple functions that execute locally on the client machine.

Applet is a dynamic, self-executing program and is intelligent enough to change it with the user inputs. The dynamic programs, when downloaded and executed, can cause serious harm to the computer as it may contain viruses like Trojan horse and other malicious programs. These programs may search for the contents on the local file system of the client computer and may gather private information like credit card numbers, passwords, etc. Earlier, viruses were scanned before executing the downloaded program but Java has resolved the issue by confining the Java programs to Java execution environment only.

Java Compiler

In most of the programming languages, the program is converted to the machine code either by using the compiler or interpreter (Figure 3.1). The machine code so generated is machine-dependent, that is, it may not run on any other machine than the one on which it is generated. Unlike other programming languages, the Java compiler does not convert source code to machine code, it converts source code to a special intermediate code known as bytecode. The bytecode so generated are in the form of class files that can be interpreted. The command used for compilation in Java is `javac` which converts the corresponding Java file into class file. The bytecode is machine-independent, that is, it can be run on any machine with the help of a Java virtual machine (JVM). Figure 1.2 shows the compilation of a Java program.

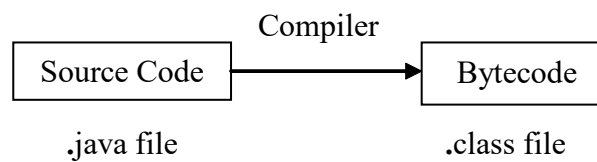


Fig. 1.2 Compilation of a Java Program

1.2.2 Bytecodes

You have just learned that the output of a Java compiler is not an executable code. Rather, it is a bytecode. In Java bytecode is a highly optimized set of instructions planned to be executed by the Java run-time system, which is referred to as the Java Virtual Machine (JVM). A Java program executed by the JVM helps solve problems associated with Web-based programs. Since the JVM is in control, it can hold the program and check it from generating side effects outside the system.

1.2.3 Java Buzzwords

As mentioned earlier, portability and security is the innovation of Java. Some additional factors too played significant roles in moulding the final form of Java. The list of buzzwords summed up by the Java team is as follows:

- **Simplicity:** Programmers find it easy to write applications as it avoids the use of the concepts of C/C++, such as pointers, operator overloading, multiple inheritance, etc. String manipulations can be implemented very easily without any explicit concatenation procedure.
- **Portability:** Java is famous for its unique feature—platform-independence (architecture-neutral). This means that a Java program compiled on one machine could be ported to any other machine/operating system and executed without any modifications. Thus, the class file, which is the result of compilation say on a DOS platform, can run on a UNIX platform unlike the .exe file of C/C++. This is an important feature for which Java is preferred for Internet applications.
- **Robust and Object-Oriented:** Java is a highly object-oriented language where reusability is of utmost importance. Java programs are very reliable on different platforms with the special features of memory allocation and de-allocation, and exception handling. Memory management, especially de-allocation, is taken care of by the Java environment, which is not the case in C/C++, where programmers have to deal with it explicitly with extra code. Exception handling is a mechanism that helps the execution of a code, even if an error occurs at run-time, by handling the exception.

Robustness is also achieved because Java is a strongly typed language. It signifies that you must declare the variable type before you use it. This is different from languages, such as PERL, JavaScript, etc., which are loosely typed.

- **Multithreaded:** Java with its multithreaded approach can run many programs concurrently, thereby saving processor time. Synchronization of code is an added feature of Java to run non-erroneous interactive applications.
- **Distributed and Dynamic:** Java is also popular for its distributed environment, as it supports the Transmission Control Protocol/Internet Protocol (TCP/IP). With Java, you can access a Uniform Resource Locator (URL) or a file on a remote server in some other country with the same ease, as you can access a file on your local system.

Java can also validate the code at run-time, which is more important for applets. Therefore, it is feasible to dynamically connect the code in a secure and practical manner.

1.2.4 Java and JavaScript

JavaScript was the first client-side Web scripting language. It first appeared in 1995 in Netscape 2.0. You can use JavaScript directly within a Web page without using any special tools to create or compile scripts and it works on most of today's

NOTES

NOTES

browsers. Despite the name, JavaScript has little to do with the Java language. Some of the commands are similar but it is a different language with a different purpose.

JavaScript can be defined of as an extension to HTML, which allows authors to incorporate some functionality in their Web pages. So now, whenever the user presses the submit button, you do not necessarily have to invoke a Common Gateway Interface (CGI) script to do the processing. Simply, you can do the processing locally using JavaScript and give back the results. JavaScript can also be used in a number of ways to spice up your Web page. You can, for example, use JavaScript to change a button's color when the mouse cursor moves over it.

This draws visitors' eyes to the button and indicates that they can follow this link. You can also use JavaScript to validate forms before visitors submit them. You can, for example, use JavaScript to ensure that a visitor completes an e-mail field before actually submitting the form. The JavaScript language offers features common to other programming languages. These features include variables, loops, conditional statements, various numeric and string operators, such as +, -, /, ++, --, user defined functions (similar to subroutines) and comments.

Client Side Features of JavaScript

The following are the client side features of JavaScript:

- Simple to use.
- Dynamic (responds to events).
- Object based.

New Features of JavaScript in Netscape Navigator 3.0

Some of the latest features of JavaScript are as follows:

- It can change GIF and JPEG images automatically, at specified time intervals, by clicking a button or icon or moving the mouse over an object.
- It can detect the presence of plug-ins on a Web page and tailor the user interface accordingly.
- It can communicate with plug-ins on the same page.
- Java applets can communicate with JavaScripts.
- The server-side of JavaScript requires LiveWire for Server Side Includes (SSI) or Internet Information Server/Personal Web Server (IIS/PWS) for Application Server Pages (ASPs).

Usage of JavaScript

The various uses of JavaScript are as follows:

- It moves action from the server to the client.
- It can locally validate form fields before submitting the form to the server.
- HTML documents can respond to local events.
- Web page developer can communicate information to/from applets and plug-ins.

- It supports unique personalized user profiles.
- It enables users to access databases.

Java is an Object Oriented Programming (OOP) language created by James Gosling of Sun Microsystems. The people at Netscape created JavaScript. It is also an OOP language. Many of the programming structures of Java and JavaScript are similar. However, JavaScript contains a much smaller and simpler set of commands than Java. It is easier for the average computer educated person to understand JavaScript.

What is OOP? Object Oriented Programming is a relatively new concept, whereas the sum of the parts of a program makes up the whole. Think of it this way, you are building a model car. You build the engine first. It can stand alone. It is an engine and everyone can see it is an engine. Next you build the body. It can also stand alone. Finally, you build the interior including the seats, steering wheel and other components. Each, by itself is an object. However, it is not a fully functioning car until all the pieces are put together. The sum of the objects (parts) makes up the whole.

Continuing with the model car example, when you built the engine, you did not use any of the parts that would later build the seats (a 350 four barrel engine with a seat belt sticking out of the piston). The point is that all the parts that made up the engine were of a certain *class* of parts. They all went together. Some is applicable with the body and then the interior. The point is that in these languages, you build objects out of *classes* of commands to create the whole.

The main difference is that Java can stand on its own while JavaScript must be placed inside an HTML document to function. Java is a much larger and more complicated language that creates 'Standalone' applications. A Java 'applet' is a fully contained program. JavaScript is text that is fed into a browser that can read it and then is enacted by the browser.

Another major difference is how the language is presented to the end user. Java must be compiled into what is known as a 'Machine Language' before it can be run on the Web. Basically what happens is after the programmer writes the Java program and checks it for errors, he or she hands the text over to another computer program that changes the text code into a smaller language. That smaller language is formatted so that it is seen by the computer as a set program with definite beginning and ending points. Nothing can be added to it and nothing can be subtracted without destroying the program.

JavaScript is text based. You write it to an HTML document and it is run through a browser. You can alter it after it runs and run it again and again. When Java is compiled, you can go back to the original text and alter it, but then you need to compile again.

Java applets run independent of the calling HTML document. Sure, they appear on the page but the HTML document did little more than call for the application and place it. If the programmer allows, parameters can be set many times by the HTML document. This includes the background color of the applet,

NOTES

NOTES

the type of text it displays, etc. The applet is executed through a download on the Web page. The HTML document calls for the application, it downloads to the user's cache and waits to run. JavaScript is wholly dependent on the browser for its execution:

What are the benefits of using one over the other? There are several advantages. If you can understand Java, it is amazingly versatile. Due to the size and structure of the language, it can be used to create anything from small Web page events to entire databases to full browsers.

JavaScript's main benefit is that it can be understood by a common computer literate person. It is much easier and more robust than Java. It allows for fast creation of Web page events. Many JavaScript commands are known as event handlers, i.e., they can be embedded right into existing HTML commands. JavaScript is a little more flexible than Java. It allows more freedom in the creation of objects. Java is very rigid and requires all items to be denoted and spelled out. JavaScript allows you to call on an item that already exists, like the status bar or the browser itself and play with just that part. JavaScript is geared to Web pages. Java is geared towards where it is needed most at the time.

Both will create Web page events and both can offer interaction between the user and the Web page. However, they are not created equally by any means. Thus, one can use whichever fits in one's requirement.

1.2.5 Generics of Java

Generics is a new addition to Java. Before the evolution of generics, a programmer had to design different programs to deal with different data types having the same logic. If one had to design a swapping program for an integer object, then it could not work for other data types. Generics has solved these overheads.

Generics is a powerful feature of Java. It was introduced by JDK 5. Using generics, a programmer can create classes, interfaces and methods that will work for various types of data in a type safe manner. One can define a single algorithm which is independent of data; then apply the same algorithm for various data types, without any changes.

The Need for Generics

Generally, type specific classes, interfaces and methods are created. However, using generics, it is possible to create classes, interfaces and methods which are type independent.

Previously, this was being done by using an object class, because it is the super class of other classes and an object reference can refer to any type of object. In order to get the actual data, one had to explicitly typecast it to the required type. Generics overcomes this overhead. In generics all typecasts are done implicitly. This makes the use of generics more secure. This is illustrated by the following example.

Example 1.1

```
// Definition of the generic class Gener
class Gener<S>
{
    S obj;
    Gener(S o)
    {
        obj =o;
    }
    void showClass ()
    {
        System.out.println("Type of S is "+
obj.getClass().getName());
    }
    void showData ()
    {
        System.out.println("Data is "+obj);
    }
}
public class GenericDemo
{
    public static void main (String[] args)
    {
        // Create a Gener reference for Integer
        Gener<Integer> ob1 = new Gener<Integer>(100);
        // prints the type of data held by it
        ob1.showClass();
        // prints the data held by it
        ob1.showData();
        // Create a Gener reference for Sting
        Gener<String> ob2 = new Gener<String>("SAMITA , LORY");
        // prints the type of data held by it
        ob2.showClass();
        // prints the data held by it
        ob2.showData();
    }
}
```

The output of the program:

```
Type of S is Java.lang.Integer
Data is 100
Type of S is Java.lang.String
Data is SAMITA, LORY
```

In the preceding example, S is the name of a type parameter. S holds the type parameter of which the Gener class object is created. S is written inside <>. Whenever one specifies the type of class object which is required to be created, it is specified inside <>. Everywhere in the class definition, S behaves like the type specified for that object.

```
Gener<Integer> ob1 = new Gener<Integer>(100);
```

In the above line, the object of Gener class, of Integer type, has been created. The type has been specified inside <> while calling the constructor. Here S holds an integer; so, for ob1, S behaves as an integer. Therefore, the argument has been passed according to the type.

```
Gener<String> ob2 = new Gener<String>("SAMITA , LORY");
```

NOTES

NOTES

Here, the object of `Gener` class of string type was created. `S` holds string; so, for `ob2`, `S` behaves as string. So the argument has been passed according to the type.

`Generics` works only on objects. This means that the type argument passed to type parameter must be a class type but it cannot be any primitive type. For example,

```
Gener<int> ob1 = new Gener<int>(100); // Error
```

The above example will result in an error because primitive type (`int`, `char`, etc.) cannot be used.

It must be understood that a reference of one specific type of generic type is different from another generic type. For example,

```
ob1 = ob2 ; // wrong
```

This is wrong because although both `ob1` and `ob2` are of type `Gener<S>`, yet they are references of different types because of their type parameters.

A Generic Class with Two Types of Parameters

A programmer can use more than one type of parameter in a generic type. If two or more type parameters are to be specified, these just have to be separated with commas. This can be seen in the example that follows:

Example 1.2

```
//Definition of the generic class Gener
class Gener<A, B>{
    A ob1;
    B ob2;
    Gener(A o1 ,B o2 )
    {
        ob1 =o1;
        ob2 =o2;
    }
    void showClass ()
    {
        System.out.println("Type of A is "+
ob1.getClass().getName());
        System.out.println("Type of B is "+
ob2.getClass().getName());
    }
    void showData ()
    {
        System.out.println("Data in ob1 "+ ob1);
        System.out.println("Data in ob2 "+ ob2);
    }
}
public class GenericDemo {
    public static void main(String[] args) {
        // Create a Gener reference for Integer and String
        Gener<Integer, String> obj1 = new
        Gener<Integer, String>(100,"SAMITA , LORY");
        // prints the type of data held by it
        obj1.showClass();
    }
}
```

```
        // prints the data held by it  
        obj1.showData();  
    }  
}
```

The output of the program:

```
Type of A is Java.lang.Integer  
Type of B is Java.lang.String  
Data in ob1 100  
Data in ob2 SAMITA , LORY
```

In the two type parameters T and V are separated by a comma. Therefore, if one wants to create a reference of Gener, he has to pass two type arguments.

Bounded Type Generic Class

This is a feature of generics in which we can restrict the type argument passed to the type parameter of generic class to a particular type. This can be seen in the example that follows.

Example 1.3

```
// Definition of the generic class Gener  
class Gener<A extends Integer >{  
    A ob1;  
    Gener (A o1 )  
    {  
        ob1 =o1;  
    }  
    void showClass ()  
    {  
        System.out.println("Type of A is "+  
ob1.getClass().getName());  
    }  
    void showData ()  
    {  
        System.out.println("Data in ob1 "+ ob1);  
    }  
}  
  
public class GenericDemo {  
    public static void main (String[] args) {  
        // Create a Gener reference for Integer  
        Gener<Integer> obj1 = new Gener<Integer>(100);  
        // prints the type of data held by it  
        obj1.showClass ();  
        // prints the data held by it  
        obj1.showData ();  
    }  
}
```

The output of the program:

```
Type of A is Java.lang.Integer  
Data in ob1 100
```

Though the preceding will work well but the following program will not work. This can be illustrated with the following example.

NOTES

NOTES

Example 1.4

```
// Definition of the generic class Gener
class Gener<A extends Integer >{
    A ob1;
    Gener (A o1 )
    {
        ob1 =o1;
    }
    void showClass ()
    {
        System.out.println("Type of A is "+
ob1.getClass().getName());
    }
    void showData ()
    {
        System.out.println("Data in ob1 "+ ob1);
    }
}
public class GenericDemo {
    public static void main (String[] args) {
        // Create a Gener reference for String
        Gener<String> obj2 = new Gener<String>("SAMITA , LORY");
        // prints the type of data held by it
        obj2.showClass ();
        // prints the data held by it
        obj2.showData ();
    }
}
```

The output of the program:

```
Exception in thread "main" Java.lang.Error: Unresolved compilation
problems:
Bound mismatch: The type String is not a valid substitute for the
bounded parameter <A extends Integer> of the type Gener<A>
Bound mismatch: The type String is not a valid substitute for the
bounded parameter <A extends Integer> of the type Gener<A>
at GenericDemo.main (GenericDemo.Java:27)
```

The preceding program will result in an error because the programmer has bounded the type of A to an integer. If one tries to give it any other type except the child classes of integer, then it will result in an error.

One can declare interfaces as bound for A. One can also declare one class and multiple interfaces as bound for A. For example,

```
class Gener<A extends Myclass and Myinterface>
```

Here, Myclass is a class and Myinterface is an interface. An operator has been used to connect them.

Wildcard Arguments

These are a special feature of generics. Suppose one wants to define a method inside the generic class which compares the value of different type of generic class objects and returns the result, irrespective of their types. Previously, it was not possible because a method defined inside the generic class could only act upon the data types which were the same as that of the object calling it. However, this

can be done using wildcard arguments. Students can understand the use of wildcard arguments from the next example.

Example 1.5

```
// Definition of the generic class Gener
class Gener<T>{
    T ob1;
    Gener(T o1 )
    {
        ob1 =o1;
    }
    // wildcard argument is used
    void Equals (Gener<?> o2)
    {
        if(ob1 == o2.ob1)
            System.out.println("TRUE");
        else
            System.out.println("False");
    }
}

public class GenericDemo {
    public static void main(String[] args) {
        // Create a Gener reference for Integer
        Gener<Integer> obj2 = new Gener<Integer>(100);
        // Create a Gener reference for Double
        Gener<Double> obj1 = new Gener<Double>(100.0);
        // Create a Gener reference for String
        Gener<String> obj3 = new Gener<String>("100");
        obj2.Equals(obj1);
        obj2.Equals(obj3);
        obj1.Equals(obj3);
    }
}
```

NOTES

The output of the program:

```
False
False
False
```

In the preceding program, a wildcard argument has been used. <?> represents the wildcard argument, i.e., it will work irrespective of types. It can be seen that the equals method, checks the values of different objects, irrespective of their types and prints the result.

Wildcards can also be bounded. In the above example, suppose one wants the equals method to only execute the numbers, otherwise results in an error. For this, a little change would be required in the method definition.

```
void Equals (Gener<? extends Number> o2)
{
    if(ob1 == o2.ob1)
        System.out.println("TRUE");
    else
        System.out.println("False");
}
```

Now, if the following statements are executed:

```
obj2.Equals(obj3);
```

```
obj1.Equals(obj3);
```

It will result in an error because one can create an object reference of Gener class for string type, but cannot use the equals method for the string type.

NOTES

Creating a Generic Method and Generic Constructor

How will you define a generic method and generic class? A generic method can be created inside a non-generic class which acts on multiple types of data independently. One can also define a generic constructor inside a non-generic class which can act on multiple types independently. An example to illustrate this is given below.

Example 1.6

```
public class GenericDemo {
    double db;
    // Generic constructor
    <T extends Number> GenericDemo (T o1)
    {
        db= o1.doubleValue ();
    }
    // Generic Method
    static < V > void Display (V o2)
    {
        System.out.println(o2);
    }
    void show ()
    {
        System.out.println(db);
    }
    public static void main (String [] args) {
        GenericDemo g1 = new GenericDemo (100);
        g1.show ();
        GenericDemo g2 = new GenericDemo (1025.54);
        g2.show ();
        GenericDemo g3 = new GenericDemo (103.9F);
        g3.show ();
        Display (100);
        Display (125.56);
        Display ("SAMITA , LORY");
    }
}
```

The oputput of the program :

```
100.0
1025.54
103.9000015258789
100
125.56
SAMITA , LORY
```

From the preceding example, one can observe the output. The generic constructor takes the numbers of different types of argument as type parameters and stores its double value in the variable db of each reference, which is similar to the Display method, taking different type of arguments as type parameters and displaying the values.

Erasure or Raw Types

A raw type is a parameterized type stripped of its parameters. The official term given to the stripping of parameters is type erasure. Raw types are necessary to support the legacy code that uses non-generic versions of classes. It is because of type erasure, that it is possible to assign a generic class reference to a reference of its non-generic (legacy) version. Therefore, the following code compiles without an error:

Example 1.7

```
Gener ob1 ;
Gener<Integer> ob2 ;
ob1=ob2; // valid
ob2=ob1 ; // will cause a unchecked warning
```

It must be remembered that during compilation, all types of parameters are erased and only the raw types actually exist.

Example 1.8

```
// Definition of the generic class Gener
class Gener<T>{
    T obj;
    Gener(T o)
    {
        obj =o;
    }
    void showClass()
    {
        System.out.println("Type of T is "+
obj.getClass().getName());
    }
    void showData()
    {
        System.out.println("Data is "+obj);
    }
}
public class GenericDemo {
    public static void main(String[] args) {
        // Create a Gener reference for Integer
        Gener ob1 = new Gener(100);
        System.out.println("Type of ob1 is "+
            ob1.getClass().getName());
        // prints the type of data hold by it
        ob1.showClass();
        // prints the data hold by it
        ob1.showData();
        // Create a Gener reference for Sting
        Gener<String> ob2 = new Gener<String>("SAMITA , LORY");
        System.out.println("Type of ob2 is "+
            ob2.getClass().getName());
        // prints the type of data held by it
        ob2.showClass();
        // prints the data held by it
        ob2.showData();
    }
}
```

NOTES

NOTES

The output of the program:

```
Type of ob1 is Gener
Type of T is Java.lang.Integer
Data is 100
Type of ob2 is Gener
Type of T is Java.lang.String
Data is SAMITA , LORY
```

From the preceding example, it can be seen that both `ob1` and `ob2` are not of integer class or string class. They are of `Gener` class. However, according to the parameters, the variables inside `ob1` and `ob2` are typecasted accordingly. When a programmer writes the code:

```
Gener<Integer> ob1 = new Gener<Integer>(100);
int i = ob1.obj;
```

It is compiled as if it is written like:

```
Gener ob1 = new Gener(100);
int i = (Integer)ob1.obj;
```

Restrictions While using Generics

These various are as follows:

Type Parameters cannot be Instantiated

```
class Gener<T>{
    T obj;
    Gener(T o)
    {
        obj =newT(); // error
    }
}
```

The preceding code will result in an error because `T` does not exist at runtime. So the compiler will not know the type of object which is to be created.

Restrictions on static members

Below are some facts which should be taken care of while using the keyword `static`.

```
class Gener<T>{
    // error , cannot make a static reference to a non-static
type
    static T obj ;
    // error, non-static method can use T
    Static T show()
    {
    // error, non-static method can acces T type object
        System.out.println(obj);
    }
}
```

1.3 OVERVIEW OF JAVA

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let application developers Write Once,

RunAnywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities, (such as reflection and runtime code modification) that are typically not available in traditional compiled languages. As of 2019, Java was one of the most popular programming languages in use according to GitHub, particularly for client-server web applications, with a reported 9 million developers.

NOTES

1.3.1 What is Object?

Objects are small, self-contained and modular units with a well-defined boundary. An object consists of a state and behaviour. The state of an object is one of the possible conditions that an object can exist in and is represented by its characteristics or attributes or data. The behaviour of an object determines how an object acts or behaves and is represented by the operations that it can perform. In OOP, the attributes of an object are represented by the variables and the operations are represented by the functions.

For example, an object Biscuit may consist of data product code P001, product name Britania Biscuits, price 20 and quantity in hand 50. These data values specify the attributes or features of the object. Similarly, consider another object Maggi with product code P002, product name Maggi Noodles, price 10, and quantity in hand 20. In addition, the data in the object can be used by the functions such as `check_qty()` and `display_product()`. These functions specify the actions that can be performed on data. Figure 1.3 shows how class and its objects are represented.

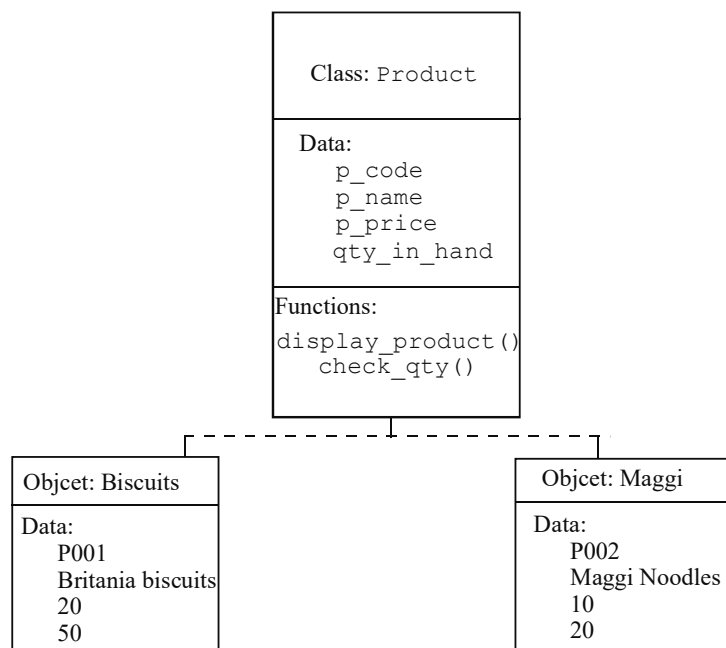


Fig. 1.3 Class and its Objects

NOTES

Objects are what actually run in the computer and are the basic run-time entities in object-oriented systems. They are the building blocks of object-oriented programming. Although, two or more objects can have same attributes, still they are separate and independent objects with their own identity. In other words, all the objects in a system take a separate space in the memory, independent of each other. The main objective of breaking down complex software projects into objects is that changes made to one part of a software should not adversely affect the other parts.

1.3.2 Features of Object Oriented Programming

You need to define an object, its variables and members to represent any real-world object in a Java program for the implementation of this object in a program. This object can be a person, a place or a record of a database. A class is a collection of data members and methods that are used to perform desired calculations and operations. Objects are the variables of type class. In object-Oriented Programming (OOP), first the classes are defined then the objects are created, and finally the objects communicate with one another to accomplish the desired programming tasks. The object-oriented approach involves grouping of data and functions into modular entities known as objects. You can group more than one object having similar attributes in a program using the class concept. Classes are user-defined data types that contain objects of similar types. You can create any number of objects of a class. Some important features of OOP as applied on objects are explained in the following sections.

Encapsulation

The technique of wrapping of data and methods into a single entity is termed as encapsulation. The data is accessed only by the methods which are encapsulated in the class. Encapsulation provides an interface between the data objects and the Java program.

Inheritance

Inheritance provides a mechanism to acquire properties of one object of a class from the object of another class. OOP uses the feature of inheritance for sharing the attributes and functions among various classes.

Inheritance provides re-usability of codes in the programs, as it allows you to add additional features to an existing class without changing the class. The derived class inherits features, such as variables and methods from a super class. This is the class from which a derived class inherits the variables and methods. A class that inherits the instance variables and methods from another class is termed as the sub-class. Instance variables are created when the objects are instantiated and, therefore, are associated with the objects. Instance variables take different values for different objects. In Java programming, the feature of inheritance is divided into the following categories:

- Implementation Inheritance
- Interface Inheritance

For example, vehicles can be classified as motorized vehicles and non-motorized vehicles. Motorized vehicles include scooter and car and non-motorized vehicles include rickshaw and cycle. Scooter and car inherit the properties of motorized vehicles and rickshaw and cycle inherit the properties of non-motorized vehicles. This example corresponds the feature of inheritance. Figure 1.4 shows the concept of inheritance.

NOTES

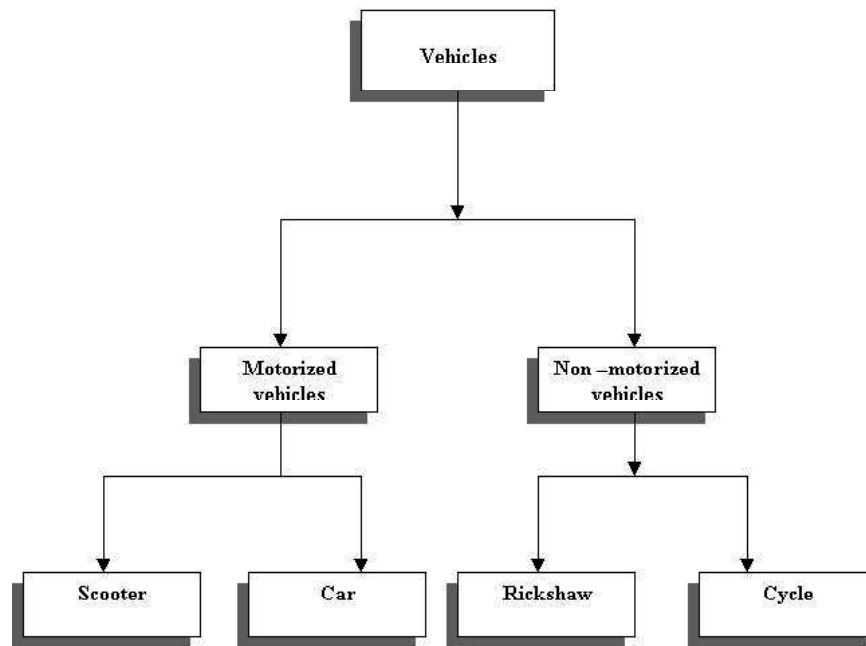


Fig. 1.4 Implementing Inheritance

Polymorphism

Polymorphism allows operators and functions to be used in different forms depending upon the parameters and operands. Parameters are the variables that are used to pass values in the function calls and the operands are the variables that are used with operators, such as addition operator, subtraction operator, etc. Polymorphism enables various entities, such as objects, variables and methods to have more than one form. For example, while multiplying an integer value with a floating point, the polymorphism technique allows the Java compiler to convert the variable of integer type into floating-point value, otherwise an error will occur due to variable mismatch.

1.3.3 Java: Simple Program and Compiling

You will learn about Java with a simple program that prints a string on the screen.

Example 1.9: A simple Java program

```
class Sample
{
    public static void main (String args[])
    {
        System.out.println("Welcome to Java Programming");
        System.out.println("Its easy and simple.");
    }
}
```

NOTES

Though this program is the simplest one, it includes the basic features that every Java program has. Now you will look at the features of Java.

Class Definition

The first statement `class Sample` declares a class where `class` is a keyword and `Sample` is the identifier that indicates the name of the class. The opening and closing curly braces `{ }` enclose the definition of a class.

The Main Statement

The statement `public static void main (String args[])` indicates the main method. This method is the point where the execution of the Java program begins. Since, it is the startup point for any Java program, it is the most essential part of any Java program.

This statement has certain keywords, namely, `public`, `static` and `void`. The descriptions of these keywords are as follows:

- **public:** It is the access specifier which specifies that the main method is accessible to all other classes.
- **static:** The main method is declared static which specifies that this method belongs to the entire class. This interpreter use this method before the creation of objects.
- **void:** The `void` keyword specifies that the main method does not return a value. The pair of parentheses contain the declaration of the parameters of the methods. In the given statement, the `String args[]` declares a parameter `args`, that contains an array of objects of the class type `String`.

The Output Statement

The statements `System.out.println("Welcome to Java Programming.");` and `System.out.println("Its easy and simple.");` are used to display information on the standard output device, that is the `println()` is a method of the `out` object, which is the static member of class `System`. These statements will display the following strings on the monitor.

```
Welcome to Java Programming.  
Its easy and simple.
```

The statements will be printed in separate lines as the method `println()` appends a newline character at the end of the string. However, if we use the `print()` method instead of the `println()`, the newline character is not appended at the end of the string. A point to be noted is that like C++, every statement in Java must end with a semicolon.

Running Java Applications

A program can be created using any text editor. Nowadays, there are several text editors available for writing programs like Notepad, Jcreator, etc. After creating the file, save the file with the name `<filename>.Java`. The name of the file must be the same as that of the class name containing the `main()` method.

For example, once the file is created, it can be compiled to generate the bytecode using Java compiler `javac` as given here.

```
javac Sample.java
```

If the source code is error-free then the compiler creates a file containing bytecode and the file will be named as `<filename>.class`. In the given example, the name of file will be `Sample.class`.

Even after compilation when the source code is converted into its equivalent bytecode, it cannot be executed. To execute this bytecode, it needs to be converted to machine code using the interpreter. The command for converting bytecode to machine code and run it is as follows:

```
java Sample
```

After giving this command, the interpreter searches for the `main()` method in the source code and starts executing the instructions written in this method and displays the corresponding output.

NOTES

1.4 DATA TYPES IN JAVA

A data type determines the type of operations that can be performed on the data. Java provides various data types and each is represented differently within a computer's memory. The type of data selected by a programmer depends on a particular application. Various data types provided by Java are categorized into primitive and non-primitive data types. (Refer Figure 1.5).

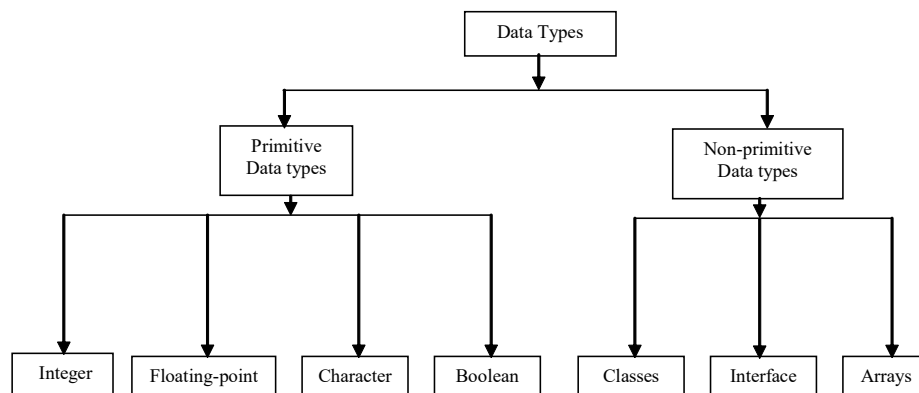


Fig. 1.5 Data Types

1. Primitive Data Types

Primitive data types also known as 'Built-in' data types. These are fundamental data types provided by a programming language. In Java, primitive data types include integer, floating-point, character and Boolean.

(i) Integer Type

The integer data type is used to store integers like: 4, 42, 5233, -32, -745. Java supports four types of integers, namely, `byte`, `short`, `int` and `long`. The default value of these integer types is 0. There is no concept of unsigned integer in Java. Various integer data types with their sizes and ranges are listed in Table 1.2.

NOTES

Table 1.2 Size and Range of Integer Types

Type	Size(bytes)	Range
byte	One	-128 to 127
short	Two	-32,768 to 32,767
int	Four	-2,147,483,648 to 2,147,483,647
long	Eight	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Note: It is recommended to use smaller data type wherever possible. This is because larger the data type we choose, more time the program will take for execution.

(ii) Floating-Point Type

A floating-point data type is used to store real numbers, such as: 3.28, 64.755765, -8.01, -24.53. Java supports two floating-point data types namely, float and double.

- **float:** The float type represents a single precision number. Single precision occupies lesser space than double precision but becomes inaccurate when the values are large. For example, it can be used to represent the value of marks of the students. The default value of float data type is 0.0f.
- **double:** The double type specifies a double-precision number. It is the best choice when numbers of large values are to be stored. For example, it can be used in case of mathematical functions like sin(), cos(), sqrt(). The default value of double data type is 0.0d.

The various floating-point data types with their size and range are listed in Table 1.3.

Table 1.3 Size and Range of Floating-Point Types

Type	Size(bytes)	Range
float	Four	3.4e-038 to 3.4e+038
double	Eight	1.7e-308 to 1.7e+308

(iii) Character Type

The 'Character' data type is used to store single characters enclosed in single quotes. It is represented by using the keyword, `char`. It occupies 16 bits of memory. The range of the character data type is 0 to 65,536. The default value of char data type is null character.

(iv) Boolean Type

The Boolean data type can hold only Boolean values, that is, either true or false. The keyword `Boolean` is used to denote the Boolean data type. The default value of Boolean data type is false.

2. Non-Primitive Data types

Non-primitive data types (user-defined data types) also known as reference types are derived from primitive data types. In Java, these include classes, interface and arrays.

1.4.1 Literals

Integer constants are whole numbers that must have at least one digit and should not have any decimal point. The three types of integer literals that Java provides are:

- (i) Decimal contains a set of digits, from 0 to 9. You cannot insert commas, spaces, and non-digit characters between digits. For example, 1234, 78, +98 and -67 are decimal integer literals.
- (ii) Octal contains a combination of digits, from 0 to 7, that starts with 0. For example, 045, 0 and 0524.
- (iii) Hexadecimal contains a sequence of digits starting with 0x or 0X. The hexadecimal integer includes alphabets from A to F, where letters A to F refer to numbers 10 to 15. For example, 0X3, 0X3F and 0xdef.

NOTES

1.4.2 Characters

Character data type is used to store a single Unicode character. Unicode character sets are 16-bits values. Therefore, the space required to store a single character is 16 bits. A character variable can be declared as:

```
char ch;  
char ch1, ch2;
```

1.4.3 Variables Declaration

A variable is an identifier that represents a memory location that is used to store data values. Data stored at a particular location can be accessed using the variable name. The value of the variable can be changed anytime during execution of the program. The variable name that is chosen must be meaningful so as to understand what it represents in the program.

Declaring Variables

Variables must be declared in a program before they are used. The declaration of a variable informs the compiler about the specific data type with which the variable is associated and the compiler allocates sufficient memory to it.

The syntax for declaring a variable is:

```
data_type variable_name;
```

For example, a variable 'a' of type `int` can be declared using the following statement:

```
int a;
```

At the time of declaration of variables, more than one variable of the same data type can be declared in a single statement. This is displayed in the following statement:

```
int x, y, z;
```

Initializing Variables

The declaration of variables allocates memory for variables but it does not store any data at the time of declaration. To store data in the variables, they need to be initialized. For example, consider these statements.

```
int i;  
i=10;
```

Here, a variable `i` of the integer type is declared and the value 10 is assigned to it. We can combine both the statements into a single statement as follows:

```
int i=10;
```

NOTES

Besides initializing the variable with constant values, it can also be initialized at run-time by using expressions. Initialization of variables at run-time is known as dynamic initialization.

Example 1.10: A program to demonstrate initialization of the variable `public` class using `dynamic_initialization` is as follows :

```
{
public static void main(String[] args)
{
int x=40,y=40,z=10; //initialization with constant values
int result=(x*y)+z; //dynamic initialization
System.out.println("The value of z is:"+result);
}
}
```

The output of the program is:

```
The value of z is:1610
```

Receiving Input through Keyboard

Variables can also be given values interactively through a keyboard by using the `readLine()` method. This can be understood with the help of Example 1.11.

Example 1.11: A program to demonstrate reading data from the keyboard is as follows:

```
//importing package for using DataInputStream class
import Java.io.*;
public class ReadingData
{
public static void main (String[] args)
{
DataInputStream in=new DataInputStream(System.in);
int num1=0;
float num2=0;
try
{
System.out.println("Enter integer value");
num1=Integer.parseInt(in.readLine());
System.out.println("Enter float value");
num2=Float.valueOf(in.readLine()).floatValue();
}
catch (Exception e)
{
}
System.out.println("The integer value is "+num1);
System.out.println("The float value is "+num2);
}
}
```

The output of the program is:

```
Enter integer value
4
Enter float value
6.7
The integer value is 4
The float value is 6.7
```

The method `readLine ()` of class `DataInputStream` is used to read a string from the keyboard which is then converted to the corresponding data type,

int and float. To handle the error which may occur while reading data from the keyboard, try and catch statements have been provided.

1.4.4 Symbolic Constants

The format for symbolic constant is as follows:

```
#define name constant
```

For example, we can define:

```
#define INITIAL 1
```

It defines INITIAL as 1.

The INITIAL type of definition is called symbolic constants. They are not variables and hence, they are not defined as part of the declarations of variables. They are specified on top of the program before the main() function. The symbolic constants are to be written in capital or upper case letters. Wherever the symbolic constant names appear in the program, the compiler will replace them with the corresponding replacement constants defined in the #define statement. In this case, 1 will be substituted wherever INITIAL appears in the program. Note that there is no semicolon at the end of the #define statement.

1.4.5 Type Casting

The process of converting one data type to another is called typecasting. Typecasting becomes inevitable on many occasions. The return type of function function2() may be a character. The return value of function2() has to go as an argument to another function function1() which accepts only the integer argument. The solution here is to typecast the return value of function2() to integer and then use it as argument for function function1().

```
Function1 ((int) function2())
```

Typecasting is performed by placing the desired type in parentheses to the left of the value to be converted.

For example:

```
(char) value
```

The storage size of the type you are attempting to cast is very important. Attempting to cast from a larger size to a smaller size may result in loss of data. Attempting to cast from a smaller size to a larger size is always safe.

The table that follows lists the casts that are guaranteed to result in no loss of information.

Typecasting that results in no loss of information.

<i>From type</i>	<i>To type</i>
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

NOTES

NOTES

1.4.6 Arrays

Java uses variables of different primitive data types to store data. However, these variables are incapable of holding more than one value at a time. For example, a single variable cannot be used for storing the marks of all students in a class. For such purposes, Java provides a different kind of data type known as arrays.

Arrays are defined as a sequence of the same type of data elements of a fixed size. These data elements can be primitive or non-primitive data types. The elements of an array are stored in contiguous memory locations and each individual element can be accessed using one or more indices or subscripts. A subscript or an index is a positive integer value, which indicates the position of an element in an array. Arrays are used when a programmer wants to store multiple data items of the same type into a single list and also wants to access and manipulate individual elements of the list. Arrays can be either single-dimensional or multi-dimensional, depending upon the number of subscripts used.

Single-Dimensional Arrays

A single-dimensional array is the simplest form of an array that requires only one subscript to access an array element. Like an ordinary variable, an array must be declared before it is used in the program.

The syntax for declaring a single-dimensional array is as follows:

```
data_type array_name[];  
or  
data_type[] array_name;
```

where,

`data_type` is any data type.

`array_name` is the name of the array.

For example, an array `marks []` of type `int` can be declared using either of the two statements.

```
int marks[];
```

or

```
int[] marks;
```

After an array is declared, we need to create it by allocating space to it in the memory. Arrays are created using `new` operator.

The syntax for creating an array is as follows:

```
array_name=new data_type[size];
```

where,

`size` is the size of the array.

For example, an array `marks []` of type `int` and `size` five can be created using the following statement:

```
marks=new int[5];
```

The above two steps of declaration and creation of an array can be combined into a single statement as shown below.

```
data_type array_name=new data_type[size];
```

Similarly, the statement to declare and create an array `marks []` of type `int` and size five is:

```
int marks[]=new int[5];
```

Note: All the elements created using `new` operator in the array will be automatically initialized to zero.

Initialization of a Single-Dimensional Array

Once an array is declared and memory is allocated to it, the next step is to initialize each array element with a valid and appropriate value. An array can be initialized at the time of its declaration.

The syntax for initializing an array at the time of its declaration is as follows:

```
data_type array_name[]={value_1,value_2,.....,value_n};
```

Values are assigned to array elements in the order in which they are listed. That is, `value_1`, `value_2` and `value_n` are assigned to the first, second and `n`th element of the array, respectively. If an array is declared and initialized simultaneously, then specifying its size is optional. For example, the statement `int marks[]={51, 62, 43, 74, 55}` is also valid. The size of an array, `marks`, can be obtained by using the `marks.length()` method.

Note: If you try to store or access values outside the range of an array (index with negative value or value greater than the length of the array), a run-time error is generated.

Accessing Single-Dimensional Array Elements

Once an array is declared and initialized, the values stored in the array can be accessed any time. Each individual array element can be accessed using the name of the array and the subscript value. Every element in an array is associated with a unique subscript value, starting from 0 to `size-1` (where, `size` refers to the maximum number of elements that can be stored in the array).

The syntax for accessing the values stored in a single-dimensional array is:

```
array_name[subscript]
```

For example, the elements of the array `marks` can be referred to as `marks[0]`, `marks[1]`, `marks[2]`, `marks[3]` and `marks[4]`, respectively. Note that index of an array starts with 0.

Note: The memory location, where the first element of an array is stored, is known as the base address, which is generally referred to by the name of the array.

Single-dimensional arrays are always allocated contiguous blocks of memory. This implies that every element in an array is always stored in sequential manner next to each other. The memory representation of the array `marks` is shown in Figure 1.6. As each element is of the type `int` (that is, 4 bytes long), the array `marks` occupies twenty contiguous bytes in the memory and these bytes are reserved in the memory at the time of compilation.

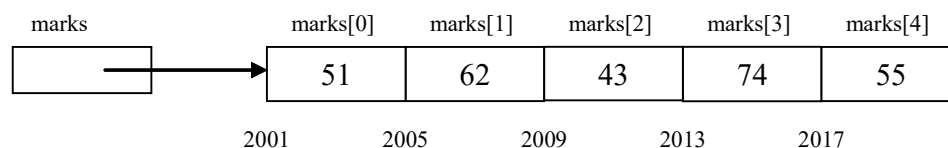


Fig. 1.6 Memory Representation of an Array `marks`

NOTES

NOTES

Manipulation of Single-Dimensional Array Elements

An array can be manipulated with the help of various operations. These operations include finding the sum, average, maximum or minimum, sorting and searching of the array elements, and so on.

Example 1.12: A program to sort the array elements is as follows:

```
class SortingArray
{
public static void main(String args[])
{
int a[]={67, 34, 12, 98, 26}; //array initialization
//at the time of declaration
int n=a.length; //returns the length of the array

System.out.print("The list of numbers:");
for(int i=0;i<n;i++)
{
System.out.print(" "+a[i]);
}

//sorting elements of an array
for(int i=0;i<n;i++)
{
for(int j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
int temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
System.out.print("\n");
System.out.print("The sorted list of given numbers:");
for(int i=0;i<n;i++) //displaying sorted array
{
System.out.print(" "+a[i]);
}
}
}
```

The output of the program is:

```
The list of numbers: 67 34 12 98 26
The sorted list of given numbers: 12 26 34 67 98
```

Multi-Dimensional Arrays

Multi-dimensional arrays can be described as ‘An array of arrays’, that is, each element of the array is itself an array. A multi-dimensional array of dimension n is a collection of items that are accessed with the help of n subscript values.

Two-Dimensional Array

A two-dimensional array is the simplest form of a multi-dimensional array that requires two subscript values to access an array element. These arrays are useful when the data being processed is to be arranged in the form of rows and columns (matrix form).

The syntax for declaring a two-dimensional array is as follows:

```
data_type array_name[] [];
```

or

```
data_type[][] array_name;
```

The syntax for creating a two dimensional array is as follows:

```
array_name[][]=new data_type[row_size][column_size];
```

The above two steps of declaration and creation can be combined into one, using a single statement as shown below:

```
data_type array_name[][]=new data_type[row_size][column_size];
```

For example, an array `a [] []` of type `int` having three rows and two columns can be declared and created using the following statement:

```
int a[][]=new int[3][2];
```

Here, 3 is the row size and 2 is the column size.

• Initialization of a Two-Dimensional Array

Like a single-dimensional array, a two-dimensional array can also be declared and initialized at the same time. To understand how to initialize a two-dimensional array, consider the following statement:

```
int a[3][2]={ {101,51},  
             {102,67},  
             {103,76}  };
```

In this statement, an array `a [] []` of type `int`, having three rows and two columns is declared and initialized. This type of initialization is generally used to increase the readability.

Now, consider another statement:

```
int b[][]={ {2,3,4}, {1,1,1} };
```

In this statement, an array `b [] []` of type `int`, having two rows and three columns is initialized.

• Accessing Two-Dimensional Array Elements

Once a two-dimensional array is declared and initialized, the value stored in the array elements can be accessed using two subscripts. The syntax for accessing a two-dimensional array element is:

```
array_name[row][column]
```

The first subscript value (`row`) specifies the row number and the second subscript value (`column`) specifies the column number. Both the subscript values specify the position of the array element within the array. For example, the elements of array `a` (declared earlier) are referred to as `a [0] [0]`, `a [0] [1]`, `a [1] [0]`, `a [1] [1]`, `a [2] [0]` and `a [2] [1]`, respectively.

Generally, two-dimensional arrays are represented with the help of a matrix. However, in actual implementation, two-dimensional arrays are always allocated contiguous blocks of memory. Figure 1.7 shows a matrix and memory representation of two-dimensional array `a`.

NOTES

NOTES

101	51
105	67
103	76

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]
101	51	105	67	103	76
2001	2005	2009	2013	2017	2021
first row		second row		third row	

Matrix Representation

Memory Representation

Fig. 1.7 Matrix and Memory Representation of Array a [][]

• Manipulation of Two-Dimensional Array Elements

A two-dimensional array can be manipulated in many ways. Some of the common operations that can be performed on a two-dimensional array include finding the sum of row elements, column elements and diagonal elements, finding the maximum and minimum values, etc.

Example 1.13: A program to calculate the sum of two matrices is as follows:

```
class MatricesSummation
{
public static void main(String args[])
{
int a[][]={{3,4,5},{3,2,7}}; //initializing matrix a
int b[][]={{2,4,7},{1,2,2}}; //initializing matrix b

int l=a.length;
System.out.println("First matrix is:" + " ");
for(int i=0;i<l;i++) //displaying first matrix
{
for (int j=0;j<3; j++)
{
System.out.print(" " +a[i][j]);
}
System.out.println();
}

int m=b.length;
System.out.println("Second matrix is:" + " ");
for (int i=0;i<m;i++) //displaying second matrix
{
for(int j=0;j<3;j++)
{
System.out.print(" " + b[i][j]);
}
}
System.out.println();
}
System.out.println("Summation of the two matrices is: ");
//displaying sum of two matrices
for(int i=0;i<m;i++)
{
for(int j=0;j<=m;j++)
{
System.out.print(" "+(a[i][j]+b[i][j]));
}
}
```

```

System.out.println();
}
}
}

```

The output of the program is:

```

First matrix is:
3 4 5
3 2 7
Second matrix is:
2 4 7
1 2 2
Summation of the two matrices is:
5 8 12
4 4 9

```

Variable Size Arrays

As already mentioned, multi-dimensional arrays are arrays of arrays. In such multi-dimensional arrays the size of each array can vary. For example, consider the following statements:

```

data_type array_name[][]=new data_type[size][];
array_name[0]=new data_type[size_1];
.
.
.
array_name[n-1]=new data_type[size_n];

```

where,

`size` is the number of rows in a two-dimensional array.

`size_1, ..., size_n` represents the number of columns in each row of a two-dimensional array.

Example 1.14: A program to demonstrate a variable size array is as follows:

```

class VariableArray
{
    public static void main(String args[])
    {
        int a[][]=new int[4][];
        a[0]=new int[2];
        a[1]=new int[4];
        a[2]=new int[3];
        a[3]=new int[5];
        int i,j;
        System.out.println("The variable sized array is: ");
        for(i=0;i<4;i++)
        {
            for (j=0; j<a[i].length; j++)
            {
                a[i][j]=j;
                System.out.print(" "+a[i][j]);
            }
            System.out.println();
        }
    }
}

```

NOTES

NOTES

The output of the program is:

The variable sized array is:

```
0 1
0 1 2 3
0 1 2
0 1 2 3 4
```

1.4.7 Array Declaration Syntax

We can declare an array by specifying its data type, name and the number of elements the array holds between square brackets immediately following the array name. The following syntax is required to declare an array:

```
data_type array_name[size];
```

For example, to declare an integer array which contains 100 elements following statement is required:

```
int a[100];
```

There are some rules on array declaration. The data type can be any valid C data types including structure and union. The array name has to follow the rule of variable and the size of array has to be a positive constant integer. A value stored into an element in the array simply by specifying the array element on the left hand side of the equals sign. The declaration `int values[10];` would reserve enough space for an array called values that could hold up to 10 integer values. Initializing arrays is like a variable in which an array can be initialized. To initialize an array, you provide initializing values which are enclosed within curly braces in the declaration and placed following an equals sign after the array name. The following statement is required to initialize an integer array:

```
int list[5] = {2,1,3,7,8};
```

1.4.8 Vectors

The `Vector` class contained in the `Java.util` package defines methods to store objects into a single unit. It can be used to implement a dynamic array of vectors which can accommodate any number and type of objects. Consider the following statements:

```
Vector v1=new Vector(); //creating vector without specifying its //
initial capacity
Vector v2=new Vector(n); //creating vector having initial capacity
//'n'
```

Here, the first statement creates a vector `v1` having an initial capacity of 10. That is, when you create a vector without specifying its initial capacity, it is automatically set to 10. Similarly, the second statement creates a vector `v2` with an initial capacity of `n`. Although, we have created the vector `v2` by specifying its initial capacity but this specification does not pose any limitation on the size of the vector. Since, the size of the vector can be increased or decreased by adding objects to it or removing them from it.

The `Vector` class provides a variety of methods which can be used to perform different operations on vectors some of which are listed in Table 1.4.

Table 1.4 Vector Methods and Their Description

Methods	Description
<code>vect1.addElement(object)</code>	adds the specified <code>object</code> at the end of the vector list <code>vect1</code>
<code>vect1.size()</code>	returns the number of objects currently present in the vector <code>vect1</code>
<code>vect1.capacity()</code>	returns the maximum capacity of the vector <code>vect1</code>
<code>vect1.removeElement(object)</code>	removes the specified object from the vector <code>vect1</code>
<code>vect1.elementAt(n)</code>	returns the name of the n^{th} object of the vector <code>vect1</code>
<code>vect1.removeElementAt(n)</code>	removes the item at the n^{th} position of the vector <code>vect1</code>
<code>vect1.removeAllElements()</code>	removes all the elements in the vector <code>vect1</code>
<code>vect1.firstElement()</code>	returns the first element of the vector <code>vect1</code>
<code>vect1.lastElement()</code>	returns the last element of the vector <code>vect1</code>
<code>vect1.trimToSize()</code>	sets the capacity of the vector <code>vect1</code> to the number of objects it is currently holding

NOTES

Example 1.15: A program to demonstrate the use of some of the methods of the `Vector` class is as follows:

```
import java.util.*; //importing package for using vectors
public class VectorMethods
{
    public static void main(String[] args)
    {
        Vector vect1 = new Vector();
        String str1 = "Hello!!!";
        String str2 = "How are you?";
        String str3 = "All the best!";
        //adding string object to the vector
        vect1.addElement(str1);
        vect1.addElement(str2);
        vect1.addElement(str3);

        System.out.println("The initial capacity of the vector is: "
            +vect1.capacity());
        System.out.println("The elements of vector: " +vect1);
        System.out.println("The size of vector is: " +vect1.size());
        System.out.println("The first element of vector is: "
            +vect1.firstElement());
        System.out.println("The last element of vector is: "
            +vect1.lastElement());
    }
}
```

The output of the program is:

```
The initial capacity of the vector is: 10
The elements of vector: [Hello!!!, How are you?, All the best!]
The size of vector is: 3
The first element of vector is: Hello!!!
The last element of vector is: All the best!
```

Note: A `String` class is used to create string objects.

NOTES

Check Your Progress

1. When and where was Java developed and who headed the development team?
2. What is JDK?
3. Define on the term multithreaded in Java.
4. What is JavaScript?
5. What are the uses of generics?
6. What are JavaScript dialog box and its types?
7. State about the objectives.
8. What is the default value of a Boolean data type?
9. Define a variable.
10. How many subscripts does a single-dimensional array require to access an array element?

1.5 OPERATORS IN JAVA

As stated earlier, operators are symbols which perform operations on various data items known as operands. For example, in $a + b$, a and b are operands and $+$ is an operator. Note that to perform an operation, operators and operands are combined together forming an expression. For example, to perform an addition operation on operands a and b , the addition ($+$) operator is combined with the operands a and b forming an expression.

Depending on the function performed, Java operators can be classified into various categories. These include arithmetic operators, increment and decrement operators, relational operators, logical operators, conditional operator, assignment operators, bitwise operators and special operators.

1.5.1 Arithmetic Operators

Arithmetic operators perform the basic arithmetic operations on operands. They can work on any built-in data type of Java except on Boolean type. Java provides various arithmetic operators that are, $+$ (addition or unary plus), $-$ (subtraction or unary minus), $*$ (multiplication), $/$ (division) and $\%$ (modulus). For example, some of the expressions which involve arithmetic operators are $x + y$, $x - y$, $x * y$, x / y and $x \% y$. When the unary minus operator is used with a single operand, the operand is multiplied by -1 .

Expressions formed by using arithmetic operators can be of the following types:

- **Integer Expression:** An arithmetic expression where both the operands are integers is called an integer expression.
- **Real Expression:** An arithmetic expression where both the operands are real is called a real expression.

- **Mixed Mode Expression:** An expression is known as mixed mode if one operand is real and the other is an integer. In this case, the integer operand is converted to real and the result is also of type real.

Note: Unlike C and C++, the modulus operator can also be applied to the floating-point data type in Java.

NOTES

1.5.2 Basic Assignment Operators

An assignment operator assigns the value of an expression to a variable. Assignment operators are of two types, namely simple and compound assignment operators.

Simple Assignment Operator

The simple assignment operator assigns a value on its right side to the variable on its left. Note that the left hand side of an assignment expression should be a variable. It cannot be a constant or an expression. However, the right side of an assignment expression can be a variable, or a constant or an expression.

To understand the simple assignment operator, consider Example 1.16.

Example 1.16: Evaluate the following statement:

```
x=8;
```

In this example, the value 8 is assigned to the variable *x*.

With the help of the assignment operator, a common value can be assigned to several variables. This is accomplished by using multiple assignments in a single statement. For example, in the statement *x=y=z=5*, the value 5 is assigned to the three variables *x*, *y* and *z*.

Compound Assignment Operators

Java provides compound assignment operators (also known as Java shorthands) which are in the following format:

```
v op=exp;
```

Here, *v* is a variable, *op* is the binary operator and *exp* is an expression. This form is equivalent to the following statement:

```
v=v op (exp) ;
```

where *v* is needed to be accessed only once. For example, the expression *x=x+6* can be written as *x+=6*. In this expression, *x* is incremented by 6 and then the result is assigned to *x*. The various compound assignment operators used in Java are '+=', '-=', '*=', '/=' and '%='.

1.5.3 Relational Operators

Relational operators are used for comparing two values or expressions. Various relational operators provided by Java are less than '<', less than or equal to '<=', greater than '>', greater than or equal to '>=', equal to '==' and not equal to '!=' operator. They return the values of Boolean type, that can either be true or false. Let us consider an example of two variables, *a* and *b* having values 20 and 30 respectively. In this case, the expression *a<b* returns true whereas the expression *a>b* returns false. The operators == and != are also known as equality operators as they are used for checking the equality of operands.

NOTES

Note: All relational operators can work on integer, floating-point and character data types.

1.5.4 Boolean Logical Operators

These operators work on logical values that are either true or false. The logical operators are:

<i>Logical Operator</i>	<i>Java Operation</i>	<i>Java Expression</i>
&	Logical AND	a & b
	Logical OR	a b
!=	Not equal to	a != b
^	Logical XOR	a ^ b
	Short circuit OR	a b
&&	Short circuit AND	a && b
&=	AND assignment	a &= b
=	OR assignment	a = b
!	Logical unary NOT	! a
^=	XOR assignment	a ^= b
==	Equal to	a == b
?:	Ternary if-then-else	a ? b : c

For example:

```
{
int a = 5;
int b = 7;
if ((a>b) && ((b=19)>10))
System.out.println (" +b);
else
System.out.println (" +a);
System.out.println (" +b);
}
```

Output of the program:

5

This example displays 5 and then 7 because the first expression in the if condition is false and so the next expression is not evaluated that is b, which does not become 19.

1.5.5 Ternary Operators

It can be said that we can execute an if else statement using ternary **if-then-else operator** (?:). The syntax is given below:

```
Evaluation_Part ? codes_of_section_1
:codes_section_2;
```

First Evaluation_Part is evaluated if it is true, then codes_of_section_1 is executed else codes_section_2 is executed.

Program 1.1

```
class demo1
{
```



```

public static void main(String args[])
{
    int i = 64;
    int k=0;
    k =(i>k)? 10 : 5;
    System.out.println(k);
    k= (i<k)? 10: 5;
    System.out.println(k);
}
}

```

The output of the program:

10
5

From the above example, it can be checked that the ternary **if-then-else operator** (?:) works like if else.

Table 1.5: Operator Precedence Table

<i>Highest</i>	<i>Associativity</i>
() []	Left to right
! ~ - + ++ - -	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >=	Left to right
== != === !===	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= <<= >>= >>>= &= ^= =	Right to left
<i>Lowest</i>	

In case there are more than one **operators** in an expression, they are evaluated according to their precedence.

1.5.6 Operator Precedence

An expression consisting of more than one operator leads to a confusion as to which operator is to be evaluated first. For example, consider the expression:

a + b * c - d

NOTES

In this expression, the compiler needs to know which operator is evaluated first. For this, it is important to determine the precedence and associativity of operators.

- **Precedence:** The order or priority in which various operators in an expression are evaluated is known as precedence. Every operator in Java has a precedence associated with it. The operators with a higher precedence are evaluated before the operators with a lower precedence. For example, multiplication is performed before addition as the multiplication operator has higher precedence than the addition operator.
- **Associativity:** The order or priority in which operators of the same precedence are evaluated is known as associativity. For example, addition and subtraction operators have the same precedence. However, addition or subtraction may be performed on an expression depending upon the order of its occurrence.

The associativity of an operator can be either from left to right or from right to left. The operators with left to right associativity are evaluated from the left side while the operators with right to left associativity are evaluated from the right side. The precedence and the associativity of Java operators are listed in Table 1.6. Note that the precedence of operators decreases from top to bottom, that is, the priority is highest at the top.

Table 1.6 Precedence and Associativity of Java Operators

Operators	Description	Associativity
.	Direct member selector	Left to right
()	Function Call	Left to right
[]	Array subscript	Left to right
-	Unary minus	Right to left
++	Increment	Right to left
--	Decrement	Right to left
!	Logical negation	Right to left
~	Ones complement	Right to left
(type)	Casting	Right to left
*	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
>>>	Right shift with zero fill	Left to right

<	Less than	Left to right
<=	Less than or equal to	Left to right
>	Greater than	Left to right
>=	Greater than or equal to	Left to right
instanceof	Type comparison	Left to right
=	Equal to	Left to right
!=	Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
?:	Conditional operator	Right to left
=	Assignment Operator	Right to left
Op=	Shorthand assignment	Right to left

NOTES

Note: The operators in the same row have same precedence.

1.6 CONTROL STATEMENTS

Conditional statements, also known as selection statements, are used to make decisions based on a given condition. If the condition evaluates to `true`, one set of statements is executed, otherwise another set of statements is executed.

The `if` Statement

The `if` statement selects and executes statement(s) based on a given condition. The syntax of the `if` statement is:

For a Single Statement

```
if (condition)
    statement1;
    nextstatement;
```

For a Set of Statements

```
if (condition)
{
    statement1;
    statement2;
}
    nextstatement;
```

Here, if the condition evaluates to `true`, then a given set of statement(s) is executed. However, if the condition evaluates to `false`, then the given set of statements is skipped and the program control passes to the statement following the `if` statement. This is illustrated in Figure 1.8.

NOTES

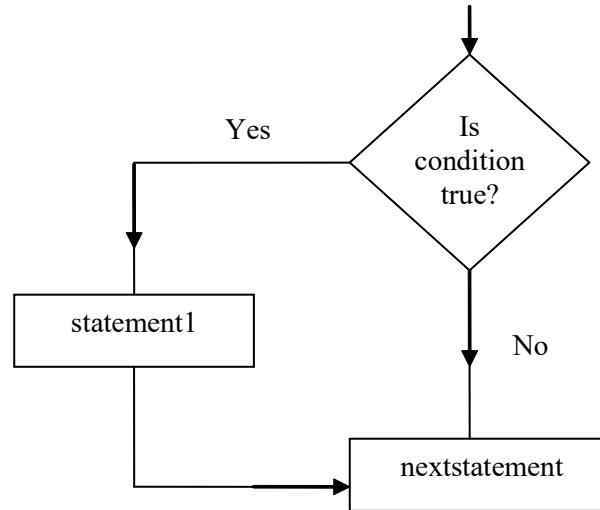


Fig. 1.8 Flow of Control in if Statement

The if-else Statement

The if-else statement causes one of the two possible statement(s) to execute, depending upon the result of the condition. The syntax of the if-else statement is:

```
if(condition) //if part
{
statement1;
}
else //else part
statement2;
nextstatement;
```

Here, the if-else statement comprises two parts, namely, if and else. If the condition is true, the statements within the if part is executed. However, if the condition is false, the statements within the else part is executed (Refer Figure 1.9).

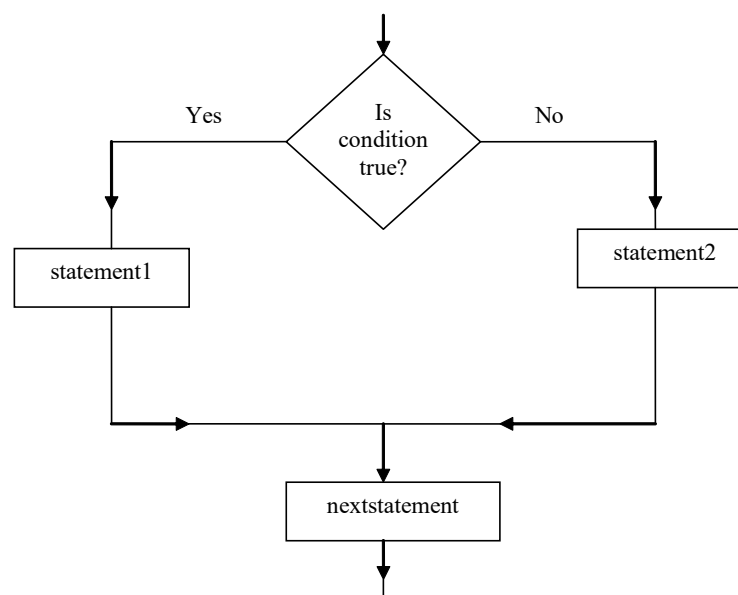


Fig. 1.9 Flow of Control in if-else Statement

Example 1.17: A program to demonstrate the use of `if-else` statement is as follows:

```
class ConditionalStatement
{
public static void main (String args[])
{
int i=5;
if (i > 0)
System.out.println("i is a positive number");
else
System.out.println("i is a negative number");
}
}
```

The output of the program is:

```
i is a positive number
```

Nested `if-else` Statement

A nested `if-else` statement contains one or more `if-else` statements. In other words, an `if-else` statement within another `if-else` statement is called a nested `if-else` statement. The `if-else` statement can be nested in three different ways which are discussed as follows:

- The `if-else` statement is nested within the `if` part.

The syntax is

```
if(condition1)
{
statement1;
    if(condition2)
        statement2;
else
    statement3;
}
else
statement4;
    nextstatement;
```

- The `if-else` statement is nested within the `else` part.

The syntax is

```
if (condition1)
    statement1;
else
{
    statement2;
    if (condition2)
        statement3;
else
    statement4;
}
nextstatement;
```

- The `if-else` statement is nested within both, the `if` and the `else` parts.

NOTES

NOTES

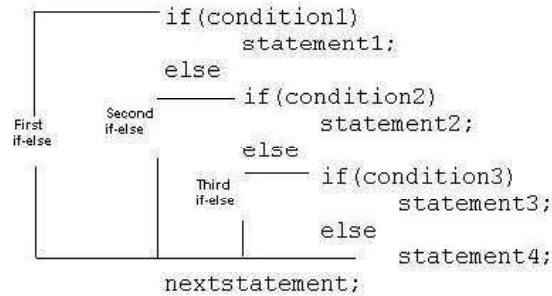
The syntax is:

```
if(condition1)
{
    statement1;
    if(condition2)
        statement2;
else
    statement3;
}
else
{
    statement4;
    if(condition3)
        statement5;
else
    statement6;
}
nextstatement;
```

The if-else-if Ladder

The if-else-if ladder, also known as the if-else-if staircase, has an if-else statement within the outermost else statement. The inner else statement can further have other if-else statements.

The syntax of the if-else-if ladders is:



Example 1.18: A program to demonstrate if-else-if statement is as follows:

```
import Java.io.*;
public class Grade
{
    public static void main (String[] args)
    {
        int marks=0;
        DataInputStream cin=new DataInputStream (System.in) ;
        try
        {
            System.out.print ("Enter the marks: ");
            //reading integer from keyboard
            marks=Integer.parseInt (cin.readLine());
            if(marks>90)
                System.out.println("Grade is A");
            else
                if(marks>75)
                    System.out.println("Grade is B");
                else
                    if(marks>60)
```

```
        System.out.println("Grade is C");  
    }  
    else  
        System.out.println("Grade is D");  
    }  
    }  
    catch (Exception e)  
    {}  
}
```

The output of the program is:

```
Enter the marks: 78  
Grade is B
```

Conditional Operator as an Alternative

The conditional operator, ‘? :’ selects one of the two values or expressions based on a given condition. Due to this decision-making nature of the conditional operator, it is sometimes used as an alternative to `if-else` statements. Note that the conditional operator selects one of the two values or expressions and not the statements as in the case of an `if-else` statement. In addition, it cannot select more than one value at a time, whereas the `if-else` statement can select and execute more than one statement at a time. For example, consider this statement.

```
max=(x>y ? x : y)
```

This statement assigns `x` and `y` to maximum.

The `switch` Statement

The `switch` statement selects a set of statements from the available sets of statements. The `switch` statement evaluates the value of an expression and compares it with the list of integer, character, short or byte constants. It should be noted that the case constants must be compatible with the expression type. When a match is found, all statements associated with that constant are executed (Refer Figure 1.10).

The syntax of the `switch` statement is:

```
switch (expression)  
{  
    case <constant1>: statement1;  
    [break;]  
    case <constant2>: statement2;  
    [break;]  
    case <constant3>: statement3;  
    [break;]  
    [default: statement4;]  
}  
nextstatement;
```

NOTES

NOTES

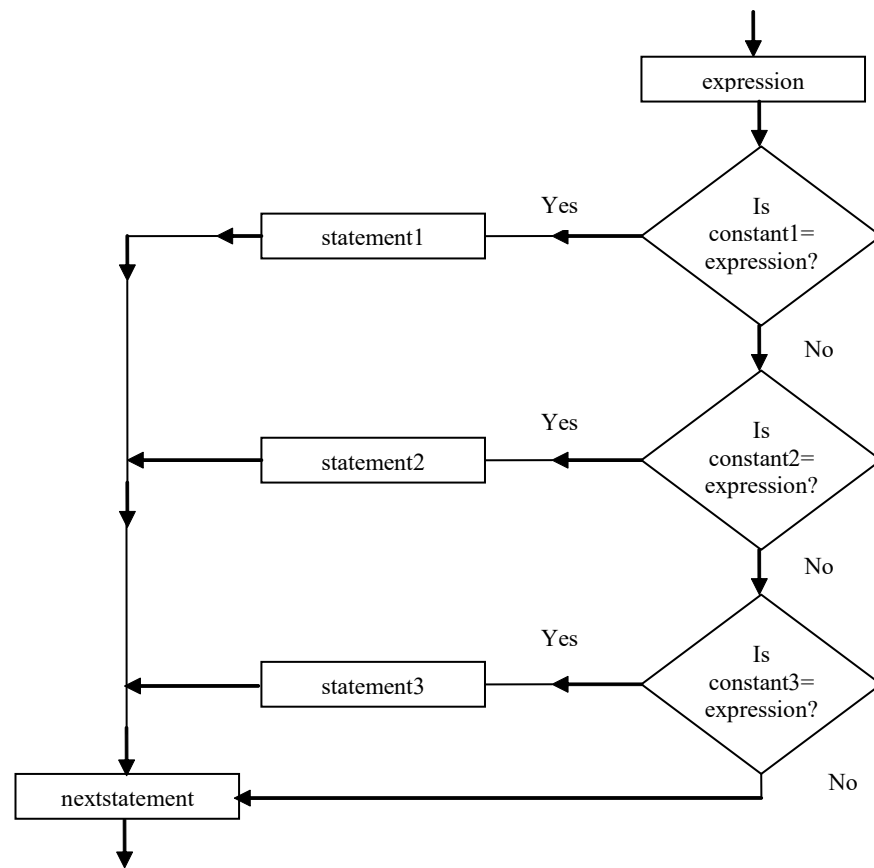


Fig 1.10 Flow of Control in switch Statement

The Java keywords `case` and `default` provide a list of alternatives. Note that it is not necessary for every `case` label to specify a unique set of statements. The same set of statements can be shared by multiple `case` labels. The keyword `default` specifies the set of statements to be executed in case no match is found. Note that there can be multiple `case` labels but there can be only one `default` label. However, `default` is an optional statement.

The `break` statements in the `switch` block are optional. However, it is used in the `switch` block to prevent a fall through. Fall through is a situation that causes execution of the remaining cases even after a match has been found. In order to prevent this, `break` statements are used at the end of statements specified by each `case` and `default`. This causes the control to immediately break out of the `switch` block and execute the next statement.

Similar to `if` and `if-else` statements, `switch` statements can also be nested within one another. A nested `switch` statement contains one or more `switch` statements within its `case` label or `default` label (if any).

Note: Switch statements cannot be used for testing floating-point values or string values.

Example 1.19: A program to demonstrate the use of `switch` statement is as follows:

```
class SwitchStatement
{
```



```
public static void main (String args [])
{
    int x=2;
    switch (x)
    {
    case 1: System.out.println("Day is Monday");
            break;
    case 2: System.out.println("Day is Tuesday");
            break;
    case 3: System.out.println("Day is Wednesday");
            break;
    case 4: System.out.println("Day is Thursday");
            break;
    case 5: System.out.println("Day is Friday");
            break;
    case 6: System.out.println("Day is Saturday");
            break;
    case 7: System.out.println("Day is Sunday");
            break;
    default: System.out.println("Invalid option!");
    }
}
```

The output of the program is:

Day is Tuesday

In Example 1.19, since the value of *x* is 2, therefore the message Day is Tuesday is displayed. In case the value of *x* is 8, the output Invalid option! will be displayed.

1.6.1 Nested Switch

`break` and `continue` statements allow the programmer to break out of the loop. However, they do not allow one to simply jump to another part of the program or out of the nested loop or `switch` statement. Java allows the user to jump from one block of statements to another with the help of labels. A label is an identifier which must follow the rules for naming identifiers in Java. It can be placed before the block of statements or loop followed by a colon (:). For example, consider the following statement:

```
LabelName: for ( ; ; )
{
    .
    .
}
```

The `break` statement passes control out of the innermost loop or the innermost `switch` statement and the `continue` statement continues with the next iteration of the innermost loop only. However, with the help of labels, the `break` statement can be used to cause the control to jump out of the outer loop or `switch` statement. For example, consider the following statements:

```
outerloop:    for (i=0;i<5;i++)
              {
innerloop:    for (j=1;j<5;j++)
```

NOTES

NOTES

```
        {  
            System.out.println("**");  
            if(i=j)  
                break outerloop;  
            .  
            .  
        }  
    }  
}
```

In this case, the `break` statement placed inside the `innerloop` will force the control to jump out of both the inner and outer loops. Similarly, with the help of labels, `continue` statement can be used to continue with the next iteration of the outer loop.

Example 1.20: A program to demonstrate use of the `continue` statement in a labeled loop is as follows:

```
class LabelExample  
{  
    public static void main (String args[])  
    {  
        outerloop: for (int i=0;i<3;i++)  
        {  
            for (int j=0;j<5;j++)  
            {  
                if (j>i)  
                    continue outerloop;  
                System.out.println(" i "+i+" j "+j);  
            }  
        }  
        System.out.println("Loop Ends");  
    }  
}
```

The output of the program is:

```
i 0 j 0  
i 1 j 0  
i 1 j 1  
i 2 j 0  
i 2 j 1  
i 2 j 2  
Loop Ends
```

In this program, if we substitute the `continue` statement with the `break` statement, the following output will be generated:

```
i 0 j 0  
Loop Ends
```

1.6.2 Iteration Constructs and Return

Statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as iteration statements. That is, as long as the condition evaluates to `true`, the set of statement(s) is executed. Various iteration statements used in Java are: *for loop*, *while loop* and *do-while loop*.

The for Loop

The `for` loop (Refer Figure 1.11) is one of the most widely used loops in Java. The `for` loop is a deterministic loop, that is, the number of times the body of the loop is executed, is known in advance.

The syntax of the `for` loop is:

```
for(initialize; condition; update)
{
    //body of the for loop
}
```

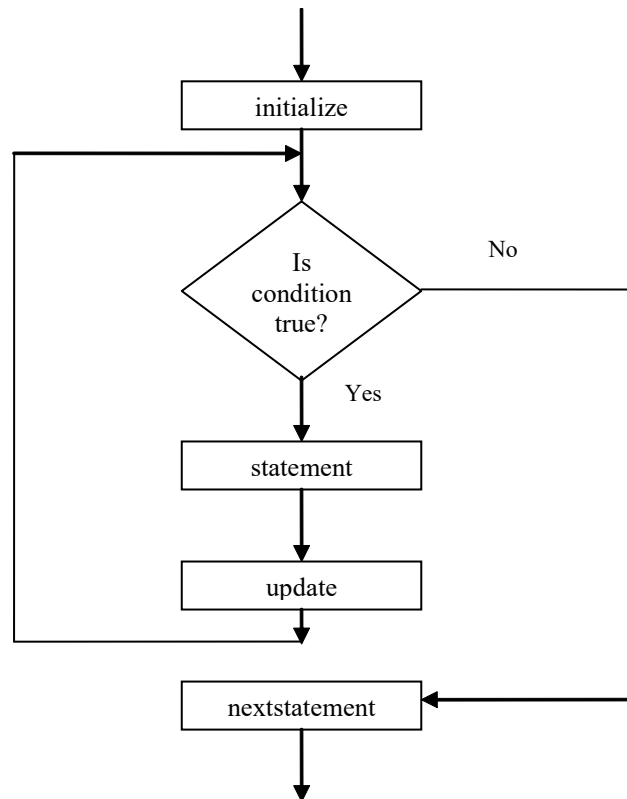


Fig 1.11 Flow of Control in for Loop

The `initialize` expression in the `for` loop can initialize one or more control variables. A loop can also update more than one variable in its `update` expression. Note that `initialize`, `condition` and `update` are optional expressions and are always specified in parentheses. All the three expressions are separated by semicolons. We can also create an infinite loop by excluding all the three expressions as follows:

```
for( ; ; )
{
    .
    .
}
```

Example 1.21: A program to display a count down using `for` loop is as follows:

```
class UpdateStatement
{
    public static void main(String args[])
```

NOTES

NOTES

```
{
for (int i=10;i>=1;i-)
{
System.out.print(" "+i);
}
System.out.println();
System.out.print("This is an example of for loop");
}
}
```

The output of the program is:

```
10 9 8 7 6 5 4 3 2 1
This is an example of for loop
```

The for Loop using Comma Operator

The for loop allows multiple variables to control the loop using a comma operator. That is, two or more variables can be used in the initialize and the update parts of the loop. For example, consider the following statement:

```
for (i=1,j=50;i<10;i++,j--)
```

This statement initializes two variables, namely `i` and `j` and updates them. Note that for loop cannot have more than one condition separated by a comma.

The while Loop

The while loop (Refer Figure 1.12) is used to perform looping operations when the number of iterations is not known in advance. That is, unlike for loop, the while loop is non-deterministic in nature.

The syntax of the while loop is as follows:

```
while(condition)
{
// body of the while loop
}
```

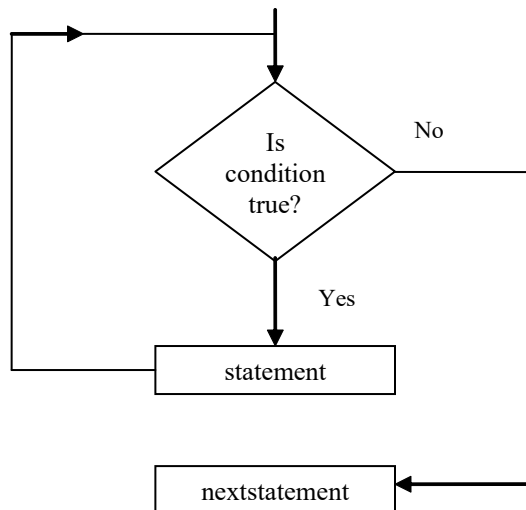


Fig 1.12 Flow of Control in while Loop

The following points should be noted about the `while` loop:

- Unlike `for` loops where explicit `initialize` and `update` expressions are specified, `while` loops do not specify any explicit `initialize` and `update` expressions. This implies that the control variable must be declared and initialized before the `while` loop and needs to be updated within the body of the `while` loop.
- The `while` loop executes as long as condition evaluates to `true`. If condition evaluates to `false`, then the body of `while` loop does not execute.

Example 1.22: A program to determine the sum of first `n` consecutive positive integers is as follows:

```
import java.io.*;
public class Sum
{
    public static void main(String[] args)
    {
        int n;
        int sum=0;
        DataInputStream cin=new DataInputStream (System.in);
        try
        {
            System.out.print("Enter n: ");
            //reading input from the user
            n=Integer.parseInt(cin.readLine());
            //loop to calculate the sum
            while(n>0)
            {
                sum=sum+n;
                n=n-1;
            }
            System.out.print("The sum is " + sum);
        }
        catch(Exception e)
        {}
    }
}
```

The output of the program is:

```
Enter n: 7
The sum is 28
```

The do-while Loop

As discussed earlier, in a `while` loop, the condition is evaluated at the beginning of the loop and if the condition evaluates to `false`, the body of the loop is not executed even once. However, if the body of the loop is to be executed at least once, no matter whether the initial state of the condition is `true` or `false`, the `do-while` loop is used. This loop places the condition to be evaluated at the end of the loop (Refer Figure 1.13).

NOTES

NOTES

The syntax of the do-while loop is as follows:

```
do
{
//body of do-while loop
}
while(condition);
```

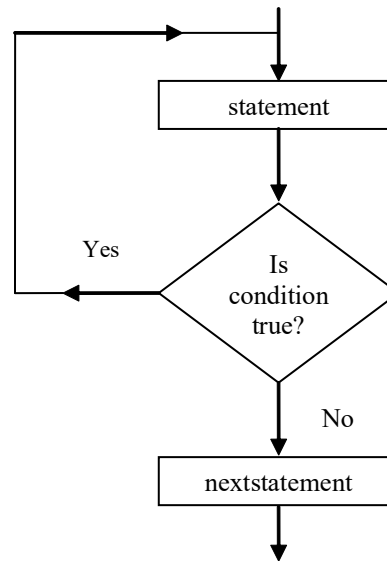


Fig 1.13 Flow of Control in do-while Loop

Example 1.23: A program to calculate the sum of an Arithmetic Progression (AP) is as follows:

```
class APSeries
{
    public static void main (String args[])
    {
        int first_term=1;
        int number_of_terms=5;
        int term=0;
        int i=1;
        int common_difference=2;
        int sum=0;
        System.out.print ("The terms are: ");
        do
        {
            term=first_term+(i-1)*common_difference;
            sum+=term;
            System.out.print (" "+term);
            ++i;
        }
        while(i<=number_of_terms);
        System.out.println();
        System.out.println ("The sum of A.P is: " +sum);
    }
}
```

The output of the program is:

```
The terms are: 1 3 5 7 9  
The sum of A.P is: 25
```

Nested Loops

Loops present within the body of another loop are known as nested. All the three loops (`for`, `while` and `do-while`) can be nested.

Example 1.24: A program to demonstrate the nested `for` loop is as follows:

```
class NestedLoop  
{  
    public static void main (String args[])  
    {  
        int a,b;  
        for (a=0;a<5;a++)                //outer loop  
        {  
            for (b=a;b<5;b++)            //inner loop  
            {  
                System.out.print ("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

The output of the program is:

```
*****  
****  
***  
**  
*
```

Jump Statements

Jump statements are used to alter the flow of control unconditionally. That is, jump statements transfer the control of a program unconditionally. The jump statements defined in Java are `break`, `continue` and `return`.

The `break` Statement

The `break` statement is extensively used in loops and `switch` statements. It immediately terminates the loop or the `switch` statement, by passing the remaining statements. Control then passes to the statement that immediately follows the loop or the `switch` statement. A `break` statement can be used in any of the three Java loops. In case of nested loops, a `break` will exit only a single loop, that is, the loop in which it is placed.

The `continue` Statement

The `continue` statement is used to continue the loop with its next iteration. In other words, it skips any remaining statements in the current iteration and immediately passes control to the next iteration. It does not terminate the loop (as in the case of `break` statements) rather it only terminates the current iteration of the loop. Like a `break` statement, a `continue` statement can be used in any of the three loops.

NOTES

NOTES

The return Statement

The `return` statement is used to transfer control out of the method explicitly. It transfers the control back to the caller and terminates the method in which it is present. When the `return` statement is encountered in the `main()` method, it transfers control back to the Java run-time system and terminates execution of the program.

Example 1.25: A program to add the factors of a number using `break` and `continue` statements is as follows:

```
class Jump
{
    public static void main (String args[])
    {
        int factor=0, number=10, sum=0;
        System.out.println("Number=" +number);
        while (true)
        {
            factor++;
            if (factor>number)
                break;
            if (number%factor!=0)
                continue;
            sum=sum+factor;
        }
        System.out.println("Sum of factors=" +sum);
        if (sum>0)
            return;
        System.out.println("This statement is not executed");
    }
}
```

The output of the program is:

```
Number=10
Sum of factors=18
```

Check Your Progress

11. What does the vector class contain?
12. Define the term relational operators.
13. Define the term operator precedence.
14. What is the other term used for conditional statements?
15. Which operator can be used as an alternative to the if-else statement?
16. What are labels?
17. What is iteration statement?
18. Write the difference between the while and the do-while loops.
19. Define the term nested loop.
20. Why are jump statements used?

1.7 ANSWERS TO ‘CHECK YOUR PROGRESS’

*Overview of Java, Data
Types and Variables,
Arrays, Operators and
Control Statements*

1. Java was developed at Sun Microsystems in 1991 by a team headed by James Gosling.
2. JDK is a set of various tools used to develop and execute Java programs.
3. Java with its multithreaded approach can run many programs concurrently, thereby saving processor time. Synchronization of code is an added feature of Java to run non-erroneous interactive applications.
4. JavaScript can be thought of as an extension to HTML, which allows authors to incorporate some functionality in their Web pages. It is the most popular scripting language on the Internet and works in all major browsers, such as Internet Explorer, Firefox, Chrome, Opera and Safari.
5. Generics is a powerful feature of Java. It was introduced by JDK 5. Using generics, a programmer can create classes, interfaces and methods that will work for various types of data in a type safe manner. One can define a single algorithm which is independent of data; then apply the same algorithm for various data types, without any changes.
6. JavaScript dialog boxes are interesting little ‘Pop-up’ boxes that can be used to display a message, ask for confirmation, user input, etc. They are very easy to create. Three types of dialog boxes exist in JavaScript—(i) ALERT, (ii) CONFIRM and (iii) PROMPT.
7. Objects are small, self-contained and modular units with a well-defined boundary. An object consists of a state and behaviour. The state of an object is one of the possible conditions that an object can exist in and is represented by its characteristics or attributes or data. The behaviour of an object determines how an object acts or behaves and is represented by the operations that it can perform.
8. The default value of a Boolean data type is False.
9. A variable is an identifier which represents a memory location that is used to store data values. Data stored at a particular location can be accessed by using a variable name.
10. A single-dimensional array requires one subscript to access an array element.
11. The vector class contains methods to store any number and any type of objects in a single unit called vectors.
12. Relational operators are used for comparing two values or expressions. Various relational operators provided by Java are less than ‘<’, less than or equal to ‘<=’, greater than ‘>’, greater than or equal to ‘>=’, equal to ‘==’ and not equal to ‘!=’ operator.

NOTES

NOTES

13. An expression consisting of more than one operator leads to a confusion as to which operator is to be evaluated first called operator precedence.
14. Conditional statements are also known as selection statements.
15. The conditional operator can be used as an alternative to the if-else statement.
16. Labels are names given to a block of code which allows a user to jump from one loop to another, within a nested loop.
17. The control statement that executes a set of statements repeatedly, based on a condition is known as iteration statement.
18. In the while loop, the condition is evaluated at the beginning of the loop and if the condition evaluates to false, the body of the loop is not executed even once. The do-while loop is used if the body of the loop is to be executed at least once, no matter whether the initial state of the condition is true or false.
19. Placing a loop within the body of another loop is known as nesting of loops.
20. Jump statements are used to transfer the control from one part of a program to another.

1.8 SUMMARY

- One of the reasons for the development of Java was the need for a software that would not be dependent on any platform and would be portable enough to be embedded in electronic devices like remote controls and microwave ovens.
- Java was developed at Sun Microsystems in 1991 by a team headed by James Gosling.
- JDK is a set of various tools used to develop and execute Java programs.
- In Java bytecode is a highly optimized set of instructions planned to be executed by the Java run-time system, which is referred to as the Java Virtual Machine (JVM).
- JavaScript was the first client-side Web scripting language. It first appeared in 1995 in Netscape 2.0. You can use JavaScript directly within a Web page without using any special tools to create or compile scripts and it works on most of today's browsers.
- JavaScript can be defined of as an extension to HTML, which allows authors to incorporate some functionality in their Web pages. So now, whenever the user presses the submit button, you do not necessarily have to invoke a Common Gateway Interface (CGI) script to do the processing.
- Generics is a new addition to Java. Before the evolution of generics, a programmer had to design different programs to deal with different data types having the same logic.

- A raw type is a parameterized type stripped of its parameters. The official term given to the stripping of parameters is type erasure. Raw types are necessary to support the legacy code that uses non-generic versions of classes.
- Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible.
- Objects are small, self-contained and modular units with a well-defined boundary. An object consists of a state and behaviour. The state of an object is one of the possible conditions that an object can exist in and is represented by its characteristics or attributes or data.
- A data type determines the type of operations that can be performed on the data.
- Primitive data types also known as ‘Built-in’ data types. These are fundamental data types provided by a programming language. In Java, primitive data types include integer, floating-point, character and Boolean.
- The Boolean data type can hold only Boolean values, that is, either true or false. The keyword Boolean is used to denote the Boolean data type. The default value of Boolean data type is false.
- A variable is an identifier which represents a memory location that is used to store data values. Data stored at a particular location can be accessed by using a variable name.
- Arrays are defined as a sequence of the same type of data elements of a fixed size. These data elements can be primitive or non-primitive data types.
- A single-dimensional array is the simplest form of an array that requires only one subscript to access an array element.
- Multi-dimensional arrays can be described as ‘An array of arrays’, that is, each element of the array is itself an array. A multi-dimensional array of dimension is a collection of items that are accessed with the help of n subscript values.
- A two-dimensional array is the simplest form of a multi-dimensional array that requires two subscript values to access an array element.
- Conditional statements, also known as selection statements, are used to make decisions based on a given condition. If the condition evaluates to true, one set of statements is executed, otherwise another set of statements is executed.
- A nested if-else statement contains one or more if-else statements. In other words, an if-else statement within another if-else statement is called a nested if-else statement

NOTES

NOTES

- The if-else-if ladder, also known as the if-else-if staircase, has an if else statement within the outermost else statement.
- The switch statement evaluates the value of an expression and compares it with the list of integer, character, short or byte constants.
- Labels are names given to a block of code which allows a user to jump from one loop to another, within a nested loop.
- Statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as iteration statements.
- The while loop is used to perform looping operations when the number of iterations is not known in advance. That is, unlike for loop, the while loop is non-deterministic in nature.
- Loops present within the body of another loop are known as nested. All the three loops (for, while and do-while) can be nested.
- Jump statements are used to alter the flow of control unconditionally. That is, jump statements transfer the control of a program unconditionally. The jump statements defined in Java are break, continue and return.

1.9 KEY TERMS

- **Genesis:** Generics is a new addition to Java.
- **Bytecode:** A simpler code made up of instructions that are one-byte long.
- **Native method:** A way to gain and merge the power of C or C++ programming into Java.
- **Architecture neutral:** The ability to work in diverse environments.
- **Compiler:** A computer program that transforms human readable source code of another computer program into the machine readable code that a CPU can execute.
- **Interpreter:** A computer program that reads the source code of another computer program and executes that program.
- **Java Virtual Machine (JVM):** A platform-independent execution environment that converts Java bytecode into machine language and executes it.
- **JavaScript:** A scripting language that reflects the object orientation of Web pages. It allows Web site authors to incorporate some functionality in their Web pages.
- **JavaScript dialog boxes:** Interesting little 'Pop-up' boxes that can be used to display a message, ask for confirmation, user input, etc. They are very easy to create.

- **Java:** An Object Oriented Programming (OOP) language created by James Gosling of Sun Microsystems.
- **Data types:** A data type determines the type of operations that can be performed on the data.
- **Array:** A sequence of the same type of data elements of a fixed size.
- **Subscript:** A positive integer value which indicates the position of an element in an array.
- **Single-dimensional array:** The simplest form of an array that requires only one subscript to access an array element.
- **Multi-dimensional array:** An array of arrays, that is, each element of the array is itself an array.
- **Operators:** A symbol or function which represents an operation.
- **Selection statement:** Statements used to make decisions based on a given condition.
- **Switch statement:** Selects a specific set of statements from an available set of statements.
- **Iteration statement:** Causes a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied.
- **Break statement:** Immediately terminates the loop or the switch statement, by passing the remaining statements.
- **Continue statement:** Continues the loop with its next iteration.
- **Return statement:** Transfers control explicitly out of the method.

NOTES

1.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Mention some of the differences between Java and C++.
2. What is the role of Java compiler in the execution of a program?
3. State about the Java applets.
4. What do you understand by JavaScript?
5. Write a JavaScript code in <HEAD>.
6. Define the term raw types in Java.
7. Define the term inheritance.
8. What do you understand by the term primitive data types.
9. what are types of literals?
10. State about the symbolic constants.
11. Define the term array.
12. Give one difference between single-dimensional and multi-dimensional arrays.

NOTES

13. What are vectors?
14. State about the arithmetic operators.
15. Define the term ternary operators.
16. What are control statements?
17. State the difference between the execution of the while and the do-while loops.
18. Differentiate between break, return and continue statements.

Long-Answer Questions

1. Discuss about the features of Java in detail.
2. Describe the purpose of JDK. Mention some tools of JDK and their purposes.
3. Explain the Java buzzwords giving appropriate examples.
4. Describe the JavaScript objects that are used for processing the HTML form.
5. Briefly the differences between Java and JavaScript.
6. Briefly explain the genesis of Java giving appropriate examples.
7. Write the features of object oriented programming giving appropriate examples.
8. Discuss and write the Java simple program and compiling.
9. Explain the data types and its types with the help of diagram.
10. What are the different ways of initializing a variable? Explain with example.
11. Analyse the types casting with the help of examples.
12. Briefly explain the array and its types giving appropriate examples.
13. Write the array declaration syntax in details.
14. Discuss how vectors are different from arrays.
15. Briefly explain the basic assignment operators and Boolean logical operators with the help of example.
16. Describe the operator precedence and its types with the help of table.
17. Discuss briefly if-else statement and conditional statement with the help of diagram.
18. Briefly explain the iteration statements and its types with the help of diagram.

1.11 FURTHER READING

Balagurusamy, E. 2007. *Programming with Java*, 3rd Edition. New Delhi: Tata McGraw-Hill.

Naughton, Patrick and Herbert Schidt. 1999. *Java 2: The Complete Reference*, 3rd Edition. New Delhi: Tata McGraw-Hill.

Das, Rashmi Kanta. 2013. *Core Java for Beginners*, 3rd Edition. New Delhi: Vikas Publishing House Pvt. Ltd.

Schildt, Herbert. 2006. *Java: The Complete Reference*, 7th Edition. New Delhi: Tata McGraw-Hill.

Hunter, Jason and William Crawford. 2001. *Java Servlet Programming*, 2nd Edition. California: O'Reilly Media.

Arnold, Ken, James Gosling and David Holmes. 2005. *The Java Programming Language*, 4th Edition. Boston: Addison-Wesley.

Wigglesworth, Joe and Paula Lumby. 1999. *Java Programming Advanced Topics*, 2 Edition. Boston: Course Technology.

Deitel, Paul and Harvey Deitel. 2011. *Java: How to Program*, 9th Edition. New Delhi: Prentice-Hall of India.

Overview of Java, Data Types and Variables, Arrays, Operators and Control Statements

NOTES



UNIT 2 CLASS, INHERITANCE, INTERFACES, PACKAGES AND EXCEPTION HANDLING

*Class, Inheritance,
Interfaces, Packages and
Exception Handling*

NOTES

Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Introduction to Class
 - 2.2.1 Method and Classes
 - 2.2.2 Method and Constructor Overloading
 - 2.2.3 Objects as Parameters
 - 2.2.4 Returning Objects
 - 2.2.5 Recursion
 - 2.2.6 Access Control/ Visibility
 - 2.2.7 Static and Final Classes
 - 2.2.8 Nested and Inner Classes
 - 2.2.9 String Class
 - 2.2.10 Command Line Arguments
- 2.3 Inheritance
 - 2.3.1 Member Access
 - 2.3.2 Super Class Variable
 - 2.3.3 Subclass Object
 - 2.3.4 Using Super to Call Superclass Constructors
 - 2.3.5 Multilevel Hierarchy
 - 2.3.6 Calling Constructor
 - 2.3.7 Overriding Methods
 - 2.3.8 Abstract Classes Method
 - 2.3.9 Final Class in Inheritance
- 2.4 Interface
- 2.5 Packages
- 2.6 Fundamentals of Exception Handling
 - 2.6.1 Types of Exception
 - 2.6.2 Try and Catch Keyword
 - 2.6.3 Finally Keywords
 - 2.6.4 Throw and Throws
 - 2.6.5 Nested Try Statements
 - 2.6.6 Java Build-In Exceptions
 - 2.6.7 User Defined Exceptions
- 2.7 Answers to 'Check Your Progress'
- 2.8 Summary
- 2.9 Key Terms
- 2.10 Self Assessment Questions and Exercises
- 2.11 Further Reading

NOTES

2.0 INTRODUCTION

The key objective of object-oriented programming is to represent various real-world objects as program elements. In Java, this objective is accomplished with the help of class that binds data and methods to manipulate that data together under a single entity. All OOP (Object Oriented Programming) concepts, such as data abstraction, encapsulation, inheritance and polymorphism are implemented with the help of classes. A class serves as a template that provides a layout which is common to all its instances known as objects. Thus, a class is only a logical abstraction that specifies what data and methods its objects will have, whereas objects are physical entities using which those data and methods can be used in a program.

Inheritance is one of the fundamental concepts of object-oriented programming. Using inheritance, you can create a general class that defines traits common to a set of related items. Other, more specific, classes can then inherit this class, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a super class, and the class that inherits the properties of super class is known as derived class or child class. Java uses extend keyword to support inheritance. An interface is defined just like a class but rather than using the keyword class, the keyword interface is used.

An exception signifies an illegal, invalid or unexpected issue during a program. Since exceptions are almost always assumed to be anticipated, you need to provide an appropriate exception handling. Exception handling means diverting the processing to a part of the program when an exception occurs. Java provides several built-in classes which define all types of exceptions. These exception classes are arranged in a hierarchy having Throwable class on the top. That is, the Throwable class is the superclass and all the exception classes inherit methods defined by it. Two immediate subclasses of the Throwable class are Exception class and Error class.

In this unit, you will study about the introduction to class, method and class in details, method and constructor overloading, objects as parameters, returning objects, recursion, access control/ visibility, static and final, nested and inner classes, string class and command line argument, inheritance and member access, super class variable and subclass object, super to call superclass constructors and multilevel hierarchy, calling constructor and overriding methods, abstract classes method and final in inheritance, interface and packages, fundamental of exception handling and types, try and catch keyword, finally keywords, throw and throws, nested try statements, Java built in exceptions and user defined exceptions.

2.1 OBJECTIVES

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

- Discuss the introduction to class
- Analyse the method and class in details

- Understand the method and constructor overloading
- Describe the objects as parameters
- Explain the returning objects and recursion
- Analyse the access control/ visibility
- Understand the static and final
- Elaborate the nested and inner classes
- Describe the string class and command line arguments
- Explain the inheritance and member access
- Define the super class variable and subclass object
- Understand the super to call superclass constructors and multilevel hierarchy
- Discuss the calling constructor and overriding methods
- Describe the abstract classes method and final in inheritance
- Define the interface and packages
- Explain the fundamental of exception handling and types
- Analyse the try and catch keywords
- Understand the finally keywords
- Explain the throw and throws
- Define the nested try statements
- Elaborate the Java build in exceptions and user defined exceptions

NOTES

2.2 INTRODUCTION TO CLASS

A class is a user-defined data type that can be used to create instances of its type called objects. Like any other user-defined data type, it also needs to be declared and defined in a program. A class definition specifies a new data type that can be treated as a built-in data type.

The syntax for defining a class is as follows:

```
class class_name
{
    //variables declaration
    //methods declaration
}
```

The variables declared in the class are known as instance variables. The variables and methods declared within the curly braces are collectively known as members of the class.

A class can also be empty, for example, the class definition given below is also valid.

```
class class_name
{
}
```

NOTES

Here, since the body of the class is empty, it does not contain any variables and methods so it cannot perform any useful action. However, this class can be successfully compiled and we can also create objects using it.

Note: In Java, there is no semicolon after closing brace in class definition.

Example 2.1: A simple class definition without any method is as follows:

```
class Cuboid
{
    int length;
    int width;        //variables declaration
    int height;
}
```

In this example, a class named `Cuboid` with three instance variables of type `int`, namely, `length`, `width` and `height` is created.

Defining Methods

As discussed earlier, a class consists of instance variables and methods. A class which consists of only variables (and without methods which manipulate them) cannot perform any useful operation. Therefore, to access the instance variables of a class and manipulate them, we must add methods in the class.

The syntax for defining a method is as follows:

```
return_type method_name (parameter_list)
{
    body of the method
}
```

where,

`return_type` is the type of data that is returned by the method.

`method_name` specifies the name of the method. This can be any name other than the keywords in Java.

`parameter_list` consists of a series of pairs of data type and identifiers separated by commas.

Note: The `parameter_list` can be empty and if a method does not return any value, its return type must be `void`.

For example, consider the following method definition:

```
int volume()
{
    body of the method
}
```

Here, the method `volume()` does not accept any parameter and returns a value of type `int`.

Example 2.2: A class definition with method is as follows:

```
class Cuboid
{
    int length;
    int width;        //variables declaration
    int height;
    int volume()     //method definition
    {
```

```

        return (length*width*height);
    }
}

```

Note: Methods must be declared immediately after the declaration of instance variables inside the body of the class.

NOTES

2.2.1 Method and Classes

Objects are data and methods bundled together into one logical software unit. The next question is how objects get into our software system and how objects get created.

A blue print that describes the techniques and variables common to all objects of a certain kind is called a class. Similarly, many objects can be constructed from a single class. A class outlines the properties of an object. Objects created from the same class show similar characteristics.

Declaring Classes

The syntax for declaring classes in Java is as follows:

```

class identifier
{
    class body
}

```

Here *identifier* specifies the name of the class. Class body consists of declarations and method definitions. Curly braces surround the class body.

For example:

Look at the following Java class declaration.

```

class ExampleClass
{
    char cc;
    int ii;
    double dd;
    void exampleMethod1 ( )
    {
        System.out.println ("Hello World");
    }
    void exampleMethod2 ( )
    {
        System.out.println("Welcome to India");
    }
}

```

In the above example. ExampleClass is the class name. Class body consists of three data item declarations:

```

char    cc;
int     ii;
double  dd;

```

and two method implementations:

Method 1

```

void exampleMethod1 ( )
{
    System.out.println ("Hello World");
}

```

NOTES

Method 2

```
void exampleMethod2 ()  
{  
    System.out.println ("Welcome to India");  
}
```

The New Operator

Here is how you can create instances of classes or object variables and invoke methods or member functions in Java.

```
ExampleClass ec;  
ec = new ExampleClass ();  
ec.exampleMethod1 ();  
ec.exampleMethod2 ();
```

The declaration 'ExampleClass ec' simply states what type of object variable ec will be. The object is actually created when the new operator is called.

The first two statements can be combined into a single statement as follows.

```
ExampleClass ec = new ExampleClass ();
```

A complete Java program is shown in which you declare a class and create objects of that class and invoke class member functions.

A Java program to show class declaration in class object creation:

```
public class Example1  
{  
    public static void main (String argv[ ] )  
    {  
        ExampleClass ec = new ExampleClass ();  
        ec.exampleMethod1 ();  
        ec.exampleMethod2 ();  
    }  
}  
  
class ExampleClass  
{  
    // char ch  
    // int ii  
    // double dd  
    void exampleMethod1 ()  
    {  
        System.out.println ("Hello World");  
    }  
    void exampleMethod2 ()  
    {  
        System.out.println ("Hello Beautiful World");  
    }  
}
```

On compiling and running, this program will print an output shown as follows:

Output of the program:

```
Hello World  
Hello Beautiful World
```

Java Objects as Class Member Variables

Java class member variables are of two types. One is the built-in type, like char, int, float, double, etc. In the examples you have seen so far, all the class member variables were of built-in types. The other type of class member variables are Java objects. A Java program in which a Java object is used as class member variable is shown as follows

A Java program illustrates that Java objects can be used as class members:

```
public class Example
{
    public static void main(String argv[ ])
    {
        ExampleClass ec1=new ExampleClass ();
        ExampleClass ec2=new ExampleClass ("Hello World");
        ec1.exampleMethod();
        ec2.exampleMethod();
    }
}
class ExampleClass
{
    String ss;           // class number is an object here
    String class is defined in
                        // JDK Library

    ExampleClass ()
    {
        ss="HelloBeautiful World";    // default constructor
    }
    ExampleClass (String s)
    {
        ss=s;
    }
    void exampleMethod ()
    {
        System.out.println(ss);
    }
}
```

Output of the program:

```
Hello World
Hello Beautiful World
```

Deriving Classes

A class can be built on another class that is already defined and existing. The newly built class is called the derived class or child class. The child class inherits all the properties of the parent class. That is to say that the child class inherits all the member variables and methods of the parent class. In addition, the child class can have its own member variables and methods.

The syntax for deriving a class from another class is as follows:

```
class ChildClass extends ParentClass
{
    // body of the child class
}
```

NOTES

NOTES

Here the ChildClass is the newly derived class. ParentClass is the already existing and previously defined class. **extends** is a keyword. **body** of the child class is the extra feature the ChildClass has over that of the ParentClass.

The Java program in which a child class is derived from a parent class is as follows. It also exhibits that the derived class inherits the member variables and member functions of the parent class.

A Java program illustrates class inheritance.

```
public class Example
{
    public static void main (String argv[ ])
    {
        ChildClass ch=new ChildClass ();
        System.out.println("ch.pi="+ch.pi); // inherited from
parent
        System.out.println("ch.ci="+ch.ci);
        ch.parentMethod(); // inherited
from parent
        ch.childMethod();
    }
}
class ChildClass extends ParentClass
{
    int ci;
    ChildClass ()
    {
        ci=100;
    }
    void childMethod ()
    {
        System.out.println("Hello World");
    }
}

    int pi;
class ParentClass ()
{
    pi=10;
}
void parentMethod ()
{
    System.out.println("Hello Beautiful World");
}
}
```

Output of the program:

```
ch.pi=10
ch.ci=100
Hello World
Hello Beautiful World
```

2.2.2 Method and Constructor Overloading

Two or more methods can be defined within the same class that shares the same name, till the time their parameter declarations are different in Java. Renaming the

same method name with different arguments with the same or different return type is known as overloading method.

*Class, Inheritance,
Interfaces, Packages and
Exception Handling*

The Java program that shows the method of overloading is as follows.

```
public class Overloading
{
    public static void main (String argv[ ])
    {
        ExampleClass ec=new ExampleClass ( );
        ec.exampleMethod ( );          // calling the method
        ec.exampleMethod (10);        // calling the overloaded
method
    }
}
class ExampleClass
{
    void exampleMethod ( )    // a method to be overloaded
    {
        System.out.println ("Hello World");
    }
    void exampleMethod (int i)    // overloading
    {
        System.out.println ("Welcome to India");
    }
}
```

NOTES

Output of the program:

```
Hello World
Welcome to India
```

Constructors Overloading

In addition to overloading normal methods, you can also overload constructor methods. Remember constructors do not return any value and even void should not be included in the constructor header.

The Java program that shows the constructor overloading is as follows.

```
public class Box
{
    double width;
    double height;
    double depth;
    Box (double w, double h, double d)    // This is the constructor
for Box.
    {
        width = w;
        height = h;
        depth = d;
    }
    double volume ( )    // Compute and return volume
    {
        return width * height * depth;
    }
}
```

2.2.3 Objects as Parameters

So far, you have only been using simple types as parameters to methods. Class member variables can be Java objects as well.

NOTES

A Java program to illustrate that Java objects can be used as class members:

```
public class ObjParameter
{
    public static void main (String argv[ ])
    {

        ExampleClass ec1=new ExampleClass ( );
        ExampleClass ec2=new ExampleClass ("Hello World" );
        ec1.exampleMethod ( );
        ec2.exampleMethod ( );
    }
}
class ExampleClass
{
    String ss;          // Class number is an object here
                      // String class is defined in JDK Library

    ExampleClass ( )
    {
        ss = "Welcome to India";           // default
constructor
    }

    ExampleClass (String s)
    {
        ss = s;
    }

    void exampleMethod ( )
    {
        System.out.println(ss);
    }
}
```

Output of the program:

```
Hello World
Welcome to India
```

2.2.4 Returning Objects

The return data type is declared in the function declaration in the `main ()` function or the calling function and the declarator is indicated in the first line of the function definition. If no value is to be returned, the return data type `void` is specified. `Void` simply means `NULL` or nothing. Therefore, it does not fall in any other data types, such as `integer` or `float` or `char`.

The return value as you have seen is the result of computation in the called function. You return a value, which is stored in a data type in the called function. The return value means that the value, thus stored in the called function is assigned or copied to a variable in the `main ()` or calling function. Therefore, to receive

the result, a data type should have been declared and preferably initialized in the calling function.

The return statement can be any of the following types:

```
return (sum) ;  
return v1;  
return " true" ;  
return 'Z' ;  
return 0;  
return 4.0 + 3.0;
```

In some examples, you have returned variables whose values are known when they are returned and in other examples, you return constants. You can even return expressions. If the return statement is not present, it means the return data type is `void`.

You can also have multiple return statements in a function. However, for every call, only one of the return statements will be active and only one value will be returned.

Arrays and Functions

There is no restriction in passing any number of values to a function; the restriction is only in the return of values from a function. Therefore, arrays can be passed to a function without any difficulty, one element at a time, as follows:

```
#include <stdio.h>  
int main()  
{  
    int a[]={1,2,3,4,5};  
    int j;  
    int func(int a);  
    for (j=0; j<=4; j++)  
        func (a[j]);  
    .....  
}  
int func(int c)  
{  
    .....  
}
```

Here, `func` has been declared as a function passing a single integer. Note here that the declaration or the prototype gives only the format of the parameters passed. The values are only indicative and are not actual values. They are the formal values. Therefore, the parameters declared inside the parentheses act only as a checklist. They cannot be used in the main function elsewhere without actually declaring them on top of the function. But, for this rule, there would have been a conflict between `a[]` which is an array and `a` which is a simple variable. Here, no conflict arises because `a` is not recognized in the `main` function. It is only a checklist to see that whenever the function calls `func`, an integer has to be passed. If we try to pass a `float`, the compiler will detect an error. This is not so in the case of variables defined in the function declarator above the functions body, as they are recognized as actual names. In this case, `int c` is declared as a variable in `func`. The initial value will be the same as passed by the calling function. Thus, since `a` is used in the function declaration, only one integer can be passed to the

NOTES

NOTES

function `func`. Actually, the entire array can be passed to a function irrespective of its size, by suitable declaration, as the following example indicates.

```
/*Example 2.3*/
To find the greatest number in an array*/
#include <stdio.h>
int main()
{
    int array[] = {8, 45, 5, 911, 2};
    int size=5, max;
    int fung(int array[], int size);
    max=fung(array, size);
    printf("%d\n", max);
}
int fung(int a1[], int size)
{
    int i, j, maxp=0;
    for (j=0; j<size; j++)
    {
        if (a1[j] > maxp)
        {
            maxp=a1[j];
        }
    }
    return maxp;
}
```

Result of the program

911

The objective of Example 3.7 is to find the greatest number in an array. In the program, an array called `array` is initialized with 5 values as given below:

```
int array[] = {8, 45, 5, 911, 2};
```

size is declared as 5 and a function called `fung` has been declared. It will pass an array and an integer to the called function. The array size has been kept open and the called function will return an integer. The next statement calls `fung` and passes all elements of the array and an integer 5 equal to `size`. The function gets the actual values and `size=5`. The maximum value in the array is found in the `for` loop and stored in `maxp`. The value `maxp` is returned to the main function and printed there. Thus, the function is called by value.

Call by Value

In this section, you have been calling functions by passing values. For example, function calls in some of the above programs are as follows:

```
change(a, b);
rev = reverse(num);
```

The values passed to the function `change` are `a` & `b` which are known. Similarly, while calling function `reverse`, we pass `num`. This is called call by value. When you call functions by value, the called functions can return only one value.

Call by Reference

You can enable a function to return more than one value. One way of accomplishing it is by call by reference.

2.2.5 Recursion

Java supports recursion. Recursion is the process of defining a method that calls itself.

An example of recursion is the computation of the factorial of a number.

A Java program that illustrates the recursion of the factorial of a number is as follows:

```
class Factorial
{
    int fact (int n);    // this is a recursive method
    int res;

    if (n == 1)
        return 1;
    res = fact (n-1) * n;
    return res;
}
}
class Recursion
{
    public static void main (String args [ ] )
    {
        Factorial f = new Factorial ( );
        System.out.println ("Factorial of 3 is " + f.fact(3));
    }
}
```

Output of the program:

```
Factorial of 3 is 6.
```

2.2.6 Access Control/ Visibility

In the programs discussed so far, the class members are accessible everywhere in the program and the subclass can inherit all the variables and methods of a superclass by using the keyword `extends`. However, there might be certain situations when you want to restrict the accessibility of the members of a class for various reasons with security being one of them. For this, Java provides three types of visibility controls— `public`, `private` and `protected`. They are also known as access modifiers.

- **Public:** When a member of a class is declared as `public`, it can be accessed everywhere in the program.
- **Private:** A member declared as `private` can be accessed only within a class.
- **Protected:** A member declared as `protected` is accessible not only to all the classes and subclasses in the same package but also to subclasses in other packages.

If no visibility control is specified, the data member of a class is visible only within the same package by default. You may use the word ‘friendly’, in connection with the default access, however, it is not a Java keyword.

Apart from the visibility controls discussed, there is another visibility control namely `private protected` which was used with the release of Java 1.0.

NOTES

NOTES

However it has been dropped in Java 1.1 and further versions. If data members are declared as private protected, they can be accessed by all the subclasses irrespective of the package to which they belong. However, they are not visible in other classes of the same package (Refer Table 2.1).

Table 2.1 Visibility of Field Inside a Class

Access location	Same class	Subclass in same package	Non-subclasses in same package	Subclass in other packages	Non-subclasses in other packages
Public	?	?	?	?	?
Private	?	X	X	X	X
Protected	?	?	?	?	X
Default	?	?	?	X	X

These are certain rules that can be followed to select the appropriate visibility control. If the data member is to be made:

- Visible everywhere in the program, use `public` visibility control.
- Invisible everywhere except within the same class, use `private` visibility control.
- Visible everywhere in the same package and subclasses in other package, use `protected` visibility control.
- Visible everywhere in the same package only, use the default visibility control.
- Visible only in the subclasses irrespective of packages, use `private protected`.

Example 2.4: A program to demonstrate the use of private, public and default visibility controls

```
class FirstClass
{
    int i;          //default variable
    private int j; //private variable
    public int k;  //public variable
    int get_data(int l)
    {
        j=l;
        return j;
    }
}
class VisibilityTest
{
    public static void main(String[] args)
    {
        FirstClass obj=new FirstClass();
        obj.i=20; //i can be accessed directly
        obj.k=60; //k can be accessed directly

        //this will cause error as j cannot be
        //accessed directly
        //obj.j=60;
    }
}
```

```
//j can be accessed through its method
int a=obj.get_data(40);
System.out.println("The value of i,j and k are:
"+obj.i+" ,"+a+" and "+obj.k+" respectively.");
}
}
```

NOTES

The output of the program is:

The value of i,j and k are: 20,40 and 60 respectively.

In this example, variable `i` is set to default access, variable `j` is declared as `private` and variable `k` is declared as `public`. In the class `VisibilityTest`, only variables `i` and `k` can be accessed directly. The variable `j` can be accessed through its method `get_data()`.

2.2.7 Static and Final Classes

A variable which has a constant value or a method that cannot be overridden in a subclass or a child class is specified by a `final` keyword. The syntax for a `final` keyword is shown as follows:

```
final public int a = 10;
final public void classMethod();
```

This keyword is used to reference the current object inside a class definition by passing the need for an instance variable.

Static Keyword

Sometimes you need a common variable or method for all the objects derived from a class. The `static` keyword specifies that a variable or a method is the same for all objects of a particular class.

Each time you create an object from a class, space is allocated for the variables. If a variable is declared `static`, space is allocated only once, i.e., the first time when you create the object. This space is shared by all subsequent objects. `Static` method is one whose mention is the same for all objects. `Static` method has access to `static` variables only. The syntax for declaring a `static` variable or method is as follow:

```
static int a;
static void classMethod();
```

The possibility of defining a class within another class is called as nested classes. The range of a nested class is enclosed by the extent of its enclosing class. Nested classes are of two types which are `static` and `non-static`. A `static` class has a `static` modifier applied i.e., it must contact the members of its surrounding class through an object.

A `non-static` nested class which is an inner class, is very important because all the variables and methods of its outer class can be accesses and referred directly.

The following program shows how to define and use an inner class:

```
class Outer
{
    int outer_x = 10;
    void test()
}
```

NOTES

```
{
    Inner inner = new Inner ( );
    inner.display ( );
}
class Inner
{
    void display ( )
    {
        System.out.println("Display : outer_x =" + outer_x);
    }
}

class InnerClass
{
    public static void main(String args [ ])
    {
        Outer outer = new Outer ( );
        outer.test ( );
    }
}
```

Output of the program:

```
Display : outer_x = 10
```

2.2.8 Nested and Inner Classes

In Java, four types of **inner classes** are used. These are:

- **static Inner Class**
- **Non-static Inner Class**
- **Local Inner Class**
- **Anonymous Inner Class**

Suppose **X** is the **outer class** and **Y** is the corresponding **inner class** present in **X**. When the source code containing these class definitions is compiled, then two class files are created. One is class **X** and the other is **XS\$Y**.

A **nested** or **inner class** is the member of its enclosing class. **Nested class** can be declared by using any access modifier. In Java, **outer class** or enclosing class is declared by using a **public** or a no access modifier. **Nested class** is able to access **private**, **protected**, **no access** or **public** member of enclosing class.

Advantages

The advantages of using inner class are as follows:

- Logical grouping of classes
- Increased encapsulation
- More readable, maintainable code

static Inner Class/Nested Class

static inner class is popularly known as nested class. If an **inner class** uses **static** modifier, then the **inner class** is treated as a **static inner class**. The following example clarifies this:

Program 2.1

```
class X
{
    static class Y
    {
    }
}
```

Here **Y** class is treated as `static inner class`.

Key Points of static Inner Class

The key points are as follows:

- A programmer declares the **static inner class** by using any access specifier.
- A **static inner class** accesses the `static` member of outer or enclosing class through the outer class name or directly.
- A **static inner class** accesses the non-static member of the outer or enclosing class through the instance or object of the outer class.
- A `static` member of the **static inner class** accesses the non-static member of the **inner class** through the instance of **static inner class**.
- A `static` member of the **static inner class** accesses the `static` member of the **inner class** through the name of the **static inner class** or directly.
- A non-static member of the **static inner class** accesses the `static` member of the **inner class** through the **inner class** name or directly.
- A non-static member of the **static inner class** accesses the non-static member of the **inner class** directly.
- A **static inner class** supports inheritance.

Program 2.1

```
public class demo {
    static class X
    {
        static int j=90;
    }
    public static void main(String[] args) {
        System.out.println(demo.X.j);
    }
}
```

Output of the program:

90

In this program, the class `demo` contains a **static** inner class **X**. **X** has a `static` member `j`. `static` members can be accessed through class name. So to access the member variable `j` of the **static** inner class **X**, one has to write `demo.X.j` or simply `X.j`. The following example clarifies this:

NOTES

NOTES

Program 2.4

```
public class demo {
    static class X
    {
        int j=90;
    }
    public static void main(String[] args) {
        X a=new X();
        System.out.println(a.j);
    }
}
```

Output of the program:

90

Here the **static** inner class **X** contains a non-static instance variable **j**. Non-static instance variables can only be accessed through object name. Therefore, the object of class **X** has been created. Have a look at the following example:

Program 2.4

```
public class Inner2
{
    String name;
    static int roll;
    private String getName(String n)
    {
        name=n;
        return name;
    }
    static int getRoll(int r)
    {
        roll=r;
        return roll;
    }
    static class Test
    {
        int age=10;
        static String add="Cuttack";
        static void display()
        {
            Inner2 i1=new Inner2();
            Test t=new Test();
            System.out.println("Name Is "+i1.getName("Sai"));
            System.out.println("Roll Number Is "+Inner1.getRoll(1));
            System.out.println("Age Is "+t.age+"\t"+"Address Is "+Test.add);
        }
    }
}
```

```
    }  
    public static void main(String args[])  
    {  
        Test.display();  
    }  
}
```

Output of the program:

```
F:\>java Inner 2  
Name Is Sai  
Roll Number Is 1  
Age Is 10 Address Is Cuttack
```

Here **Test** is a **static** inner class and `display()` is a static method. So `display()` is called through a class name. Inside the body of `display()`, the `getRoll()` method is called through a class name, i.e., **Inner2** since it is a static method. The following example shows this:

Program 2.5

```
public class Inner3  
{  
    String name;  
    static String add;  
    static int age;  
    private static class Test1  
    {  
        int roll;  
        String getName(String n)  
        {  
            Inner3 i3=new Inner3();  
            i3.name=n;  
            return i3.name;  
        }  
        int getAge(int a)  
        {  
            Inner3.age=a;  
            return Inner3.age;  
        }  
    }  
    protected static class Test2 extends Test1  
    {  
        int getRoll(int r)  
        {  
            roll=r;  
            return roll;  
        }  
        String getAddress(String a)  
        {
```

NOTES

NOTES

```
        Inner3.add=a;
        return Inner3.add;
    }
}

public static void main(String args[])
{
    Test2 t=new Test2();
    System.out.println("Name Is "+t.getName("Sai"));
    System.out.println("Age Is "+t.getAge(10));
    System.out.println("Roll Number Is "+t.getRoll(1));
    System.out.println("Address Is "+t.getAddress("Cuttack"));
}
}
```

Output of the program:

```
F:\>java Inner 3
Name Is Sai
Age Is 10
Roll Number Is 1
Address Is Cuttack
```

In this program, **Test1** is a static private inner class and **Test2** is a protected **static inner class**. Since **Test2** is private, it can only be accessed within the outer class object, and since **Test2** extends **Test1**, the object of **Test2** can call the methods of **Test1**. The next example clarifies this further.

Program 2.6

```
class X
{
    static class Y
    {
        static int j=10;
    }
}

public class demo
{
    public static void main(String[]args)
    {
        System.out.println(X.Y.j);
    }
}
```

Output of the program:

```
10
```

Y is a **static** inner class present in class **X**. **Y** contains a static variable **j**. A static member can be accessed through a class name. Therefore, to access **j**, one has to write **X.Y.j**. This is shown in the following example:

Program 2.7

```
class X
{
    static class Y
    {
        int j=10;
    }
}
public class demo {
    public static void main(String[]args) {
        X.Y a=new X.Y();
        System.out.println(a.j);
    }
}
```

Output of the program:

10

Y is a **static** inner class having an instance variable **j** present in class **X**. In order to access **j**, one has to create the object of **Y**. However, it is not possible to create the object of **Y** directly, because **Y** is inside class; hence not visible to class **demo**. One has to create the reference of class **Y** through the outer class **X**. To create the object, the constructor of class **Y** has to be called through the outer class **X**, as shown in the above program.

Non-static Inner Class

Class declared within another class without using `static` modifier is treated as a **non-static inner class**. A **non-static inner class** is popularly known as an **inner class**. A **non-static inner class** is declared by using any access modifier. This can be seen in Program 2.8.

Program 2.8

```
class X
{
    class Y
    {
    }
}
```

Here class **Y** is treated as **non-static inner class**.

Key Points of Non-Static Inner Class

- A **non-static** inner class non-static member, accesses the `static` member of the outer class either directly or through the outer class name.
- A **non-static** inner class non-static member, accesses the non-static member of the outer class directly without creating any instance of the outer class.
- Within the non-static inner class, one cannot declare any `static` member.
- This **inner class** is very popularly used.

NOTES

NOTES

Program 2.9

```
public class Inner4
{
    String name;
    static int age;
    private String getName(String n)
    {
        name=n;
        return name;
    }
    static int getAge(int a)
    {
        age=a;
        return age;
    }
    private class Test1
    {
        int roll;
        String add;
        int getRoll(int r)
        {
            roll=r;
            return roll;
        }
        String getAddress(String s)
        {
            add=s;
            return add;
        }
        void display()
        {
            System.out.println("Name Is "+getName("Sai"));
            System.out.println("Age Is "+Inner4.getAge(10));
            System.out.println("Roll Number Is "+getRoll(1));
            System.out.println("Address          Is
"+getAddress("Cuttack"));
        }
    }
    public static void main(String args[])
    {
        Inner4 t1=new Inner4();
        t1.show();
    }
    void show()
    {
```

```
Test1 t1=new Test1();  
t1.display();  
}  
}
```

Output of the program:

```
Name Is Sai  
Age Is 10  
Roll Number Is 1  
Address Is Cuttack
```

In the above example, the main class is **Inner4** which contains a private **inner class Test1**. Inside the main class, the programmer has created an **Inner4** class object which then calls the `show()` method present inside **Inner4** class. In `show()`, the programmer has created an object of the private **inner class Test1** and through that object, the `display()` method present inside the **inner class Test1** has been called. In `display`, the programmer has called the `getName()` method directly to print the name and then the static `getName()` method of outer class by using the outer class name.

Program 2.10

```
public class Outer1  
{  
    String name;  
    int roll;  
    public class Inner1  
    {  
        String getName(String n)  
        {  
            name=n;  
            return name;  
        }  
        int getRoll(int r)  
        {  
            roll=r;  
            return roll;  
        }  
    }  
    public static void main(String args[])  
    {  
        Outer1 o1=new Outer1();  
        Inner1 i1=o1.new Inner1();  
        System.out.println("Name Is "+i1.getName("Asit"));  
        System.out.println("Roll Number Is "+i1.getRoll(4));  
    }  
}
```

Run the program by: C:\>java Outer1

NOTES

NOTES

Output of the program:

Name Is Asit
Roll Number Is 4

Program 2.11

```
class X
{
    class Y
    {
        int j=10;
    }
}
public class demo
{
    public static void main(String []args)
    {
        X a=new X();
        X.Y z=a.new Y();
        System.out.println(z.j);
    }
}
```

Output of the program:

10

Y is a non-static **inner class** present in **X**. **Y** is not visible to class **demo**. To create the reference of class **Y**, one has to use the outer class **X** as shown in the program. Since **Y** is **non-static inner class**, its constructor cannot be called through the outer class name **X**. For this purpose, an object of outer class **X** is needed. Here **a** is the object of outer class **X**. Through the object of outer class, constructor of **Y** is invoked (see the program). Once the object of class **Y** is created, one can access its instance variable **j**. The following example program shows this.

Program 2.12

```
public class demo
{
    class Y
    {
        int j=10;
    }
    public static void main(String[]args)
    {
        demo a=new demo();
        Y z=a.new Y();
        System.out.println(z.j);
    }
}
```


Output of the program:

10

Since the **inner class Y** is present inside the `demo`, one can simply use the class name **Y** to create the reference of class **Y**. However, to create an object, one needs the object of class `demo`. It is only through the object of class `demo` that it is possible to call the constructor of class **Y**.

Local Inner Class

Class declared within the method is treated as a **local inner class**.

Program 2.13

```
class X
{
    void show()
    {
        class Y
        {
        }
    }
}
```

Here class **Y** is treated as **local inner class** as it is declared within the method `show()`.

Key Points of Local Inner Class

- A **local inner class** is declared through a no-access modifier, but cannot be declared through `public`, `private` and `protected` access modifier.
- A **local inner class** accesses the private member of the outer class.
- A **local inner class** non-static member directly accesses the static member of the outer class.

Program 2.14

```
public class demo
{
    static int a=10;
    public static void main(String[] args) {
        class X
        {
            int j=a;
        }
        System.out.println(new X().j);
    }
}
```

NOTES

NOTES

Output of the program:

10

A **local inner class** non-static member directly accesses the non-static member of the outer class, if the **inner class** is inside a non-static method. Otherwise, a compilation error will be generated.

Program 2.15

```
public class demo
{
    int a=10;
    public static void main(String[]args)
    {
        demo d=new demo();
        d.fun();
    }
    void fun()
    {
        class X
        {
            int j=a;
        }
        System.out.println(new X().j);
    }
}
```

Output of the program:

10

A **local inner class** can only access the `final` member of the method where the class is declared. If one tries to access any other local variable, then compile time error will be generated. This can be seen in the following example:

Program 2.16

```
public class demo
{
    public static void main(String[]args)
    {
        final int a=50;
        class X
        {
            int j=a;
        }
        System.out.println(new X().j);
    }
}
```

Output of the program:

50

A programmer creates the object of a **local inner class** within the method where the class is declared.

A **local inner class** present in a method can be `static` and `abstract`. The given example shows this.

Program 2.17

```
public class Local
{
    String name;
    private static int roll;
    protected static int age;
    String add;
    void go(final int a,int b)
    {
        final int x=a+b;
        int y=a-b;
        class Inner
        {
            String getName(String n)
            {
                name=n;
                return name;
            }
            int getAge(int a)
            {
                age=a;
                return age;
            }
            int getRoll(int r)
            {
                roll=r;
                return roll;
            }
            String getAddress(String s)
            {
                add=s;
                return add;
            }
            void show()
            {
```

*Class, Inheritance,
Interfaces, Packages and
Exception Handling*

NOTES

NOTES

```
System.out.println("Name Is "+getName("Sai"));
System.out.println("Age Is "+getAge(10));
System.out.println("Roll Number Is "+getRoll(1));
System.out.println("Address Is
"+getAddress("Cuttack"));
    }
    void display()
    {
        System.out.println("Value of A Is "+a);
        System.out.println("Value of X Is "+x);
    }
}
Inner i1=new Inner();
i1.show();
i1.display();
}
public static void main(String args[])
{
    Local1 l1=new Local1();
    l1.go(10,2);
}
}
```

Output of the program:

```
Name Is Sai
Age Is 10
Roll Number Is 1
Address Is Cuttack
Value of A Is 10
Value of X Is 12
```

Anonymous Inner Class

One can also declare an **inner class** within the body of a method without naming it. Such classes are known as **anonymous inner classes**. In other words, the declaration and initialization of the class is done on the same line. Have a look at the following program:

The main advantage of anonymous inner class is we create the object of abstract class and interface. As a programmer we know that we never instantiated abstract class and interface.

Program 2.18

```
public abstract class Test
{
    public abstract void show();
    void fun()
```

```
{
    System.out.println("Good Morning");
}
public static void main (String args[])
{
    Test tt=new Test ()
    {
        public void show ()
        {
            System.out.println ("Hi Everybody");
        }
    };
    tt.fun ();
    tt.show();
}
}
```

Output of the program:

Good Morning

Hi Everybody

Here we know abstract class cannot be instantiated but within the `main ()` method we construct abstract class through anonymous inner class.

2.2.9 String Class

The `String` class is more commonly used to display messages and when strings need to be compared, searched or individual characters in a string have to be extracted as a substring.

The syntax to declare `string` is:

```
String string_name;
```

The syntax for creating a `string` is:

```
string_name=new String("Sequence_of_characters");
```

These two steps of declaration and creation can be combined into a single statement as shown:

```
String string_name=new String("Sequence_of_characters");
```

For example, the statement to declare and create a `string str1` using `String` class is,

```
String str1=new String("Java Programming Language");
```

The `String` class provides various methods for manipulating strings. Some of the most commonly used methods of `String` class along with their descriptions are listed in the Table 2.2.

NOTES

Table 2.2 String Class Methods and their Description

NOTES

Methods	Description
<code>str1.length()</code>	Returns the length of the string <code>str1</code> .
<code>str1.equals(str2)</code>	Returns 'true' if string <code>str1</code> is equal to string <code>str2</code> .
<code>str1.compareTo(str2)</code>	Returns negative if <code>str1 < str2</code> , positive if <code>str1 > str2</code> , otherwise 0.
<code>str1.concat(str2)</code>	Concatenates string <code>str1</code> and string <code>str2</code> .
<code>str1=str2.trim()</code>	Removes all the white spaces at the beginning and end of the string <code>str2</code> and assigns it to <code>str1</code> .
<code>str1=str2.replace('a','b')</code>	Replaces all a appearing in the string <code>str2</code> with b and assigns it to <code>str1</code> .
<code>str1=str2.toLowerCase()</code>	Converts uppercase letters in a string <code>str2</code> to lowercase and assigns it to <code>str1</code> .
<code>str1=str2.toUpperCase()</code>	Converts lowercase letters in a string <code>str2</code> to uppercase and assigns it to <code>str1</code> .
<code>str1.indexOf('a')</code>	Gives the position of the first occurrence of character 'a' in the string <code>str1</code> .
<code>str1.indexOf('a',n)</code>	Gives the position of the first occurrence of character 'a' that occurs after n th position in the string <code>str1</code> .

Program 2.18: A program to demonstrate the use of some of the methods of a String class

```
class StringDemonstrate
{
    public static void main (String args[])
    {
        String str1=new String("New"); //creating string str1
        String str2=new String("Delhi"); //creating string
        str2
        String str3=str1.concat (str2); //concatenating strings
        str1 and str2
        String str4=str3.toUpperCase ();
        String str5=str3.toLowerCase ();

        System.out.println("Combined String is: " +str3);
        System.out.println("Combined String in UPPER CASE
is: " +str4);
        System.out.println("Combined String in LOWER Case
is: " +str5);
    }
}
```

Output of the program:

```
Combined String is: NewDelhi
Combined String in UPPER CASE is: NEWDELHI
Combined String in LOWER Case is: newdelhi
```

2.2.10 Command Line Arguments

A command line argument is the information that directly follows the program's name on the command line when it is executed. They are stored in String array passed to the **args** parameter `main ()`. The first command-line argument is stored at **args [0]**, the second at **args [1]**, and so on.

The following program displays the command line arguments:

```
class CommandLine
{
    public static void main (String args[ ])
    {
        for (int a=0; a<args.length; a++)
            System.out.println ("args[" + a + " ]: " + args[a]);
    }
}
```

Try executing this program as follows:

```
Java CommandLine This is a test 10 -1
```

The output of the program:

```
args [0] : This
args [1] : is
args [2] : a
args [3] : test
args [4] : 10
args [5] : -1
```

NOTES

2.3 INHERITANCE

Inheritance is an important concept in Java. It facilitates code reusability. It is one of the corner-stones of object-oriented programming principles. The philosophy of inheritance is inculcated from the life cycle of a creature. Just like a child acquires some of the characteristics of a parent, the child class inherits codes that include variables, and methods from the parent class. A programmer needs to know the basics of inheritance in order to have an object-oriented model of business logic. Inheritance is one of the fundamental concepts of object-oriented programming. Using inheritance, one can create a general class that defines traits common to a set of related items. This class can then be inherited by other more specific classes, each adding those things that are unique to it. In the terminology of Java, the class that is inherited is called *super class* and the class that inherits the properties of super class is known as *derived class* or *child class*. Java uses the *extends* keyword to support inheritance.

Getting into the Concept

The general syntax of inheritance is

```
class X
{
    //Codes
}
class Y extends X
{
    //Codes
}
class baseClass extends superClass{
//Codes
} should be the syntax.
```

Here X is the super class and Y is the child class or derived class.

NOTES

Benefit of Inheritance

One of the benefits of inheritance is code reusability because the derived class (child class) copies the member of the super class (parent class).

Restriction: However, if the super class members are private, then the child class cannot copy them. Another thing to be remembered is that Java class does not support multiple inheritances.

To inherit a class, the definition of one class has to be incorporated into another by using the extends keyword. This can be seen in the following example.

Codes to Show Inheritance

Example 2.5

```
class X{
    void show()
    {
        System.out.println("Hello, Java I am inherited");
    }
}
class Y extends X
{
}
class Demo
{
    public static void main(String args[])
    {
        Y a=new Y();
        a.show();
    }
}
```

Output of the program:

```
Hello, Java I am inherited
```

In the above example, X is known as the super class and Y is the derived class or the child class. Inside the main() method, the object of child class is created but the show() method of parent class is invoked. The show() method is not a member of Y. In case of inheritance, the child class copies those members of the parent class that are not private, to itself. show() method belongs to class X. It has a default access specifier. Therefore, it can be inherited within a package. By using extends keyword, show() method is copied to class Y. Hence, any object of class Y can invoke show() method.

Multi-Level Inheritance

This can be shown by the following example:

Example 2.6

```
class X
{
    void show()
    {
        System.out.println("Hello, Java");
    }
}
```



```
    }  
}  
class Y extends X  
{  
}  
class Z extends Y  
{  
    Z()  
    {  
        super.show();  
    }  
}  
public class Demo extends Z  
{  
    public static void main(String args[])  
    {  
        Z obj=new Z();  
    }  
}
```

NOTES

Output of the program:

Hello, Java

Here show() method of X is copied to class Y and from class Y to class Z. Hence the object of class Z can invoke show() method which is written in class X. In this way, inheritance facilitates code reusability.

2.3.1 Member Access

Each object of a class has its own set of variables. These variables should be assigned values before being used in the program. The instance variables and methods added in the program cannot be accessed directly outside the class using their names. To access the variables and methods outside the class, the dot (.) operator is used as follows:

```
object_name.variable_name  
object_name.method_name(parameter_list)
```

where,

object_name is the name of the object.

variable_name is the name of the instance variable that is to be accessed.

method_name is the name of the method which is to be called.

parameter_list is the series of pairs of data types and their respective identifiers.

For example, the instance variable length of Cuboid class can be accessed as follows:

```
cobj.length;
```

Similarly, the method volume() of Cuboid class can be accessed as follows:

```
cobj.volume();
```

NOTES

Program 1.19: A program to demonstrate the accessing of members of a class.

```
class Cuboid
{
    int length;
    int width;
    int height;
    int volume() //method definition
    {
        return(length*width*height);
    }
}
class ClassDemo
{
    public static void main(String args[])
    {
        Cuboid cobj=new Cuboid(); //object creation
        cobj.length=60;
        cobj.width=20; //accessing variables
        cobj.height=40;
        int vol=cobj.volume(); //calling method
        System.out.println("The volume of the cuboid is: "
+vol);
    }
}
```

Output of the program:

```
The volume of the cuboid is: 48000
```

In this example, the instance variables `length`, `width` and `height` of the object `cobj` are assigned values outside the class using the dot operator. Alternatively, the instance variables can be assigned values by using a parameterized method.

2.3.2 Super Class Variable

In Java, the concept of inheritance is implemented through super class. The super class is used to save the work of an existing class that can inherit the property of general class. It introduces better data analysis, reduces development time and gives fast performance. The Java super class is a type of class that provides methods to Java subclass. Other classes can extend the super class. It allows the extended class to build behaviour and state upon it. For example, `Color` class is the super class of `Car`:

```
public class Car extends Colour
{
}
```

The keyword 'Super' is used to point the super class instance. For example, the following code shows how to use super key:

NOTES

```
class Number
//Class Number is declared
{
    int num = 20;
//Variable num is defined as integer data type
}
class DisplayNum extends Number
{
    public void num_function()
    {
        System.out.println(super.num);
//Prints the value of num variable accessing by super keyword
    }
}
```

In the previous example, the super keyword is used to access the super class variable.

The following code is written in Java using super class:

```
import java.util.ArrayList;
//Importing java.util.ArrayList
import java.util.Vector;
//Importing java.util.Vector
public class Main {
//Constructor
    Public Main()
    {
        checkObjectSuperClass(new Vector());
        checkObjectSuperClass(new ArrayList());

        checkObjectSuperClass("Test String");
        checkObjectSuperClass(new Integer(1));
    }
//Checking which superclass the object has.
    public void checkObjectSuperClass(Object testObject)
    {
        System.out.println("Object has the superclass " +
        testObject.getClass().getSuperclass().getName());
    }
//Starting the main program
    public static void main(String[] args)
    {
        new Main();
    }
}
```

Output of the program:

```
Object has the superclass java.util.AbstractList
Object has the superclass java.util.AbstractList
Object has the superclass java.lang.Object
Object has the superclass java.lang.Number
```

The super class of Vector and ArrayList classes reside in the java.util.AbstractList class. The String class is derived from java.lang.Object class. The Integer class is derived from the java.lang.Number class.

NOTES

Every object has instance variables of super classes, such as parent class, grandparent class, etc. These super classes are initialized before the instance variable of class. The various methods are used for super class as follows:

Method I: Automatic insertion of super class constructor call:

Automatic insertion of super class takes place if created object is necessary to call the constructors, otherwise Java automatically performs this task.

```
public Two_Var(int aa, int bb)
{
    super();
    // Automatically inserted
    a = aa;
    b = bb;
}
```

Method II: Explicit call to super class constructor:

```
class my_Window extends JFrame
{
    . . .
    //Constructor
    public my_Window (String s_title)
    {
        super(s_title);
        . . .
    }
}
```

A parent constructor is called having parameters but the default constructors has basically no parameters.

2.3.3 Subclass Object

The class that is inherited by other classes is called a base class or superclass or parent class. The class that inherits the properties of the superclass is called a subclass or derived class or child class. The subclass inherits all of the instance variables and methods that are defined by the superclass and at the same time it also contains its own members. For example, in Figure 2.1, animal is the superclass which is inherited by three subclasses—carnivore, herbivore and omnivore. Hence, carnivore, herbivore and omnivore inherit all the members of the superclass animal.

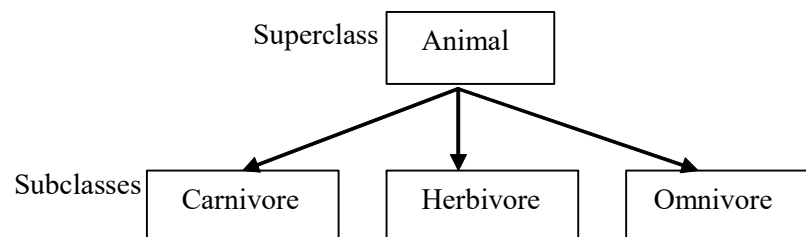


Fig. 2.1 Superclass and Subclass

Defining a Subclass

Inheritance is implemented while defining the subclass. The name of the superclass is specified in the subclass definition. A subclass can be defined by using extends keyword.

The syntax to define a subclass is

```
class sub_class extends super_class
{
    //variables and methods declaration
}
```

where,

sub_class is the name of the subclass that inherits the superclass

super_class is the name of the superclass that is being inherited

extends is the keyword that indicates that the super_class properties have been extended to the sub_class

2.3.4 Using Super to Call Superclass Constructors

In Java, you can use the subclass constructor to initialize instance variables of both the superclass and the subclass.

Example 2.7: A program to explicitly initialize the instance variables of superclass inside the subclass constructor

```
class FirstClass
{
    int x,y;
    int func1()
    {
        return (x+y);
    }
}
class SecondClass extends FirstClass
{
    int z;
    SecondClass(int a, int b, int c) //subclass constructor
    {
        // initializing instance variables of superclass
        x=a;
        y=b;
        // initializing instance variable of subclass
        z=c;
    }
    int func2()
    {
        return (x+y+z);
    }
}
class ExplicitInitialization
{
    public static void main (String args[])
    {
        SecondClass obj=new SecondClass (10,20,30);
        int result1=obj.func1();
        int result2=obj.func2();
        System.out.println("The first result is: "+result1);
        System.out.println("The second result is:
        "+result2);
    }
}
```

NOTES

The output of the program is

```
The first result is: 30  
The second result is: 60
```

NOTES

In this example, the subclass `SecondClass` explicitly initializes the instance variables `x` and `y` of the superclass. However, if instance variables of the superclass are declared as `private`, then subclass cannot access or initialize these instance variables of superclass. For this, Java provides a keyword `super` that can be used by the subclass to pass values to the private variables of superclass by calling its constructor.

The syntax for using `super` keyword is

```
super(parameter_list);
```

Here, `parameter_list` specifies the parameters required by the superclass constructor.

These are certain points that should be kept in mind while using `super` keyword. These points are as follows:

- It is used only within a subclass constructor.
- It should be the first statement to be executed inside the subclass constructor.
- The parameters specified in the `super()` method must match the order and type of the variables declared in the superclass' constructor.

2.3.5 Multilevel Hierarchy

You have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain many layers of inheritance. To see how a multilevel hierarchy can be useful, consider the following program.

```
class Worker  
{  
    int salary;  
    float overtime;  
    double total;  
  
    Worker() {}  
  
    public Worker(int x, float y)  
    {  
        salary = x;  
        overtime = y;  
    }  
    public double totalSalary()  
    {  
        return total = salary + overtime;  
    }  
    public double netSalary()  
    {  
        return totalSalary() - totalSalary() * 30 / 100;  
    }  
}  
public class Supervisor extends Worker  
int supervisory_allowance;
```

```
public Supervisor(int x, float y, int z)
{
    salary=x;
    overtime=y;
    supervisory_allowance = z;
}
public static void main (String args[ ] )
{
    Worker shopfloor =new worker (2500,376.5f);
    Worker frontoffice=new worker(2200,85.00f);
    Supervisor gmoffice=new supervisor(5700,459.25f,1876);
    Supervisor planningdept=newsupervisor(6780.145.85f,2567);

    double d;
    d=shopfloor.totalSalary();
    System.out.println("Total Salary of shopfloor worker is Rs." +d);
    System.out.println("Total Salary of frontoffice worker is Rs"+
    frontoffice.totalSalary());
    d=gmoffice.totalSalary();
    System.out.println("Total Salary of GM office supervisor is Rs"
    + d);
    System.out.println("Total Salary of Planning department supervisor
    is Rs." + planning dept.totalSalary());
    System.out.println("Supervisory allowance of gmoffice is Rs."
    +gmoffice.supervisory_allowance);
    System.out.println("Supervisory allowance of planningdept is
    Rs." +
    planningdept.supervisory_allowance);
}
```

NOTES

Output of the program:

```
Total Salary of shopfloor worker is Rs. 2876.5
Total Salary of frontoffice worker is Rs. 2285.0
Total Salary of GM office supervisor is Rs. 6159.25
Total Salary of Planning department supervisor is Rs. 6925.85009765625
Supervisor allowance of gmoffice is Rs. 1876
Supervisor allowance of planningdept is Rs. 2567
```

2.3.6 Calling Constructor

In case of **inheritance** the **super** class default constructor is implicitly invoked when the programmer creates the child class object by calling the child class constructor through a new operator. This can be shown by the following example:

Program 2.20

```
class X
{
    X()
    {
        System.out.println("Inside super class default
        constructor");
    }
    X(int i)
    {
```

```
System.out.println("Inside super class parameterized  
constructor");  
}
```

NOTES

```
}  
class Y extends X  
{  
}  
  
class Demo  
{  
public static void main(String args[])  
{  
Y d=new Y();  
}  
}
```

Output of the program:

Inside super class default constructor

If the **super** class parameterized constructor is defined but the **super** class default constructor is not defined, then compile time error arises. The following example program shows this.

Program 2.21

```
class X  
{  
X(int i)  
{  
System.out.println("Inside super class parameterized  
constructor");  
}  
}  
class Y extends X  
{  
}  
class Demo  
{  
public static void main(String args[])  
{  
Y d=new Y();  
}  
}
```

Output of the program:

The compile time error will arise as follows:

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Implicit super constructor X() is undefined for default constructor. Must define an explicit constructor


```
at demo.Y.<init>(Demo.java:12)
at demo.Demo.main(Demo.java:20)
```

*Class, Inheritance,
Interfaces, Packages and
Exception Handling*

If the child class parameterized constructor is called, the parent class default constructor is also called at that time. This can be shown by the given program.

Program 2.22

```
class X
{
    X()
    {
        System.out.println("Inside super class default
        constructor");
    }
    X(int i)
    {
        System.out.println("Inside super class parameterized
        constructor");
    }
}
class Y extends X
{
    Y(int i)
    {
        System.out.println("Inside child class parameterized
        constructor");
    }
}
class Demo
{
    {
        public static void main(String args[])
        {
            Y d=new Y(6);
        }
    }
}
```

Output of the program:

```
Inside super class default constructor
Inside child class parameterized constructor
```

If the super class default constructor is not defined, then inside child class constructor calls the parent class parameterized constructor as shown in the following example.

Program 2.23

```
class X
{
    X(int i)
    {
```

NOTES

NOTES

```
        System.out.println("Inside super class parameterized
constructor");
    }
}
class Y extends X
{
    Y(int i)
    {
        super(8);
        System.out.println("Inside child class parameterized
constructor");
    }
}
class Demo
{
public static void main(String args[])
    {
        Y d=new Y(6);
    }
}
```

Output of the program:

```
Inside super class parameterized constructor
Inside child class parameterized constructor
```

Here, parent class parameterized constructor is called through the **super** keyword and it is the first statement inside the child class constructor.

2.3.7 Overriding Methods

If a subclass method has the same name, same parameter list and same return type as a superclass method, then we say that the method in the subclass overrides the method in the superclass. When the overridden method is called, the version of the method defined in the subclass will be invoked instead of the method defined in the superclass. That is, the method in the subclass will hide the method defined in the superclass.

Example 2.8: A program to demonstrate method overriding is as follows:

```
class Person
{
String name="John";
int age=30;
void result ()
{
System.out.println("The name and the age of the person are: "+name+"
and "+age+ "respectively ");
}
}
class Employee extends Person
{
int salary=40000;
```

```
        void result()           //result() is overridden
        {
        System.out.println("The name and the salary of the employee are:
        "+name+" and "+salary+ "respectively ");
        }
    }
    public class MethodOverriding
    {
    public static void main(String[] args)
    {
    Employee obj=new Employee();
    obj.result();           // invokes Employee's result()
    }
    }
```

NOTES

The output of the program is:

The name and the salary of the employee are: John and 40000, respectively

In this example, when the method `result()` is invoked through an object `obj` of type `Employee`, the method `result()` defined within `Employee` is executed. That is, the method `result()` of the subclass `Employee` overrides the method `result()` of the superclass `Person`.

2.3.8 Abstract Classes Method

As discussed earlier, a method can be prevented from being overridden in the subclass by using the `final` keyword. However, sometimes there might be a situation, when the method in the superclass always needs to be redefined in the subclass and for this overriding becomes necessary. This kind of situation can arise when the superclass is not able to provide meaningful implementation of its methods. In such a case, the superclass provides a generalized structure of the method and leaves the implementation part to its subclass. To deal with this type of a situation, Java allows us to specify that a method must always be overridden in the subclass by using the keyword `abstract` in the method declaration.

The syntax for declaring an abstract method is as follows:

```
abstract data_type method_name();
```

If a class has one or more abstract methods, the class must also be declared as `abstract` by using the keyword `abstract`.

The syntax for declaring an abstract class is as follows:

```
abstract class class_name
{
:
abstract data_type method_name();
}
```

The class which inherits the abstract class must provide implementation for all methods or the class should declare itself as `abstract`.

The following points should be kept in mind while using an abstract class:

- An abstract class cannot be instantiated.
- Abstract methods of an abstract class must always be implemented in the subclass.

NOTES

- Abstract methods must end with semicolon (;) because they do not have any functionality.
- There can be both defined and undefined methods in an abstract class.
- Constructors or static methods cannot be declared as abstract.

2.3.9 Final Class in Inheritance

When a class is declared as `final` then the class cannot be inherited. A `final` class cannot be declared through the `abstract` keyword.

```
final class X
{
} /*X is a final class. No class cannot be inherited from the final
class. In java String, StringBuffer, Math, Array, All wrapper classes
are treated as final class.*/
class Y extends from X
{
}
//generates compile time error.
```

What is final Method?

`final` method cannot be overridden. When we define a `final` method we never use `abstract` keyword.

What is final Variable?

As a programmer we declare local variable, `static` variable and instance variable as `final`. But when we declare `static` variable as `final` then in Java it is treated as constant. As it is a `final` variable it must be initialized.

```
final static int i=10;//it's a constant
```

In Java we never declare a constant by using `const` keyword. We are bound to declare a constant by using two modifiers `static` and `final`. If an instance variable is declared through `final` keyword then the programmer is bound to initialize it. `final` variable is bound to be initialized.

The Volatile Keyword

`volatile` is a keyword or modifier used for variable. In Java `volatile` variable is used in multithreaded application. When multiple numbers of threads use the same variable then the thread will have its own copy of the local cache for that variable. When the thread updates the value for the variable, the updated value is stored in local cache but not in the main memory.

Restriction

`volatile` variable is never used in `final` modifier.

```
volatile int i;
```

Here `i` is a `volatile` variable.

Check Your Progress

1. Give the definition of class.
2. Define the term method overloading.
3. Write the definition of the term return data type.
4. What is recursion?
5. Write the advantage of inner classes.
6. State about the string class.
7. Write the one benefit of inheritance.
8. What do you understand by the term super class.
9. How inheritance is defined in a sub class?
10. When the overriding method is called in a sub class?
11. What happens when a class is declared as `final`?

NOTES

2.4 INTERFACE

An interface is just like a class. The only difference is that it contains only final variables and method declarations. Hence, we can think of an interface as a 'fully abstract class'. There is no limitation to the number of interfaces that a class can implement.

Defining an Interface

An interface is defined just like a class but rather than using the keyword `class`, the keyword `interface` is used.

The syntax to define an interface is as follows:

```
interface interface_name
{
    //variables and methods declaration
}
```

where,

`interface` is the Java keyword.

`interface_name` is the name of the interface.

If there is no access specifier included in the interface definition, then the default access is used and the interface is visible only to members of the same package. However, to make the interface accessible in any other code, it can be declared as `public`. The variables in an interface are by default `static` and `final`. Hence, they cannot be altered by the implementing class. The methods are `abstract` by default. All methods must be implemented by the class which implements the interface. For example, consider the following code segment:

```
interface Area
{
    double pi=3.142;
    void compute();
}
```

NOTES

Here, `Area` is the name of the interface. The variable `pi` is initialized with a constant value. Note that the method `compute ()` does not have the body part and its declaration ends with a semicolon.

Note: If the interface is declared as `public` then, all variables and methods are implicitly `public`.

Implementing an Interface

Once an interface is defined, it can be used as a superclass whose members and properties can be inherited by other classes. One or more classes can implement the interface by using the keyword `implements` in the class definition.

The syntax for implementing an interface is as follows:

```
class class_name implements interface_name
{
    //variables and methods declarartion
}
```

For example, consider the following code segment which implements the interface `Area`:

```
class Circle implements Area
{
    float r=4.3F;
    public void compute ()
    {
        double carea=pi*r*r;
        System.out.println("The area of circle is: " +carea);
    }
}
```

When the methods in an interface are defined in the implementing class, the `public` keyword must be used. Also, the signature of the method implementing the interface must exactly match the signature of the method declaration in the interface.

A class can implement more than one interface as shown below.

```
class class_name implements interfaced1, interface2
{
    :
}
```

A class can extend another class while implementing interfaces as shown below.

```
class class_name extends superclass implements interface_name
{
    :
}
```

Partial Implementations

If a class that implements the interface does not provide complete implementation of the methods declared in the interface, then it is necessary for the class to be declared as `abstract`. For example, consider another class, `Square` which implements the interface, `Area`.

```
abstract class Square implements Area
{
```

```
float side=2.4;  
double sqarea=side*side;  
void display()  
{  
    System.out.println("The area of square is: "+sqarea);  
}  
}
```

Here, the class `Square` is declared as `abstract` as it does not implement the method `compute()` declared in `Area`. Any class that inherits `Square` must implement `compute()` method or the class itself must be declared as `abstract`.

Extending Interfaces

An interface can inherit another interface by using `extends` keyword in the same way as a class inherits from another class. Like a class, a subinterface will inherit all the properties of the superinterface and also adds its own data members. For example, consider the following code segment.

```
interface Interface1  
{  
    :  
}  
interface Interface2 extends Interface1  
{  
    :  
}
```

An interface can also inherit from more than one interface. To define an interface that extends several interfaces, the names of superinterfaces are separated by commas (,) as shown here:

```
interface Interface3 extends Interface1, Interface2  
{  
    :  
}
```

Note that the methods declared in the superinterfaces cannot be implemented by the subinterfaces. They must be implemented only by the class which implements the interface. When a class implements an interface which is inherited from another interface then the class must provide implementation for all the methods declared in both the interfaces.

Note: An interface cannot extend classes. It can only extend another interface. Also, an interface cannot implement another interface.

After being familiar with the concept of interface, we look at how interface can be used to implement multiple inheritance by looking at a simple example. In this example, the class `Faculty` extends a class `Employee` and implements an interface `Bonus`.

Example 2.9: A program to demonstrate implementation of multiple inheritance through interface is as follows:

```
class Person  
{  
    String name;  
    int age;
```

NOTES

NOTES

```
String address;
void persondetails (String nm, int ag, String add)
{
    name=nm;
    age=ag;
    address=add;
}
void displayperson ()
{
    System.out.println("Name: "+name);
    System.out.println("Age: "+age);
    System.out.println("Address: "+address);
}
}
class Employee extends Person
{
    int empid;
    int salary;
    void empdetails (int id, int sal)
    {
empid=id;
        salary=sal;
    }
    void displayemployee ()
    {
        System.out.println("Empid: "+empid);
        System.out.println("Salary: "+salary);
    }
}
interface Bonus
{
    int bonus=1000;
    void compute ();
}
class Faculty extends Employee implements Bonus
{
    int amount;
    public void compute ()
    {
        System.out.println("The bonus is: "+bonus);
        amount=salary+bonus;
    }
    void facultydetails ()
    {
        displayperson ();
        displayemployee ();
        compute ();
        System.out.println("The total amount is: "+amount);
    }
}
}
public class MultipleInheritance
{
    public static void main (String[] args)
    {
        Faculty obj=new Faculty ();
    }
}
```



```

obj.persondetails("Surabhi", 23, "115, Greenfield Apartment.
Patparganj, New Delhi-110092");
    obj.empdetails(001, 20000);
    obj.facultydetails();

    System.out.println("");

obj.persondetails("Mili", 27, "D-50, Old Gupta Colony, Delhi-110009");

obj.empdetails(002, 30000);
    obj.facultydetails();
}
}

```

*Class, Inheritance,
Interfaces, Packages and
Exception Handling*

NOTES

The output of the program is:

```

Name: Surabhi
Age: 23
Address: 115, Greenfield Apartment, Patparganj, New Delhi-110092
Empid: 1
Salary: 20000
The bonus is: 1000
The total amount is: 21000

Name: Mili
Age: 27
Address: D-50, Old Gupta Colony, Delhi-110009
Empid: 2
Salary: 30000
The bonus is: 1000
The total amount is: 31000

```

2.5 PACKAGES

A software development, a task is divided into different modules and then each module is developed by different programmers. After that, all the modules are integrated together. If the software is developed in Java, then each module is definitely a class or combination of classes. A collection of such classes is called a **package**. From the point of view of software development, a **package** is quite important. A Java library consists of various packages. **Package** is a container and consists of classes and interfaces grouped together according to functionality. It contains a set of classes in order to ensure that the class names are unique. The classes and interfaces are in a hierarchical order and the **packages** are imported when the programmer wants to access classes or interfaces within it.

Advantages

The following are advantages of packages:

- (a) Classes contained in the packages of other programs can be easily reused.
- (b) Two classes in two different packages can have the same name.
- (c) Packages provide a way to hide classes.
- (d) Packages provide a way for separating design from coding.

NOTES

Creating Packages

Classpath Variable

If the programmer has not declared the package name, then the class files are stored by default, in the current working directory included in the classpath variable. At the run-time, the Java interpreter searches the class files in the path specified in the classpath environmental variable. For creating a user-defined package, it must be ensured that the root directory of the package is included in the classpath variable.

A package program follows three steps:

- (a) Package declaration
- (b) Import statement
- (c) Class definition

The format of package declaration is quite simple. The keyword the 'package' is followed by the package name. The package name must be the same as the directory name. When class files are created, they must be placed in a directory hierarchy that reflects the package name.

```
package p1;
```

Import is a keyword in Java that is used to access the class files of the user-defined package and the predefined package.

```
import p1.*;  
import java.util.*;
```

Here '*' is used to import all the classes and interfaces present in the package which are not those of its sub-package.

Naming Convention

- (a) Package begins with lower case letter.
- (b) Every package name must be unique, to the best use of package.
- (c) To ensure uniqueness in naming packages, domain name is used as prefix to the package name.

The following example shows this:

Example 2.10

```
package p2;  
class Test  
{  
    public static void main(String [] args)  
    {  
        System.out.println("hello");  
    }  
}
```

Compiling this program will generate Test.class file. This class file can be kept inside the directory p2. Assume that p2 is present in D drive.

This program can be run using:

Example 2.11

```
D:\>java p2.Test
This will produce the desired output
hello
If p2 is a subdirectory of p1
package p1.p2;
class Test
{
    public static void main (String [] args)
    {
        System.out.println("hello");
    }
}
```

The above code may be compiled to create Test.class file. Keep it inside p2 directory which is inside p1. Run the class file by the following command.

```
D:\>java p1.p2.Test
```

Access Protection

Java has four different types of access specifiers, namely; private, no access, protected, and public. In Java, if the methods and variables are declared without any access specifier, then, it is called ‘No access by default’. Methods and variables are accessed to all the classes within the same package.

Table 2.3 Accessibility of Members Inside and Outside the Package

Members	private	no access	protected	public
Same class	Yes	Yes	Yes	Yes
Within the same package class is inherited	No	Yes	Yes	Yes
Within the same package class is not inherited	No	Yes	Yes	Yes
Outside the package class is inherited	No	No	Yes	Yes
Outside the package class is not inherited	No	No	No	Yes

- (a) If the members are private, then data is only accessed within the class, but it is not accessed outside the class.
- (b) If the members are no access, then data is accessed within the class and accessed within the same package, whether the class is inherited or not.
- (c) If the members are protected, then the data is accessed within the class and accessed within the package, whether the class is inherited or not, but outside the package only when the class is inherited.
- (d) If the members are public, then the data is freely accessed within the package and outside the package, with or without inheritance or without inheritance.

NOTES

NOTES

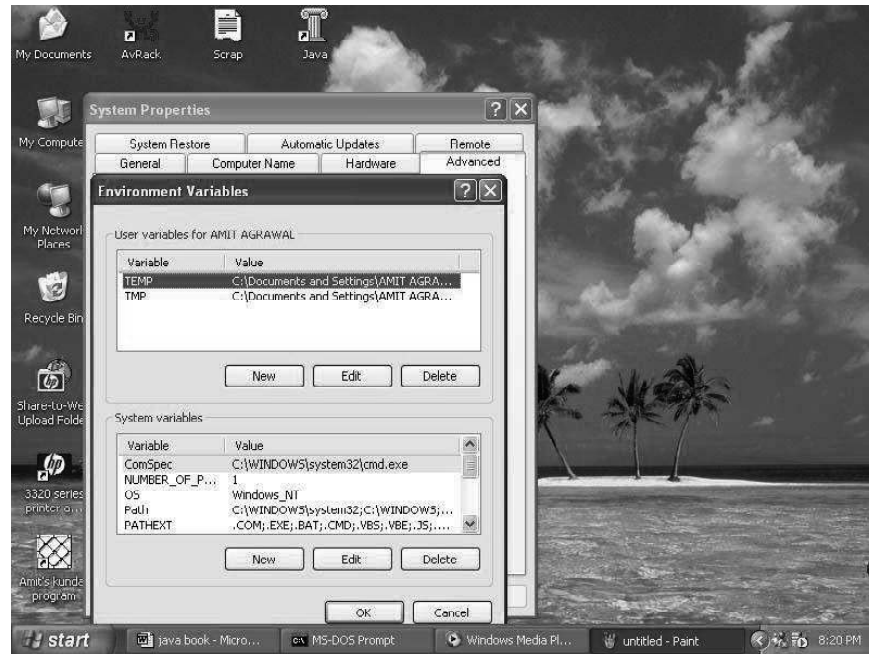


Fig. 2.2 Setting the Classpath in the Environmental Variable for Smooth Operation of the Package Program

Program:

- (a) Create a directory named as pack.
- (b) Open a file Name.Java.

Example 2.12

```
package pack;
class Name
{
    String n;
    String setName (String name)
    {
        n=name;
        return n;
    }
}
```

In Name class, the class, method and variable are declared without using any access specifier. This means that the members are only accessed within the same package pack. Consider the following example:

Example 2.13

In the same package open Roll.Java

```
package pack;
public class Roll
{
    protected int roll;
    protected int getRoll (int r)
    {
        roll=r;
        Name n1=new Name ();
        System.out.println ("Name Is "+n1.setName ("Amit"));
        return roll;
    }
}
```

```
    }  
}
```

As the Roll class members are protected, it is accessed outside the package through inheritance.

- (a) Inside the pack directory, create a subdirectory named subpack.
- (b) Inside the subpack directory, create a file named **Address.Java**.

Consider the following example:

Example 2.14

```
package pack.subpack;  
import pack.Roll;  
public class Address extends Roll  
{  
    public String address;  
    public String getAddress (String a)  
    {  
        address=a;  
        Address a1=new Address ();  
        System.out.println("Roll Number Is "+a1.getRoll (7));  
        return address;  
    }  
}
```

In this class, the members are public; so it can be freely accessed within the package and outside the package, either through inheritance or without inheritance. Here, c:\Java is the root directory.

In the root directory, write the main application PackDemo.Java.

In c:\Java open PackDemo.Java

Consider the following example clarifies this more:

Example 2.15

```
import pack.subpack.Address;  
public class PackDemo  
{  
    public static void main (String args [])  
    {  
        Address a1=new Address ();  
        System.out.println("Address Is "+a1.getAddress  
        ("Nayabazar, Cuttack"));  
    }  
}
```

Output of the program:

```
C:\java>java PackDemo  
Name Is Amit  
Roll Number Is 7  
Address Is Nayabazar, Cuttack
```

Using Packages

A Java package is used in the program after creating it. It can be used within the program or in the Java Application Programming Interface (API). Three steps are required to use the resources in a package. In a package, interfaces along with

NOTES

NOTES

classes are grouped together because they provide similar functionalities. A new name space is created while creating a package. A package will be nested in other packages if classes are defined at the same level of the package. Package works with inline member declarations and also with a single package member. These steps are defined as follows:

Inline Member Declarations

In this step, the package member is declared that is to be used with fully qualified package name. For example, the `Vector` class is used in `java.util` package by defining a vector using `java.util.Vector vector;` statement. The public class `Vector` provides a growable array of objects. The size of `Vector` can increase or decrease if the items are added or removed after creating `Vector`. The code for inline member declaration is written as follows:

```
class vec_test
{
    java.util.Vector vector;
    vec_test()
    {
        vec_test() = new java.util.Vector();
    }
}
```

Importing a Single Package Member

Once the inline declaration is implemented, you need the source code; hence, it is better to use member name wherever it is needed. This mechanism can be achieved easily by `import` keyword followed by fully qualified name of the member to be used. The code for importing single package member is written as follows:

```
import java.util.LinkedList;
import java.util.Vector;
class vector_test
{
    Vector vector;
    vec_test()
    {
        Vector = new Vector ();
    }
}
```

Importing a Package

A number of members from a package can be imported by `import` keyword. So, you can import the entire package if it is needed in the program. The code for importing single package member is written as follows:

```
import java.util.Vector;
//Import Vector class
import java.util.LinkList;
//Import only LinkedList class
import java.util.Hashtable;
//Import only Hashtable class
import java.util.Stack;
//Import stack class
import java.util.Set;
class vec_test
```

```
{  
    ...  
}
```

You can use wildcard (*) symbol to replace all the import statements to import the entire packages. For example,

```
import java.util.*;  
//Import all public classes from java.util package  
class vec_test  
{  
    ...  
}
```

The Java packages can be used to specify the package that you want to use. The following example shows how to use the various java packages 'package' along with declaration:

```
package java.awt.event;
```

If you use package in Java program, the first step required is to import classes from package. For this, you can use 'import' keyword. The statement is written as follows:

```
import java.awt.event.*;
```

The statement written as `java.awt.event` package imports all classes.

```
import java.awt.event.ActionEvent;
```

This statement imports `ActionEvent` class from package.

This `ActionEvent` class is referenced by itself. For example,

```
ActionEvent myEvent = new ActionEvent();
```

The following figure shows the graphical representation of the java package that shows the level of nesting. It contains the sub-packages, the classes in sub-packages and subroutines of the classes.

Figure 2.3 shows that `java.lang.Math` class contains two methods as `sqrt()` for getting the square root value of double data type the specified number value and `random()` method generates the random number. The `jav.awt.Graphics` class contains the two methods as `drawRect()` and `setColor()`. The `drawRect()` draws the outline of rectangle and `drawRect()` sets the graphics context with current colour.

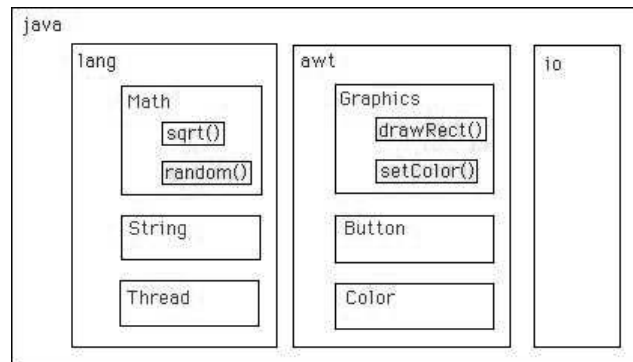


Fig.2.3 Graphical Representation of Java Package Defined in Class

NOTES

NOTES

Interfaces

Interface is a keyword. An Interface declaration is same as class declaration. Interface is designed to support multiple inheritance in Java.

For example:

```
public interface X {  
}  
interface X {  
}
```

The interface is implicitly abstract in nature. It cannot be instantiated. In interface all the methods are public and abstract by default. Therefore, these methods have to be overridden in their corresponding child class. The methods belonging to an interface are looked up at run-time. In interface the variables are implicitly public, static and final. The advantage of declaring variables within an interface is that they become globally available to all the classes and need not be declared explicitly in classes implementing them. If a programmer is not overriding the methods present in the interface then a compilation error will be generated. Have a look at the codes in the following example.

Example 2.16

```
interface P {  
    String fun();  
}  
public class Demo implements P {  
    public static void main(String []args) {  
    }  
}
```

Output of the program:

The above program generates a compilation error. The message that pops up is:
Demo is not abstract and does not override abstract method fun() in P.

When one implements an interface, the methods present in the interface have to be overridden or the class has to be made abstract.

The following points are to be remembered about an interface:

- (a) An interface can only extend one interface.
- (b) It cannot extend a class.
- (c) A class can implement more than one interface and extend only one class.
- (d) An interface cannot implement an interface. Have a look at the following example:

Example 2.17

```
interface p {  
    String fun();  
}  
public class demo implements p {  
    public static void main(String []args) {  
        demo d=new demo();  
        System.out.println(d.fun());  
    }  
}
```



```
public String fun ()  
{  
    return "Hello Interface";  
}  
}
```

Output of the program:

Hello Interface

Here `p` is an `interface` containing an abstract method `fun ()` which returns a string object. This `interface` is implemented in class `demo`. Hence, the `fun ()` method is overridden there. Once the method is overridden, it is possible to create the object of `demo` class and invoke the `fun ()` method. Consider the following example:

Example 2.18

```
interface p {  
    String fun ();  
}  
public class demo implements p {  
    public static void main (String [] args) {  
        p obj=new demo ();  
        System.out.println(obj.fun ());  
    }  
    public String fun ()  
    {  
        return "Hello Interface";  
    }  
}
```

Output of the program:

Hello Interface

In this program consider the statement
`p obj=new demo ();`

Here, `obj` is a reference of the `interface p` and `p` holds the object of `demo` class. This is because the parent class reference can hold the object of child class.

2.6 FUNDAMENTALS OF EXCEPTION HANDLING

An exception is an error that can occur in the course of execution of a program, which may put a stop it from continuing. If exceptions are not handled properly, the program will meet with an abrupt end and an error message will come up.

Exception Handling

An exception signifies an illegal, invalid or unexpected issue during a program. Since exceptions are almost always assumed to be anticipated, you need to provide an appropriate exception handling. Exception handling means diverting the processing to a part of the program when an exception occurs. This part of the program will try to cope with the error or at least will allow the program to die peacefully. The error can be anything like ‘unable to open the file’, ‘array subscript

NOTES

NOTES

out of range', 'no memory left to allocate', 'division by zero', etc. When an exception occurs, the Java run time system creates an object.

Keywords Frequently Used in Exception Handling

The important keywords of exception handling are **try**, **catch**, **throw**, **throws** and **finally**. They are not methods, but termed as **try block** and **catch block** and in these blocks the handling codes are written.

As usual, each block contains statements delimited by braces ({ }). Statements that are suspected to raise the exceptions (like $c = a/b$) are written in try block and the statements to handle the situation when the try block statements raise an exception are written in catch block (like catch ArithmeticException).

The general form of an exception-handling block:

```
try
{
    // block of code
}

catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
//...
finally
{
    // block of code
}
```

For example:

```
public class ArrayIndex
{
    public static void main(String args[ ])
    {
        int marks[ ] = { 10, 20, 30, 40 ,50};
        try
        {
            System.out.println ( marks[10] );
        }
        catch (ArrayIndexOutOfBoundsException ai )
        {

            System.out.println("Hello! Exception is caught by me" +
ai);

        }
        finally {
            System.out.println("This executes irrespective of raising
an exception");
        }
    }
}
```

```
        System.out.println("marks[4] =" + marks[4] );  
    }  
}
```

Output of the program:

```
Hello! Exception is caught by me java.lang.  
ArrayIndexOutOfBoundsException : 10  
This executes irrespective of raising of an exception  
marks [4] = 50
```

Every try block must be followed by at least one catch block. ‘Finally’ block is optional and if present, its execution is guaranteed whether exception is raised or not. The ‘finally’ block is useful when ‘Cleanup’ must be performed after the related try. Operations such as closing files and releasing resources are often performed in a ‘finally’.

Triggering a Predefined Exception

A predefined exception is generated when a program does some illegal operation, e.g., division by zero.

A Java program which will trigger an exception is as follows:

```
public class Example 2.19  
{  
    public static void main (String argv[])  
    {  
        int i=1, j=0, k;  
        k=i/j;  
        System.out.println("Hello World");  
    }  
}
```

Compiling and running the above program will give the following result:

```
java.lang.ArithmeticExceptions: / by zero
```

In this example, division by zero is an illegal operation and it caused an exception and the program came to an abrupt halt with an error message.

Handling the Exception

The exception thrown in the earlier program can be captured and handled properly. The Java program that shows how to do it is as follows:

```
public class Example 2.20  
{  
    public static void main (String argv[])  
    {  
        int i=1, j=0, k;  
        try  
        {  
            k=i/j;  
        } catch (ArithmeticException ae)  
        {  
            System.out.println("Divison by zero, illegal  
operation");  
        }  
        System.out.ptintln("Hello World");  
    }  
}
```

NOTES

NOTES

Output of the program:

```
Division by zero, illegal operation  
Hello World
```

Here the exception created does not end the program. The program runs to complete itself as is evident from the second line **Hello World** of the output.

Throwing an exception is equivalent to a break statement. The statement below the point where exception occurred in the try-catch segment will not be executed. A java program to illustrate that an exception is equivalent to a break statement is as follows:

```
public class Example 2.21  
{  
    public static void main (String argv[])  
    {  
        int i=1, j=0, k;  
        try  
        {  
            k=i/j;  
            System.out.println("Welcome");  
        }  
        catch (ArithmeticException ae)  
        {  
            System.out.println("Division by zero, illegal  
operation");  
        }  
        System.out.println("Hello World");  
    }  
}
```

Output of the program:

```
Division by zero, illegal operation  
Hello World
```

Notice that this program does not print a line **Welcome**. This indicates that the statement below the statement $k=i/j$; i.e., $1/0$, where the exception occurred is not executed.

Try-Catch Construct

The try-catch construct is the technique used to capture and handle exceptions. The statements between try and catch will be executed.

While executing, if an exception occurs which matches with the argument of catch, the statements in the curly braces that follow the catch keyword will be executed. After executing these statements, the program will continue with the next statement.

If the exception created does not match with the argument of catch, what will happen? This program gives the answer.

A Java program in which the exception created does not match with the argument of catch:

```
public class Example 2.22  
{  
    public static void main (String argv[])  
    {  
        int i=1, j=0, k;
```

```
try
{
    k=i/j;
}
catch(ClassCastException cce)
{
    System.out.println("Division by zero, illegal
operation");
}
System.out.println("Hello World");
}
```

Output of the program:

```
Java.lang.ArithmeticException : / by zero
```

This output is exactly the same as program Example 2.19.

The group of statements between the try-catch clauses may be generating several exceptions. Because the first catch clause that matches is executed, you can build chains of catch clauses.

```
try
{
    _____
    _____
}
catch (NullPointerException npe)
{
    _____
    _____
}
catch (RuntimeException re)
{
    _____
    _____
}
catch (IOException ioe)
{
    _____
    _____
}
catch (Exception e)
{
    _____
    _____
}
catch (Throwable t)
{
    _____
    _____
}
```

User Created Exceptions

Exceptions can be set in motion explicitly by the user with the throw statement. The throw statement has the following format:

```
throws ExceptionObject
```

NOTES

NOTES

The ExceptionObject is an object of the class that extends Exception class.

For example:

```
class OutofRange extends Exception
{
    _____
    _____
}
_____
_____
int i=10;
try
{
    if(i<12) throw new OutofRange ();
}
catch (OutofRange oor)
{
    _____
    _____
}
```

Any method that throws a user-defined exception must also catch the exception. This program shows an example.

A Java program to illustrate user created exceptions and its handling:

```
class Out ofRange extends Exception
{
    OutofRange (String ss)
    {
        super (ss);
    }
}
public class Example 2.23
{
    public static void main (String argv[])
    {
        int i=10;
        try
        {
            if (i<12) throw new OutofRange ("10 is the limit");
        }
        catch (OutofRange o)
        {
            System.out.println ("Error:" +o.getMessage ());
        }
    }
}
```

Output of the program:

```
Error: 10 is the limit
```

Handling Related Exceptions

Exception thrown should match with the argument exception of the catch clause. Exception thrown can be a subclass of the ArgumentException. It must be clear that ArithmeticException is derived from RuntimeException. The program can be written as follows:

```
public class Example 2.24
{
    public static void main (String argv[])
    {
        int i=1, j=0, k;
        try
        {
            k=i/j;
        }
        catch(RuntimeException re)
        {
            if (re instanceof ArithmeticException)
            {
                System.out.println("Divison by zero is illegal");
            }
        }
        System.out.ptintln("Hello World");
    }
}
```

Output of the program:

```
Division by zero is illegal
Hello World
```

Handling Group of Related Exceptions

A group of exception objects, all derived from the same exception class, can be caught and assigned to a single class which is the same as the parent class.

For example:

```
class Exception0 extends Exception
{
    _____
    _____
}
class Exception1 extends Exception0
{
    _____
    _____
}
class Exception2 extends Exception0
{
    _____
    _____
}
class Exception3 extends Exception0
{
    _____
    _____
}

_____
_____
int i=2;
try
{
    if (i==1) throw new Exception1 ();
```

NOTES

NOTES

```
        if(i==2) throw new Exception2();
        if(i==3) throw new Exception3();
    }
    catch(Exception0 e)
    {
        if(e instanceof Exception1)
        {
            _____
            _____
        }
        if(e instanceof Exception2)
        {
            _____
            _____
        }
        if(e instanceof Exception3)
        {
            _____
            _____
        }
    }
}
```

Exception Propagation

An exception should be handled in the method in which it is thrown. In case it is not handled in the method in which it was thrown, the method's signature should be modified so that the caller of this method is forewarned about the exception. The signature of the method is modified as shown as follows:

```
public void method() throws ExceptionObject
{
    _____
    _____
}
```

In this case, the caller of the method should do the exception handling. It propagates like this till the top most level. If it is not handled even at the top most level, the program will abruptly end with an error message.

A Java program in which exception is not handled:

```
public class Example 2.25
{
    public static void main(String argv[])
    {
        System.out.println("Inside main().calling method1()");
        method1();
        System.out.println("End of the program");
    }
    public static void method1()
    {
        System.out.println("Inside method1().calling
method2()");
        method2();
        System.out.println("Returning from method1()");
    }
    public static void method2()
    {
```



```
        System.out.println("Inside method2().calling  
method3()");  
        Method3();  
        System.out.println("Returning from method2()");  
    }  
    public static void method3()  
    {  
        System.out.println("Inside method3().executing  
method3()");  
        int i=1,j=0,k;  
        k=i/j;  
        System.out.println("Returning from method3()")  
    }  
}
```

NOTES

Output of the program:

```
Inside main().calling method1()  
Inside method1().calling method2()  
Inside method2().calling method3()  
Inside method3().executing method3()  
java.lang.ArithmeticException: / by zero
```

The program does not run to completion. It comes to an abrupt halt with an error message thrown.

The Java program listing in Example 2.25 shows how an exception is handled in the method in which it occurred:

```
public class Example 2.26  
{  
    public static void main(String argv[])  
    {  
        System.out.println("Inside main().calling method1()");  
        method1();  
        System.out.println("End of the program");  
    }  
    public static void method1()  
    {  
        System.out.println("Inside method1().calling  
method2()");  
        method2();  
        System.out.println("Returning from method1()");  
    }  
    public static void method2()  
    {  
        System.out.println("Inside method2().calling  
method3()");  
        method3();  
        System.out.println("Returning from method2()");  
    }  
    public static void method3()  
    {  
        System.out.println("Inside method3().executing  
method3()");  
        int i=1,j=0,k;  
        try  
        {  
            k=i/j;  
        }  
    }  
}
```

NOTES

```
    }  
    catch (ArithmeticException ae)  
    {  
        System.out.println("Division by zero, illegal  
operation");  
    }  
    System.out.println("Returning from method3()")  
    }  
}
```

Output of the program:

```
Inside main().calling method1()  
Inside method1().calling method2()  
Inside method2().calling method3()  
Inside method3().executing method3()  
Division by zero, illegal operation  
Returning form method3()  
Returning form method2()  
Returning form method1()  
End of the program
```

The exception is handled in the method in which it was thrown. The program runs into completion. Every part of the program is done. It does not come to an abrupt halt. No error message is thrown out by the runtime system.

The Java program listing in which the exception is propagated one level up is as follows:

```
public class Example2.27  
{  
    public static void main(String argv[])  
    {  
        System.out.println("Inside main().calling method1()");  
        method1();  
        System.out.println("End of the program");  
    }  
    public static void method1()  
    {  
        System.out.println("Inside method1().calling  
method2()");  
        method2();  
        System.out.println("Returning from method1()");  
    }  
    public static void method2()  
    {  
        System.out.println("Inside method2().calling  
method3()");  
        try  
        {  
            method3();  
        }  
        catch (ArithmeticException ae)  
        {  
            System.out.println("Division by zero, illegal  
operation");  
        }  
  
        System.out.println("Returning from method2()");  
    }  
}
```

```
        public static void method3() throws ArithmeticException
        {
            System.out.println("Inside method3().executing
method3()");
            int i=1,j=0,k;
            k=i/j;
            System.out.println("Returning from method3()");
        }
    }
```

Output of the program:

```
Inside main().calling method1()
Inside method1().calling method2()
Inside method2().calling method3()
Inside method3().executing method3()
Division by zero, illegal operation
Returning from method2()
Returning from method1()
End of the program
```

In this program the exception is propagated one level up. The program runs to completion. However, every part of the program is not executed as you can see from the output. The output of this program is one line ('Returning from method3()') less compared to the output of the program in Example 2.26. The program does not come to an abrupt halt. No error message is thrown by the runtime system.

The Java program listing in which the exception is propagated two levels up is as follows:

```
public class Example2.28
{
    public static void main(String argv[])
    {
        System.out.println("Inside main().calling method1()");
        method1();
        System.out.println("End of the program");
    }
    public static void method1()
    {
        System.out.println("Inside method1().calling
method2()");
        try
        {
            method2();
        }
        catch(ArithmeticException ae)
        {
            System.out.println("Division by zero, illegal
operation");
        }
        System.out.println("Returning from method1()");
    }
    public static void method2() throws ArithmeticException
    {
        System.out.println("Inside method2().calling
method3()");
```

NOTES

NOTES

```
        method3 ();
        System.out.println("Returning from method2 ()");
    }
    public static void method3 () throws ArithmeticException
    {
        System.out.println("Inside method3 ().executing
method3 ()");
        int i=1,j=0,k;
        k=i/j;
        System.out.println("Returning from method3 ()");
    }
}
```

Output of the program:

```
Inside main() .calling method1 ()
Inside method1 () .calling method2 ()
Inside method2 () .calling method3 ()
Inside method3 () .executing method3 ()
Division by zero, illegal operation
Returning from method1 ()
End of the program
```

In this program, the exception is propagated two levels up. The program runs to completion in the sense that it does not come to an abrupt end with an error message thrown. However, every part of the program is not executed. Two lines

```
Returning from method3 ()
Returning from method2 ()
```

which were present in the output of the program listing in Example8 are missing from the current output. This is the penalty you pay for pushing the exception handling to higher levels instead of doing it in the method in which it has occurred.

The exception is propagated to the top most level in the Java program listing as follows:

```
public class Example2.29
{
    public static void main (String argv[])
    {
        System.out.println("Inside main() .calling method1 ()");
    try
        {
            method1 ();
        }
        catch (ArithmeticException ae)
        {
            System.out.println("Division by zero, illegal
operation");
        }
        System.out.println("End of the program");
    }
    public static void method1 () throws ArithmeticException
    {
        System.out.println("Inside method1 () .calling
```

```
method2 ()");
    method2 ();
    System.out.println("Returning from method1 ()");
}
public static void method2 () throws ArithmeticException
{
    System.out.println("Inside method2 ().calling
method3 ()");
    method3 ();
    System.out.println("Returning from method2 ()");
}
public static void method3 () throws ArithmeticException
{
    System.out.println("Inside method3 ().executing
method3 ()");
    int i=1,j=0,k;
    k=i/j;
    System.out.println("Returning from method3 ()");
}
}
```

NOTES

Output of the program:

```
Inside main () .calling method1 ()
Inside method1 () .calling method2 ()
Inside method2 () .calling method3 ()
Inside method3 () .executing method3 ()
Division by zero, illegal operation
End of the program
```

In this program, the exception is propagated to the top most level. When you compare this output with the output of the program listing Example 2.26, you will find that this output is less by three lines.

```
Returning from method3 ()
Returning from method2 ()
Returning from method1 ()
```

This indicates that every part of the program is not executed.

Finally Clause

Suppose there is some action you absolutely must perform, no matter whatever happens while executing a group of statements, the following Java language construct will help you.

```
_____
_____
try
{
    _____
    // a group of statements
} finally
{
    _____
    // absolutely must do it
    _____
}
```

NOTES

The group of statements enclosed between try and finally clauses can create exception. There can be a break statement, a continue statement or a return statement. In any case, the group of statements in the finally clause will be executed.

A Java program illustrating the use of the finally clause is as follows:

```
class UnnamedException extends Exception
{
}
public class Example12.30
{
    public static void main (String argv[])
    {
        int x=1;
        while (true)
        {
            System.out.print ("Who");
            try
            {
                System.out.println ("is");
                if (x= =1) return;
                System.out.println ("that");
                if (x= =2) break;
                System.out.println ("strange");
                if (x= =3) continue;
                System.out.println ("but kindly");
                if (x= =4) throw new UnnamedException ();
                System.out.print ("not at all");
            }
            catch (UnnamedException ue)
            {
            }
            finally
            {
                System.out.println ("amusing man");
            }
            System.out.println ("I would like to meet
the man");
        }
        System.out.println ("Please tell me");
    }
}
```

Output of the program:

1. if (x= =1) Who is an amusing man
2. if (x= =2) Who is that amusing man
Please tell me
3. if (x= =3) Who is that strange amusing man
Who is that strange amusing man

4. if (x= =4) Who is that strange but kindly amusing man
I would like to meet him
Who is that strange but kindly amusing man
I would like to meet him

```

5. if (x= =5)    Who is that strange but kindly not at all amusing
man

                I would like to meet the man
                Who is that strange but kindly not at all
amusing man

                I would like to meet the man
                _____
                _____
    
```

NOTES

2.6.1 Types of Exception

Java provides several built-in classes which define all types of exceptions. These exception classes are arranged in a hierarchy having `Throwable` class on the top (Refer Figure 2.4). That is, the `Throwable` class is the superclass and all the exception classes inherit methods defined by it. Two immediate subclasses of the `Throwable` class are `Exception` class and `Error` class.

- **Exception Class:** It defines those exceptions which are thrown by methods of standard Java class library or methods defined in user's program and can be trapped within the program. That is, the program can reasonably recover from these types of exceptions. This class is also used (inherited) when the users want to create their own exceptions in the application.
- **Error Class:** It defines those exceptions that do not occur frequently and are difficult to be recovered from. For example, a class file is missing or system runs out of memory.

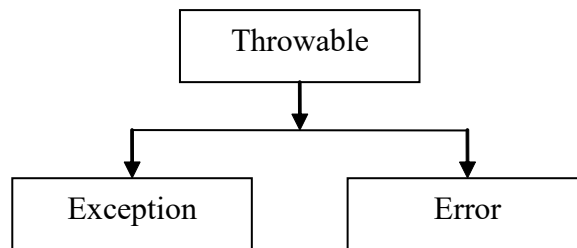


Fig. 2.4 Java Exception Hierarchy

Some of the most commonly used exceptions that will be encountered are listed in Table 2.4.

Table 2.4 Some Common Exceptions in Java

Exception	Description
<code>ArithmeticException</code>	Thrown when an arithmetic error occurs in the program, such as divide-by-zero.
<code>NullPointerException</code>	Thrown when the user tries to use an object without initializing the object or in other words when an object that has not been allocated memory is used.
<code>IOException</code>	Thrown when an error occurs during input/output of data.

NOTES

ArrayIndexOutOfBoundsException	Thrown when an attempt is made to access an array element with invalid index value.
ArrayStoreException	Thrown when an attempt is made to store an incompatible data type in an array.
IllegalAccessException	Thrown when an illegal attempt is made to access a class.
NumberFormatException	Thrown when an invalid conversion of a string to a numeric format takes place.
StringIndexOutOfBoundsException	Thrown when an attempt is made to access a string element that is beyond the index of the string.
IllegalArgumentException	Thrown when an illegal argument is used to invoke a method.
NegativeArraySizeException	Thrown when an array of negative size is created.

2.6.2 Try and Catch Keywords

The default exception handler provided by Java runtime system does not prevent the abrupt termination of the program. To prevent this, Java provides us the facility to construct our own exception handler. By constructing our own exception handler, we can fix the errors ourselves. This can be achieved by enclosing the code that may throw an exception within a `try` block. The `try` block is enclosed by curly braces and preceded by the keyword `try`. Whenever an exception occurs within the `try` block, it is thrown. This passes the control to the `catch` block associated with the `try` block.

The syntax to define `try-catch` block is as follows:

```
try //try block begins
{
    //code that may cause an exception
} //try block ends
catch(exception_type ex)
{
    //code to handle the exception
}
```

If the first statement of the `try` block causes an exception, the remaining statements are not executed and the control passes to the `catch` block. The `catch` statement requires a single argument, which is of the same type as of exception that needs to be handled. This exception type must be a subclass of `Throwable` class. It is not necessary that every time the program is executed, an exception occurs. If an exception is not thrown, the `catch` block is skipped and the control passes to the statement immediately following the `catch` block.

The `try` and `catch` blocks form a logical unit. The scope of the `catch` block is limited only to those statements which are enclosed within the immediately preceding `try` block. Example 2.31 illustrates exception handling using `try` and `catch` blocks.

Note: Compile-time error is generated if the `try` statement is not followed by any `catch` statement.

Example 2.31: A program to illustrate handling of an exception using `try` and `catch` blocks is as follows:

```
class TryCatchBlock
{
    public static void main (String args[])
    {
        int a=15;
        int b=3;
        int c=0;
        try
        {
            System.out.println("try block begins");
            c=a/(a-(5*b)); //exception generated
            System.out.println("try block ends");
        }
        catch (ArithmeticException ae)
        {
            System.out.println("Arithmetic Exception is caught here");
            System.out.println("The resultant value of c is: " +c);
        }
        c=a/b;
        System.out.println("New value of c is: " +c);
    }
}
```

The output of the program is:

```
try block begins
Arithmetic Exception is caught here
The resultant value of c is: 0
New value of c is: 5
```

In Example 2.31, the exception generated within the `try` block is caught inside the `catch` block; thus, preventing the abnormal termination of the program.

Multiple catch Blocks

It is not necessary that the code enclosed within the `try` block throws a single exception. In case, multiple exceptions are thrown within a `try` block, Java allows using multiple `catch` blocks for handling all these exceptions (Refer Example 2.32).

The syntax to define multiple `catch` blocks is as follows:

```
try //try block
{
//code that may cause exceptions
}
catch (exception_type e1) //catch block 1
{
.
.
}
```

NOTES

NOTES

```
catch (exception_type e2)           //catch block 2
{
.
.
}
.
.
.
catch(exception_type en)           //catch block N
{
.
.
}
```

Note that there is only one `try` block from which the exception is thrown and depending on the type of exception thrown, the corresponding `catch` block will be executed. Whenever an exception is thrown, the `catch` blocks are searched in sequential order for an appropriate match. The first `catch` block whose parameter type matches with the type of exception, gets executed and other `catch` blocks are ignored. Once the execution of the appropriate `catch` block gets over, the control passes to the statement immediately following the last `catch` block.

Example 2.32: A program to demonstrate the concept of multiple `catch` statements is as follows:

```
class MultipleCatchExceptions
{
    public static void main (String args[])
    {
        try
        {
            int a=0;
            int b=7/a; //divide by zero exception
            int c[]={1,2,3,4,5};
            c[6]=15; //array out of bound exception
        }
        //this block handles array out of bounds exception
        catch (ArrayIndexOutOfBoundsException aioe)
        {
            System.out.println("Array out of bounds Exception");
        }
        //this block handles arithmetic exception
        catch (ArithmeticException ae)
        {
            System.out.println("Division by zero error");
        }
        catch (Exception e)
        {
            System.out.println("Exception " +e.getMessage());
        }
    }
}
```

The output of the program is:

```
Division by zero error
```

NOTES

In the program illustrated in Example 2.32, the divide by zero error is caught by the second `catch` block containing the instance of `ArithmeticException` class. It should be noted that the exception superclasses must be placed after their subclasses. This is because if we place `catch` block containing the superclass before its subclasses then this superclass will handle all the exceptions of its type as well as of its subclasses. As a result, subsequent `catch` blocks will never get executed and compile-time error is generated. This problem is known as unreachable code problem.

In Example 2.32, if we place `Exception` class before `ArithmeticException` and `ArrayIndexOutOfBoundsException` `Exception` classes, then compiling this program will display the following error message:

```
MultipleCatchExceptions.java:17:exception
java.lang.ArrayIndexOutOfBoundsException has already been caught
    catch (ArrayIndexOutOfBoundsException aioe)
        ^
MultipleCatchExceptions.java:22:exception
java.lang.ArithmeticException has already been caught
    catch (ArithmeticException ae) //this block handles arithmetic
exception
        ^
2 errors
```

This is because `ArrayIndexOutOfBoundsException` class and `ArithmeticException` class are the subclasses of `Exception` class. Thus, `Exception` class catches all the thrown exceptions relative to these classes and their corresponding `catch` blocks are never executed.

Nested `try` Blocks

The `try` blocks can be nested, that is, one `try-catch` block can be placed inside another `try-catch` block. If an exception occurs within a particular `try` block, then the `catch` blocks associated with this `try` block are searched for an appropriate match. If no match is found then the control passes to the next outer `try-catch` block. This process continues until an appropriate match is found. If no match is found, the program terminates abnormally. This is illustrated in Example 2.33.

The syntax of nested `try` block is as follows:

```
try //outer try block
{
    try //inner try block
    {
        .
        .
    }
    catch //inner catch block
    {
        .
        .
    }
}
```

NOTES

```
catch //outer catch block  
{  
.  
.  
}
```

Example 2.33: A program to demonstrate the concept of nested `try` blocks is as follows:

```
class NestedTryBlock  
{  
    public static void main (String args[])  
    {  
        //outer try-catch block  
try  
    {  
        int a[]={0,1};  
        //inner try-catch block  
try  
    {  
        int b[]={0,5};  
        int d=b[1]/b[0]; //exception thrown  
System.out.println("Division of two numbers is :"+d);  
    }  
    catch (ArrayIndexOutOfBoundsException ai)  
    {  
System.out.println("Inside inner try-catch block");  
        System.out.println(ai.getMessage());  
    }  
    }  
    catch (ArithmeticException a)  
    {  
System.out.println("Inside outer try-catch block");//the thrown  
exception is caught here  
        System.out.println(a);  
    }  
    }  
}
```

The output of the program is:

```
Inside outer try-catch block  
java.lang.ArithmeticException: Divided by zero
```

In the program illustrated in Example 2.33, the division by zero error is thrown by the inner `try` block; however, it could not be handled by the inner `catch` block as the type of exception defined by the inner `catch` does not match with the exception thrown by the inner `try`. The matching `catch` block is then searched in a sequential order for an appropriate match and the exception is caught inside the outer `catch` block.

2.6.3 Finally Keyword

It has been observed that when an exception is thrown in the program, the remaining statements in the `try` block are not executed and the control directly gets transferred to the subsequent `catch` block. However, there are certain statements in the program that need to be executed whether or not an exception is raised. For this, Java provides `finally` keyword. The code within the `finally` block will

NOTES

always be executed whether or not the exception is thrown. If an exception is raised with a matching catch block, then the finally block gets executed after the execution of that catch block. On the other hand, if no matching catch block is found then also the finally block is executed after execution of the try block. The finally block is optional; however, it is necessary to include either catch or finally block with try block. Example 2.34 shows the use of finally block.

The diagrammatic representation of working of try-catch-finally block is shown in Figure 2.5.

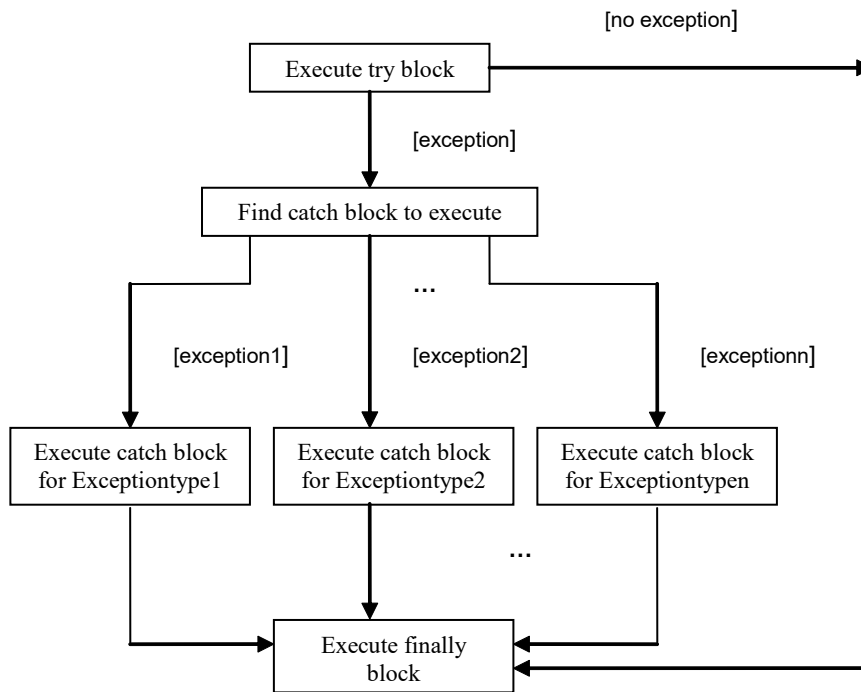


Fig. 2.5 try-catch-finally Block

Example 2.34: A program to illustrate the use of finally block is as follows:

```

class FinallyBlock
{
    public static void main (String args[])
    {
        int a=67;
        int b=0;
        try
        {
            System.out.println("The value of a: " +a);
            System.out.println("The value of b: " +b);
            int c=(a/b); //exception thrown
            System.out.println("Result is: " +c);
        }
        catch (Exception e)
        {
            System.out.println(e); //prints the corresponding exception
        }
        finally
        {
            System.out.println("Denominator cannot be zero");
        }
    }
}
  
```

NOTES

```
    }  
  }  
}
```

The output of the program is:

```
The value of a: 67  
The value of b: 0  
java.lang.ArithmeticException: / by zero  
Denominator cannot be zero
```

Here, the thrown exception is caught in the `catch` block and an appropriate error message is displayed. After that, the statement within the `finally` block is executed.

2.6.4 Throw and Throws

The basic concept and the difference between `throw` and `throws` keywords are discussed below:

Using `throw` Keyword

As stated earlier, Java runtime system automatically throws system-generated exceptions. However, Java provides a mechanism to throw an exception explicitly by using the `throw` keyword.

The syntax of `throw` statement is as follows:

```
throw ExceptionObject;
```

Where, `ExceptionObject` is an object of `Throwable` class or its subclass.

When a `throw` statement is encountered in a program, the execution of the subsequent statements in the `try` block stops and the corresponding `catch` block is searched. The nearest `try` block is checked to determine if it contains a `catch` block to match the exception of its type. If it is found, then that `catch` block is executed; else, subsequent `try` blocks are inspected. In case, if no matching `catch` block is found, then the default exception handler comes into action and stops the normal execution of the program and displays the error message on the output screen.

It should be noted that instances of classes other than `Throwable` class or its subclasses cannot be used as exception objects. The `Throwable` object can be created using a `new` operator or using a parameter inside `catch` clause. Example 3.25 shows how to use a `throw` keyword.

Example 2.35: A program to demonstrate the use of `throw` keyword is as follows:

```
class ThrowExampleDemo  
{  
    public static void main (String args[])  
    {  
        try  
        {  
            ThrowExample (); //invoking ThrowExample () method  
        }  
        catch (ArithmeticException ae)
```

```
        }
        System.out.println("The exception is recaught here: "+ae);
    }
    static void ThrowExample()
    {
        int a=0;
        int b=6;
        try
        {
            int c=b/a;           //system-generated exception
            System.out.println(c);
        }
        catch(ArithmeticException aoe)
        {
            System.out.println("The exception is caught inside the method
            ThrowExample");
            throw aoe;           //exception thrown explicitly
        }
    }
}
```

NOTES

The output of the program is:

```
The exception is caught inside the method ThrowExample
The exception is recaught here :java.lang.ArithmeticException: /
by zero
```

In Example 2.35, the `ArithmeticException` occurs inside the `ThrowExample()`. This exception is caught inside the `catch` block inside the same method, which explicitly rethrows it using the `throw` keyword. This is called rethrowing of the exception. Now, the control passes back to the `catch` block of the `main()` method and the thrown exception is again caught here.

Using throws Keyword

Sometimes, a method may generate an exception, but cannot handle it. That is, there may be a method in the program which is generating (throwing) an exception, but it does not have the appropriate exception handling mechanism. The methods which are calling such methods must be cautioned about this behavior so that calling methods can take appropriate measures to safeguard themselves against the exceptions. This is done by appending `throws` keyword after method name in the method declaration statement. The `throws` clause includes all types of exceptions excluding those belonging to `Error` or `Runtime` classes or their subclasses. All other exceptions which a method may throw must be listed after the `throws` keyword in the method declaration; otherwise compile-time error is generated. Example 2.36 shows how to use `throws` clause.

The syntax of the `throws` clause is as follows:

```
return_type method_name() throws exception_list
{
    //body of the method
}
```

Where, `exception_list` includes all the exceptions that the method might throw.

NOTES

Example 2.36: A program to illustrate the use of throws clause is as follows:

```
class ThrowsExample
{
    //ClassExample method throwing ClassNotFoundException
    static void ClassExample () throws ClassNotFoundException
    {
        System.out.println("Inside ClassExample");
        throw new ClassNotFoundException("This is an example of Class not
        found Exception");
    }
    public static void main (String args[])
    {
        try
        {
            ClassExample ();
        }
        catch (ClassNotFoundException c)
        {
            System.out.println("Exception caught: " +c);
        }
    }
}
```

The output of the program is as follows:

```
Inside ClassExample
Exception caught: java.lang.ClassNotFoundException: This is an
example of Class not found Exception
```

In the program illustrated in Example 2.36, we are explicitly throwing the `ClassNotFoundException` in the method `ClassExample`. Since we are not handling the thrown exception in the same method, therefore we use the `throws` statement after the method name and catching the exception inside the `main()` method.

2.6.5 Nested Try Statements

The `try` statements can be nested. That is, a `try` statement can be inside the block of another `try`.

For example

```
class NestTry
{
    public static void main (String args[ ])
    {
        int i = args.length;
        int j = 42 / i;
        System.out.println ("i =" + i);
        try
        {
            if(i == i / (i - i));
            if(i == 2 )
            {
```



```
int k [ ] = { 1 };
k[42] = 99;
    }
}
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Array index out-of-bounds :" + e);
}
}
catch (ArithmeticException e)
{
    System.out.println (" Divide by 0 :" + e);
}
}
}
```

Output of the program:

```
i = 0
Divide by 0 : java.lang.ArithmeticException: Division by zero

i = 1
Divide by 0 : java.lang.ArithmeticException: Division by zero

i = 2
Array index out-of-bounds : java.lang.ArrayIndexOutOfBoundsException
: 42
```

2.6.6 Java Build-In Exceptions

Several exception classes are defined by Java. Exceptions are subclasses of the standard type **RuntimeException**. The unchecked exceptions are defined in **java.lang**.

RunTimeException Exceptions that inherit from this class include

- i. bad cast
- ii. out of bound array access
- iii. null pointer access

These problems arise out of wrong programming logic and must be corrected by the programmer himself. Some of these exceptions are:

- ArimeticException
- NullPointerException
- ClassCastException
- ArrayIndexOutOfBoundsException.

Creating Your Own Exception

Although most common errors are handled by Java's built-in exceptions, you can create your own exception types for handling situations particular to the applications. Hence a subclass of Exception is defined.

Methods of the Throwable Class

- `public Throwable ()` : This constructs a new Throwable object without any message.

NOTES

NOTES

- `public Throwable` : This constructs a new `Throwable` object
`(String message)`
with the message given. It can be used to provide information for debugging. All derived exception classes support both a default constructor with a detailed message.
- `public static` : This method obtains the detailed message
`String getMessage ()`
of the throwable object.
- `public void` : Prints this throwable and its back
trace `printStackTrace ()` to the
standard error stream.

For example:

```
class MyException extends Exception
{
    private int detail;
MyException (int p)
{
    detail = p;
}
public String toString ( )
{
    return "MyException[" + detail + "]";
}
}
class ExceptionDemo
{
    static void compute (int p) throws MyException
    {
        System.out.println(" Called compute (" + p + ")");
        if (p > 10)
            throw new MyException(p);
        System.out.println("Normal exit");
    }
public static void main (String args[ ])
{
    try
    {
        compute (1);
        compute (20);
    }
    catch (Myexception e)
    {
        System.out.println(" Caught" + e);
    }
}
}
```

Output of the program:

```
Called compute (1)
Normal exit
Called compute (20)
Caught MyExceptio0 [20]
```

2.6.7 User Defined Exceptions

As we come across built-in exception, you create own customized exception as per requirements of the application. On each application there is a specific constraint. For example in the case of a banking application, A customer whose age is less than eighteen needs to open Joint account. Thus, error-handling become necessary while developing a constraint application .The exception class and its subclass in Java is not able to meet up the required constraint in application. Thus you create user-defined exceptions to address these constraints and ensure the integrity in the application. Creating these functions makes an application more easily understood and user friendly. The keywords used in Java application are `try`, `catch` and `finally` to handle user defined exceptions.

NOTES

Program 2.24

```
import java.util.Scanner;
class ArmException extends Exception
{
    public ArmException()
    {
        super();
    }
    public String toString()
    {
        return "Armstrong Exception";
    }
}
public class Arm
{
    public static void main(String args[])
    {
        int i,j,k,sum=0;
        Scanner s1=new Scanner(System.in);
        System.out.println("Enter a number");
        i=s1.nextInt();
        j=i;
        while(i>0)
        {
            k=i%10;
            sum=sum+(k*k*k);
            i=i/10;
        }
        if(j==sum)
        {
            try{
                throw new ArmException();
            }
            catch(ArmException ae)
```

NOTES

```
        {  
            System.out.println(ae);  
        }  
    }else{  
        System.out.println("Not an Armstrong Number");  
    }  
}  
}
```

In the above program if we enter a Armstrong number then it generates Armstrong Exception and this is the user defined exception.

Restriction

When a method is overridden, one can throw only the **exceptions** that have been specified in the **base class method**. This is a very useful **restriction** since it means that the code which works with the base class will automatically work with any object derived from the base class, including the **exceptions**.

It is useful to realize that although **exception** specifications are enforced by the compiler during inheritance, the **exception** specifications are not part of the type of a method.

Check Your Progress

12. How does an interface differ from a class?
13. Give the definition of packages.
14. Write the definition of the term exception handling.
15. What is the significance of catch block?
16. How do you start a thread?
17. Why threads are called lightweight processes?
18. State about the nested try statement.
19. What are Java built-in exceptions?
20. Define the term restriction.

2.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A class is a user-defined data type that can be used to create instances of its type called objects. Like any other user-defined data type, it also needs to be declared and defined in a program. A class definition specifies a new data type that can be treated as a built in data type.
2. Two or more methods can be defined within the same class that shares the same name, till the time their parameter declarations are different in Java. Renaming the same method name with different arguments with the same or different return type is known as overloading method.

3. The return data type is declared in the function declaration in the main () function or the calling function and the declarator is indicated in the first line of the function definition.
4. Java supports recursion. Recursion is the process of defining a method that calls itself.
5. The advantages of using inner class are as follows:
 - Logical grouping of classes
 - Increased encapsulation
 - More readable, maintainable code
6. The String class is more commonly used to display messages and when strings need to be compared, searched or individual characters in a string have to be extracted as a substring.
7. One of the benefits of inheritance is code reusability because the derived class (child class) copies the member of the super class (parent class).
8. In Java, the concept of inheritance is implemented through super class. The super class is used to save the work of an existing class that can inherit the property of general class. It introduces better data analysis, reduces development time and gives fast performance. The Java super class is a type of class that provides methods to Java subclass.
9. Inheritance is implemented while defining the subclass. The name of the superclass is specified in the subclass definition. A subclass can be defined by using extends keyword.
10. If a subclass method has the same name, same parameter list and same return type as a superclass method, then we say that the method in the subclass overrides the method in the superclass. When the overridden method is called, the version of the method defined in the subclass will be invoked instead of the method defined in the superclass. That is, the method in the subclass will hide the method defined in the superclass.
11. When a class is declared as `final` then the class cannot be inherited. A `final` class cannot be declared through the `abstract` keyword.
12. An interface differs from a class such that, unlike a class, an interface contains only `final` variables and method declarations.
13. A software development, a task is divided into different modules and then each module is developed by different programmers. After that, all the modules are integrated together. If the software is developed in Java, then each module is definitely a class or combination of classes. A collection of such classes is called a package.
14. An exception signifies an illegal, invalid or unexpected issue during a program. Since exceptions are almost always assumed to be anticipated, you need to provide an appropriate exception handling. Exception handling means diverting the processing to a part of the program when an exception occurs.

NOTES

NOTES

15. The catch block catches and handles the exception thrown by the statements within the try block. The exceptions must belong to Exception or Error class.
16. We can start a thread by calling the start () method.
17. Threads are called lightweight processes because all the threads in a main application program share the same address space in the memory.
18. The try statements can be nested. That is, a try statement can be inside the block of another try.
19. Although most common errors are handled by Java's built-in exceptions, you can create your own exception types for handling situations particular to the applications. Hence a subclass of Exception is defined.
20. When a method is overridden, one can throw only the exceptions that have been specified in the base class method. This is a very useful restriction since it means that the code which works with the base class will automatically work with any object derived from the base class, including the exceptions.

2.8 SUMMARY

- A class is a user-defined data type that can be used to create instances of its type called objects. Like any other user-defined data type, it also needs to be declared and defined in a program. A class definition specifies a new data type that can be treated as a built in data type.
- Objects are data and methods bundled together into one logical software unit.
- Two or more methods can be defined within the same class that shares the same name, till the time their parameter declarations are different in Java. Renaming the same method name with different arguments with the same or different return type is known as overloading method.
- The return data type is declared in the function declaration in the main() function or the calling function and the declarator is indicated in the first line of the function definition
- Java supports recursion. Recursion is the process of defining a method that calls itself.
- A variable which has a constant value or a method that cannot be overridden in a subclass or a child class is specified by a final keyword.
- A nested or inner class is the member of its enclosing class. Nested class can be declared by using any access modifier. In Java, outer class or enclosing class is declared by using a public or a no access modifier. Nested class is able to access private, protected, no access or public member of enclosing class.
- Class declared within another class without using static modifier is treated as a non-static inner class. A non-static inner class is popularly known as an inner class.

NOTES

- One can also declare an inner class within the body of a method without naming it. Such classes are known as anonymous inner classes.
- The String class is more commonly used to display messages and when strings need to be compared, searched or individual characters in a string have to be extracted as a substring.
- A command line argument is the information that directly follows the program's name on the command line when it is executed.
- Inheritance is an important concept in Java. It facilitates code reusability. It is one of the corner-stones of object-oriented programming principles.
- One of the benefits of inheritance is code reusability because the derived class (child class) copies the member of the super class (parent class).
- In Java, the concept of inheritance is implemented through super class. The super class is used to save the work of an existing class that can inherit the property of general class. It introduces better data analysis, reduces development time and gives fast performance. The Java super class is a type of class that provides methods to Java subclass.
- Inheritance is implemented while defining the subclass. The name of the superclass is specified in the subclass definition. A subclass can be defined by using extends keyword.
- If a subclass method has the same name, same parameter list and same return type as a superclass method, then we say that the method in the subclass overrides the method in the superclass.
- Final method cannot be overridden. When we define a final method we never use abstract keyword.
- An interface differs from a class such that, unlike a class, an interface contains only final variables and method declarations.
- A software development, a task is divided into different modules and then each module is developed by different programmers. After that, all the modules are integrated together. If the software is developed in Java, then each module is definitely a class or combination of classes. A collection of such classes is called a package.
- A number of members from a package can be imported by import keyword. So, you can import the entire package if it is needed in the program.
- An exception signifies an illegal, invalid or unexpected issue during a program. Since exceptions are almost always assumed to be anticipated, you need to provide an appropriate exception handling. Exception handling means diverting the processing to a part of the program when an exception occurs.
- The catch block catches and handles the exception thrown by the statements within the try block. The exceptions must belong to Exception or Error class.
- It has been observed that when an exception is thrown in the program, the remaining statements in the try block are not executed and the control directly gets transferred to the subsequent catch block.

NOTES

- Threads are called lightweight processes because all the threads in a main application program share the same address space in the memory.
- The try statements can be nested. That is, a try statement can be inside the block of another try.
- Although most common errors are handled by Java's built-in exceptions, you can create your own exception types for handling situations particular to the applications. Hence a subclass of Exception is defined.
- When a method is overridden, one can throw only the exceptions that have been specified in the base class method. This is a very useful restriction since it means that the code which works with the base class will automatically work with any object derived from the base class, including the exceptions.

2.9 KEY TERMS

- **Class:** A class is a user-defined data type that can be used to create instances of its type called objects.
- **Recursion:** Java supports recursion. Recursion is the process of defining a method that calls itself.
- **Final keyword:** A variable which has a constant value or a method that cannot be overridden in a subclass or a child class is specified by a final keyword.
- **String class:** The String class is more commonly used to display messages and when strings need to be compared, searched or individual characters in a string have to be extracted as a substring.
- **Inheritance:** Inheritance is an important concept in Java.
- **Subclass:** Inheritance is implemented while defining the subclass.
- **Final method:** Final method cannot be overridden. When we define a final method we never use abstract keyword.
- **Package:** A collection of such classes is called a package.
- **Exception handling:** An exception signifies an illegal, invalid or unexpected issue during a program.
- **Try block:** Contains a set of statements that needs to be monitored for exceptions.
- **Thread:** A program which has a single flow of control and as a starting point, an execution part and an end.

2.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is a class and how is it related to objects?
2. Define the term objects parameters.

3. Name the types of visibility controls.
4. Write the syntax for `final` keyword.
5. Differentiate between static and non-static inner classes.
6. Write the key points of local inner classes.
7. Define the term command line arguments.
8. What is inheritance?
9. Define the term member access.
10. What is a subclass?
11. State about the calling constructor.
12. Define the term abstract class method.
13. What is `final` variable?
14. State the difference between an interface and an abstract class.
15. What is simple package member?
16. Why is exception propagation used?
17. What is the importance of `finally` block?
18. What is the difference between `throw` and `throws` keywords?
19. Define the term nested `try` statements.
20. Write the methods of the `Throwable` class.

Long-Answer Questions

1. Explain in detail the significant characteristics of methods and classes in C giving appropriate examples.
2. Discuss briefly method overloading and constructor overloading with the help of relevant example programs.
3. Explain in detail about the returning objects giving appropriate examples.
4. Briefly discuss the private, public and default visibility controls giving syntax and examples.
5. Write a C program to demonstrate the use of private, public and default visibility controls.
6. Briefly explain the significance of `final` and `static` keywords giving appropriate examples.
7. Write a C program using the `final` and `static` keywords.
8. Describe the types of C inner classes with the help of examples.
9. Discuss briefly the C string classes with the help of C programs. Write the output of the programs.
10. Briefly explain the inheritance and its types giving appropriate examples.
11. Differentiate between the super class and subclass with the help of syntax, relevant examples and programs.

NOTES

NOTES

12. How are the superclass constructors called? Explain with the help of C programs.
13. Describe the multilevel hierarchy in C giving appropriate examples.
14. Briefly explain the `final` class used in inheritance giving relevant examples.
15. 'Java supports the concept of multiple inheritances through interfaces.' Justify the statement giving appropriate examples.
16. In C programming how the packages are created? Explain with the help of C program.
17. Briefly explain the concept of exception handling in C programming giving relevant examples.
18. Write a C program using 'Divide by Zero' rule based on the concept of exception handling.
19. Briefly explain the `try-catch-finally` block with the help of C programs.
20. Explain the role of `throw` and `throws` keywords giving syntax and C programs.
21. Discuss briefly user define exceptions with the help of C programs.

2.11 FURTHER READING

- Balagurusamy, E. 2007. *Programming with Java*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Naughton, Patrick and Herbert Schidt. 1999. *Java 2: The Complete Reference*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Das, Rashmi Kanta. 2013. *Core Java for Beginners*, 3rd Edition. New Delhi: Vikas Publishing House Pvt. Ltd.
- Schildt, Herbert. 2006. *Java: The Complete Reference*, 7th Edition. New Delhi: Tata McGraw-Hill.
- Hunter, Jason and William Crawford. 2001. *Java Servlet Programming*, 2nd Edition. California: O'Reilly Media.
- Arnold, Ken, James Gosling and David Holmes. 2005. *The Java Programming Language*, 4th Edition. Boston: Addison-Wesley.
- Wigglesworth, Joe and Paula Lumby. 1999. *Java Programming Advanced Topics*, 2 Edition. Boston: Course Technology.
- Deitel, Paul and Harvey Deitel. 2011. *Java: How to Program*, 9th Edition. New Delhi: Prentice-Hall of India.

UNIT 3 MULTITHREADED PROGRAMMING, APPLETS, HANDLING STRING, java. lang AND UTILITY CLASSES

NOTES

Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Multithreaded Programming
- 3.3 Java Thread Model
 - 3.3.1 Thread Priorities
 - 3.3.2 Synchronization Messaging
 - 3.3.3 Thread Class
 - 3.3.4 Runnable Interface
 - 3.3.5 Creating Multiple Threads
 - 3.3.6 Suspending, Resuming and Stopping Threads
- 3.4 Basic Input/Output
 - 3.4.1 Streams (Byte and Character)
 - 3.4.2 Reading From and Writing To Console
 - 3.4.3 Reading and Writing Files
 - 3.4.4 PrintWriter Class
- 3.5 Fundamentals of Applets
 - 3.5.1 Transient and Volatile Modifier
 - 3.5.2 Modifier Strictfp
 - 3.5.3 Native Interface
- 3.6 String Handling
 - 3.6.1 Operations on String and Extract Character Methods
 - 3.6.2 StringBuffer
- 3.7 Wrapper Classes
 - 3.7.1 Memory Management
 - 3.7.2 java.lang Environment Properties
 - 3.7.3 Security Manager and SecurityManager Class
- 3.8 Java Utility Class
- 3.9 Enumeration Interface
 - 3.9.1 Using Store () and Load ()
- 3.10 Answers to 'Check Your Progress'
- 3.11 Summary
- 3.12 Key Terms
- 3.13 Self Assessment Questions and Exercises
- 3.14 Further Reading

3.0 INTRODUCTION

The concept of threads and the idea of multithreading was introduced a decade back, however, it has been accepted into the main stream programming only recently. Threads are very valuable and useful. Programs that are written with threads are good even to the not so regular users. A thread is just like a program which has a

NOTES

single flow of control. It also has a starting point, the execution part and an end. The main programs in the preceding examples can be called single-threaded programs. Java also allows us to use multiple flows of control in a program and such a program is known as multithreaded program. In a multithreaded program, each thread is a separate tiny module which runs in parallel with other threads. Running in parallel does not mean that they are running at the same time. As most of the times the threads run on the single processor, in reality, only one thread is executed at a given time.

Console-based application programs are full-featured, stand-alone Java programs, written, modified and run through Java Virtual Machine (JVM). Java defines another category of application programs known as applets. Java streams can be broadly categorized into two types, namely input stream and output stream. The streams which help to read data from various sources, namely keyboard, mouse, files, storage devices, etc., in order to supply it to the program are called input streams. The streams which receive data from the program and directly write it to the physical devices or other programs are called output streams.

The `java.lang` package is the primary and the most important package available to the programmer by default. This package contains classes like: `runtime` and `process`, which are used for system call, `thread` and `Runnable` class, required to design multithreaded applications, `Cloneable` interface, which is a marker interface, required to create a copy of an object. The `java.lang` is the fundamental package that is available to the programmer by default. He can use the classes, interfaces and methods available in this package, without requiring to import these. In fact, this is a package that is widely used by all Java programmers. The `java.util` package provides many interfaces and utility classes for easy manipulation of data. `java.util` package contains the collections framework, legacy collection classes, event model, date and time facilities, and miscellaneous utility classes, such as a string tokenizer, a random number generator, and a bit array.

In this unit, you will study about the multithreaded programming, Java thread model, priorities, synchronization messaging, `Thread` class and `Runnable` interface, creating multiple threads, suspending, resuming and stopping threads, basic input/output, streams (byte and character), reading from and writing to console, readings and writing files, `PrintWriter` class, fundamentals of applets, `transient` and `volatile` modifier, `Modifier` `strictfp`, `Native` interface, handling string, string length, operations on strings, extract character methods, string comparison method, `StringBuffer`, wrapper classes, `void`, `AbstractProcess` class, `runtime` class and memory management, `System` class, environments properties, using `Clone ()`, and `Cloneable ()` interface, `ClassLoader`, `Math` class, `Thread`, `ThreadGroup` and `Runnable` interface, `Throwable` class, `SecurityManager`, `java.lang.ref` and `java.lang.reflect` packages, Java utility classes, `Enumeration` interface, `Vector`, `Stack`, `Dictionary`, `Hashtable`, properties, using `Store ()` and `Load ()`, string tokenizer, `BitSet` class, date and date comparison, time zones, `Random` class, `Observable`.

3.1 OBJECTIVES

*Multithreaded
Programming, Applets,
Handling String, java.
lang and Utility Classes*

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

- Understand the Java multithreaded programming
- Explain the Java thread model and priorities
- Analyse the synchronization messaging
- Define the thread class and runnable interface
- Elaborate on multiple threads
- Discuss the suspending, resuming and stopping threads
- Describe the basic input / output
- Understand the streams (byte and character)
- Elaborate on the reading from and writing to console
- Define the reading and writing files
- Analyse the `PrintWriter` class and fundamentals of applets
- Explain the transient and volatile modifier, `strictfp`
- Describe the native interface
- Understand the handling strings and string length
- Explain the operations on strings and extract character methods
- Analyse the `StringBuffer` and `wrapper` classes
- Describe the void and abstract process class
- Understand the runtime class and memory management
- Discuss the system class
- Explain the environment properties using `Clone ()` and `Cloneable ()` interface
- Analyse the `ClassLoader` and `Math` class
- Define the thread and thread group and runnable interface
- Describe the `Throwable` class and security manager
- Understand the `java.lang.ref` and `java.lang.reflect` packages
- Explain the Java utility classes and enumeration interface
- Elaborate on the vector, stack and dictionary
- Understand the hashtable properties using `Store ()` and `Load ()`
- Define the string tokenizer and bitset class
- Explain the date and date comparison and time zone
- Discuss the random class and observe

NOTES

NOTES

3.2 MULTITHREADED PROGRAMMING

The concept of threads and the idea of multithreading was introduced a decade back, however, it has been accepted into the mainstream programming only recently. Threads are very valuable and useful. Programs that are written with threads are good even to the not so regular users.

A program starts executing when it runs its initialization code, calls methods and procedures and continues to run until the program reaches an end. This is a scenario familiar to you in single user systems and even in multi-user systems. The program uses a single thread or a single locus of control. A thread is not restricted to one function alone. Any thread in a multithreaded program can call any series of statements and functions that would be called in a single threaded program.

Creating a Thread

There are two ways to create a thread in Java. One method is used exclusively in Java applications. The other method can be used in applications as well as in applets.

Creating Threads in Applications

There is a class called Thread in java.lang package and you can create a thread of your own by deriving a class from java.lang.Thread class. This is how it is done:

```
class ExampleThreadClass extends Thread
{
    public void run()
    {
        _____
        // do something useful
        _____
    }
}
```

You have created a new class called ExampleThreadClass which does something useful when its run () method is called. To call the run () method, you have to create an instance of the class ExampleThreadClass and invoke the start () method shown as follows:

```
ExampleThreadClass thread = new ExampleThreadClass();
```

Thread.start ();

This may look bit confusing since you have to start the method running by calling a method start () which is not there in the class ExampleThreadClass .

You have created ExampleThreadClass by extending the Thread class which looks something shown as follows:

```
class Thread
{
    public void start()
    {
        _____
        _____
    }
}
```

```
}
```

`Thread.start()` (`ExampleThreadClass.start()`) is equivalent to calling `Thread.start()` which will cause `Thread.run()` running.

Declaring an object of class `ExampleThreadClass` creates a new thread whose execution will start in the method `run()`. The `run()` method is the place where you specify what a thread has to perform.

Creating Multiple Threads

Suppose you have a program in which there is a long computation in the beginning. If you use single threaded program you have to wait for the completion of the calculation before the rest of the program can continue running. In a multithreaded program, you can put the computation in its own thread, enabling the rest of the program to continue.

For example:

```
class Example1Thread1Class extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.println("Hello Thread1");
        }
    }
}

class Example1Thread2Class extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.println("Hello Thread2");
        }
    }
}

public class Example1
{
    public static void main(String argv[])
    {
        Example1Thread1Class thread1 = new Example1Thread1Class(
);
        Example1Thread2Class thread2 = new Example1Thread2Class(
);

        thread1.start();
        thread2.start();
    }
}
```

Output of the program:

```
Hello Thread1
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread1
Hello Thread1
Hello Thread1
```

NOTES

NOTES

Hello Thread1
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread1
Hello Thread1
Hello Thread1
Hello Thread1
Hello Thread2
Hello Thread2
Hello Thread2
Hello Thread1
Hello Thread1

The output generated in a subsequent run may not be exactly identical. However, the general pattern will be the same. The program will print the two messages Hello Thread1 and Hello Thread2 in a shuffled fashion.

The Thread class has many methods:

Thread.start()	-	calls the run() method
Thread.stop()	-	stops the thread
Thread.suspend()	-	suspends a thread execution
Thread.resume()	-	resumes a suspended thread
Thread.sleep(100)	-	sleeps for 100 milli seconds
Thread.wait()	-	waits for something to happen
Thread.yield()	-	yields the thread control to another thread
Thread.setPriority()	-	set thread priority of the current thread
Thread.getPriority()	-	gets thread priority of the current thread
Thread.currentThread()	-	gets the thread in which the method is running
Thread.getName()	-	gets the name of the thread in which the method is running

For example:

```
class simpleThread extends Thread
{
    public SimpleThread(String str)
    {
        super(str);
    }
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.println(getName());
        }
    }
}
public class Example2
{
    public static void main(String argv[])
    {
        new SimpleThread("Thread1").start();
        new SimpleThread("Thread2").start();
    }
}
```



```
}
```

Output of the program:

```
Thread1  
Thread1  
Thread1  
Thread1  
Thread1  
Thread1  
Thread1  
Thread2  
Thread2  
Thread2  
Thread2  
Thread2  
Thread1  
Thread1  
Thread2  
Thread2  
Thread2  
Thread2  
Thread2  
Thread1
```

In a subsequent run, the output need not be exactly the same. However, the general pattern of the output will be the same.

In this example, you have only one `run ()` method. Two threads are sharing the same run method. The class method `getName ()` is called to get the thread name in which the `run ()` method is currently running. The class `SimpleThread` is a direct descendent of the class `Thread` and hence the method `getName ()` of `Thread` class could be called.

The Runnable Interface

The `Runnable` interface looks like this:

```
public interface Runnable  
{  
    public abstract run ();  
}
```

The interface `Runnable` specifies only one method, the `run ()` method. In the implementation of this `run ()` method, however, you cannot use methods like `stop ()`, `sleep ()`, `getName ()`,..... which are member functions of the `Thread` class.

A class that implements `Runnable` interface works much in the same way as a class that extends `Thread` class. You can create an instance of that class and pass that instance to the constructor for making a new thread as follows:

1. A class is created implementing `Runnable` Interface.

```
public class ExampleThreadClass implements Runnable  
{  
    public void run ()  
    {  
        _____  
        _____  
    }  
}
```

NOTES

NOTES

- ```
 }
```
2. Create an instance of that class  
`ExampleThreadClass aETC = new Example Thread  
Class ();`
  3. Pass that instance to the constructor making a new thread  
`Thread aThread = new Thread(aETC);`
  4. Then the statement  
(a) `Thread.start ();`  
calls the `run ()` method indirectly.

Listing below gives a Java program in which named threads are created by implementing the Runnable interface.

Creating multiple threads by implementing Runnable interface is as follows.

```
class ExampleThreadClass implements Runnable
{
 public void run()
 {
 for(int i=0;i<10;i++)
 {
 }
 }
 }
public class Example
{
 public static void main (String argv[])
 {
 ExampleThreadClass aETC=new ExampleThreadClass ();
 Thread thread1=new Thread (aETC, "My Thread");
 Thread thread2=new Thread (aETC, "Your Thread");
 thread1.start ();
 thread2.start ();
 }
}
```

### Output of the program:

```
MyThread
Your Thread
Your Thread
Your Thread
Your Thread
Your Thread
MyThread
MyThread
MyThread
MyThread
Your Thread
Your Thread
Your Thread
Your Thread
MyThread
MyThread
MyThread
MyThread
```

```
MyThread
Your Thread
Your Thread
```

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

In a subsequent run, the output need not be exactly the same. However, the general pattern of the output will be the same.

In this example, you have only one `run ()` method. Two threads share the same `run ()` method. You cannot use `getName()` method here because the class `ExampleThreadClass` is not derived from the class `Thread`. The static method `currentThread ()` of the `Thread` class is used to get the current thread in which `run ()` method is running. `Thread.currentThread () . getName ()` will give the name for the thread in which the `run ()` method is currently running.

### Thread Scheduling

Thread scheduling specifies exactly in what order, your thread will be run. There are two strategies: non pre-emptive scheduling and pre-emptive time slicing. In non pre-emptive scheduling scheme. It always asks for permission to schedule. Most modern schedulers use pre-emptive time slicing. In pre-emptive time slicing, each thread will run for a few milliseconds before it yields control to another thread. To test whether your system uses non pre-emptive scheduling or pre-emptive time slicing, try the program listing on your computer system.

A Java program to test the type of scheduler in your system is as follows.

```
class ExampleThreadClass implements Runnable
{
 public void run ()
 {
 while (true)
 {
 System.out.println (Thread.currentThread () . getName ());
 }
 }
}
public class Example
{
 public static void main (String argv [])
 {
 ExampleThreadClass aETC = new ExampleThreadClass ();
 Thread thread1 = new Thread (aETC, "boys");
 Thread thread2 = new Thread (aETC, "girls");
 thread1.start ();
 thread2.start ();
 }
}
```

For a non-preemptive scheduler, the above program will print

```
boys
boys
boys
.
.
.
```

### NOTES



```
class ExampleThreadClass implements Runnable
{
 public void run()
 {
 while (true)
 {
 System.out.println(Thread.currentThread().getName());
 Thread.yield();
 }
 }
}

public class Example
{
 public static void main (String argv[])
 {
 ExampleThreadClass aETC = new ExampleThreadClass ();
 new Thread(aETC, "boys").start ();
 new Thread(aETC, "girls").start ();
 }
}
```

## NOTES

### Output of the program:

```
.
.
.
.
girls
girls
boys
boys
girls
boys
girls
boys
girls
boys
girls
.
.
.
```

.....until the program was interrupted. The `yield()` method gives a chance to other waiting threads to run. If there are no waiting threads, the thread that made the yield will continue to run.

Yet another method to ensure alternate execution of two threads is to use `sleep()` method. The `sleep()` method forces the current thread to yield and then wait for at least the specified amount of time to elapse before allowing the thread to run again. Another thread, however, might interrupt the sleeping thread. In such a case, it throws an `InterruptedException`. The following program shows how to use it.

A Java program ensures that both the threads are running alternatively, irrespective of the scheduling strategy. `sleep()` method in use.

## NOTES

```
class ExampleThreadClass implements Runnable
{
 public void run()
 {
 while (true)
 {
 System.out.println(Thread.currentThread().getName());
 try
 {
 Thread.sleep(100);
 } catch (InterruptedException e)
 {
 return;
 }
 }
 }
}

public class Example
{
 public static void main (String argv[])
 {
 ExampleThreadClass aETC = new ExampleThreadClass ();
 new Thread(aETC, "boys").start ();
 new Thread(aETC, "girls").start ();
 }
}
```

### Output of the program:

```
:
:
boys
girls
boys
girls
boys
girls
boys
:
:
```

until the program was interrupted.

### Thread priorities

Threads have priorities that can be set and changed. A higher priority thread executes ahead of a low priority thread. Priorities run from 1 (lowest) to 10 (highest). Threads take off with the same priority.

The Java program shows the priorities:

```
class ExampleThreadClass implements Runnable
{
 public void run()
 {
 while (true)
 {
 System.out.println(Thread.currentThread().getName (
```

```
));
 }
}
}
public class Example
{
 public static void main (String argv[])
 {
 ExampleThreadClass aETC=new ExampleThreadClass ();
 Thread thread1=new Thread(aETC, "boys");
 Thread thread2=new Thread(aETC, "girls");
 thread2.setPriority (thread1.getPriority ()+2);
 thread1.start ();
 thread2.start ();
 }
}
```

## NOTES

### Output of the program:

```
:
:
girls
girls
girls
girls
:
:
```

until the program was interrupted by pressing Ctrl +C.

In the above program, change the statement

```
thread2.setPriority (thread1.getPriority ()+2);
thread2.setPriority (thread1.getPriority ()+1);
to
thread2.setPriority (thread1.getPriority ()+3);
thread2.setPriority (thread1.getPriority ()+4);
```

and observe the difference in the output generated in each case.

---

## 3.3 JAVA THREAD MODEL

---

A single sequential flow of control, that is, the program simply starts, performs a series of operations and eventually ends. There is only one statement under execution at any given point of time. A thread is just like a program which has a single flow of control. It also has a starting point, the execution part and an end. The main programs in the preceding examples can be called single-threaded programs. Java also allows us to use multiple flows of control in a program and such a program is known as multithreaded program. In a multithreaded program, each thread is a separate tiny module which runs in parallel with other threads. Running in parallel does not mean that they are running at the same time. As most of the times the threads run on the single processor, in reality, only one thread is executed at a given time. However, the switching from one thread to another

## NOTES

thread occurs so fast that it gives an illusion to the user that all the threads are being executed at the same time. Threads are called lightweight processes. This is because all the threads in a main application program share the same address space in the memory.

There are several advantages of using multithreaded programs over single-threaded programs. In a traditional single-threaded environment, the CPU sits idle most of the times, as the program has to wait for each of the task to complete before proceeding to the next task. In multithreaded environment, since different tasks can be assigned to different threads, the program makes maximum utilization of the CPU, keeping the idle time of the CPU to minimum. For example, one thread can read data, another thread can process it and a third thread can write it; thus, improving the overall performance. Multithreading is best-suited for those applications that require multiple tasks to be done simultaneously.

### Main Thread

Java program always contains at least one thread, even if we do not create one. This thread is called main thread and it is the one which immediately starts executing when we start a program. The main thread can be used to create and start other child threads and it must often be the last thread to finish execution because it performs various other actions such as, shutdown action and releasing resources which are used by the program. The main thread is created automatically, but it can be controlled through a `Thread` object. For this, its reference is needed which can be obtained by calling the method `currentThread()`, which is a public static member of `Thread` class. This method returns a reference to the thread on which it is called. We can control the main thread just like any other thread, once we have a reference to it. This has been explained with the help of Example 3.1.

**Example 3.1:** A program to control the main thread is as follows:

```
class MainThread
{
 public static void main (String [] args)
 {
 Thread th=Thread.currentThread();
 System.out.println ("The name of the current thread is: "+th);

 //changing the name of the thread
 th.setName ("MyThread");
 System.out.println ("The name of the current thread after changing
the name is: "+th);
 System.out.println ("Main Thread exiting");
 }
}
```

**The output of the program is:**

```
The name of the current thread is: Thread[main,5,main]
The name of the current thread after changing the name is:
Thread[MyThread,5,main]
Main Thread exiting
```



In the program illustrated in Example 3.1, the method `currentThread()` is called to obtain a reference to the main thread and this reference is stored in the variable `th`. Here, the name of the main thread is changed by calling the method `setName()` and the new thread name is redisplayed.

## Creating Threads

In Java, threads can be created in two ways, which are as follows:

- (i) By defining a class that extends the `Thread` class
- (ii) By implementing the `Runnable` interface

In both the approaches, threads are implemented in the form of objects which contain a method called `run()`. It is the most important method in the `Thread` class. It is the entry point of a new thread and it is the place where the task to be performed by the thread is defined. The execution of a thread starts with the call to `run()` method. The `run()` method is automatically invoked when we invoke another method of `Thread` class called `start()`.

## Extending Threads

We can create a thread by creating a new class that extends the `Thread` class defined in `java.lang` package and creating an instance of the class. The extending class must override the `run()` method. Inside the `run()` method, the code that needs to be executed by the thread will be defined.

The code segment to extend the `Thread` class and override the `run()` method is as follows:

```
class ThreadName extends Thread
{
 public void run()
 {
 : // code for the new thread
 }
}
```

Now, the instance of the class `ThreadName` can be created and run using the statements as follows:

```
ThreadName objectname=new ThreadName();
objectname.start();
```

The second statement invokes the `start()` method, after which the thread will be ready to run. It will start running once the Java runtime invokes its `run()` method. Example 3.2 illustrates how to create threads by extending the `Thread` class.

**Example 3.2:** A program to demonstrate creating threads by extending the `Thread` class is as follows:

```
class Thread1 extends Thread
{
 public void run() //entrypoint of Thread1
 {
 int i=0;
 while(i<5)
 {
```

## NOTES

## NOTES

```
 System.out.println("FirstChildThread:"+i);
 i=i+1;
 }
 System.out.println("\tFirst child exited");
}
}
class Thread2 extends Thread
{
 public void run() //entrypoint of Thread2
 {
 int j=0;
 while(j<5)
 {
 System.out.println("SecondChildThread:"+j);
 j=j+1;
 }
 System.out.println("\tSecond child exited");
 }
}

class ExtendingThread
{
 public static void main (String[] args)
 {
 Thread1 firstthread=new Thread1 ();
 firstthread.start (); // starts the first thread
 Thread2 secondthread=new Thread2 ();
 secondthread.start (); // starts the second thread

 int k=0;
 while (k<5)
 {
 System.out.println("Main Thread:"+k);
 k=k+1;
 }
 System.out.println("\tMain thread exiting");
 }
}
}
```

### The output of the program is:

```
FirstChildThread:0
MainThread:0
SecondChildThread:0
FirstChildThread:1
MainThread:1
SecondChildThread:1
FirstChildThread:2
MainThread:2
SecondChildThread:2
FirstChildThread:3
MainThread:3
SecondChildThread:3
FirstChildThread:4
MainThread:4
SecondChildThread:4
First child exited
Main thread exiting
Second child exited
```

When we start the program in Example 3.2, the main thread immediately starts running. The main thread then starts two child threads the Thread1 and Thread2, both of which will perform different tasks. Once the main thread reaches the end of the main () method, there will be altogether three threads running concurrently on their own in the program, Thread1, Thread2 and the main thread. These three threads will run independently whenever the CPU is available to them. There is no specific order of their execution. Hence, the program may generate different output every time we run it.

### **Implementing Runnable Interface**

Another way of creating a thread is to create a class that implements the Runnable interface (Refer Example 3.3). The Runnable interface consists of a single method run () which is required for implementing a thread. We will create a thread and pass the object of the class that implements the Runnable interface as an argument of the Thread class's constructor. The thread will now be activated by calling the start () method. Implementing Runnable interface is much more convenient than extending a Thread class when a program needs to inherit from a class apart from the Thread class, since Java allows only a single base class.

The code segment to implement Runnable interface is as follows:

```
class MyNewThread implements Runnable
{
 public void run ()
 {
 : // code for the new thread
 }
}
```

**Example 3.3:** A program to demonstrate creating threads by implementing Runnable interface is as follows:

```
class MyNewThread implements Runnable // implements Runnable
{
 public void run () // implements run () method
 {
 int i=0;
 while (i<=4)
 {
 System.out.println ("ChildThread: "+i);
 i++;
 }
 }
}
class RunnableInterface
{
 public static void main (String [] args)
 {
 /*an object of class implementing Runnable interface*/
 MyNewThread runnableobj=new MyNewThread ();

 /*an object of Thread class taking runnable object as argument*/
 Thread threadobj=new Thread (runnableobj);
 }
}
```

## **NOTES**

## NOTES

```
threadobj.start();
int j=0;
while(j<=4)
{
 System.out.println("MainThread:"+j);
 j++;
}
System.out.println("MainThreadExiting");
}
```

### The output of the program is:

```
MainThread:0
ChildThread:0
MainThread:1
ChildThread:1
MainThread:2
ChildThread:2
MainThread:3
ChildThread:3
MainThread:4
ChildThread:4
MainThreadExiting
```

In the program illustrated in Example 3.3, `MyNewThread` is a class which implements the `Runnable` interface. Inside the `main()` method, an instance `runnableobj` of `MyNewThread` is created which is passed as an argument to the `Thread` class's constructor. When the new thread starts, the `run()` method of `runnableobj` is called.

### 3.3.1 Thread Priorities

The threads we have seen so far are of equal priority in which the Java scheduler selects the thread for execution on the first-come, first-serve basis. However, each thread can be assigned different priority which will decide the order in which it is scheduled for running. Priorities are the integers which specify the relative priority of one thread to another. When a thread is created, it inherits its priority from the thread that created it. However, the priority of a thread can be changed by using the `setPriority()` method of the `Thread` class.

The syntax to set the priority of a thread is as follows:

```
ThreadName.setPriority(n);
```

where, `n` is an integer value which ranges from `MIN_PRIORITY` (1) and `MAX_PRIORITY` (10). The default priority is `NORM_PRIORITY` whose value is 5. `MIN_PRIORITY`, `MAX_PRIORITY` and `NORM_PRIORITY` are the constants defined in `Thread` class.

When there are multiple threads ready to execute, the highest priority thread is chosen and executed. Only when the high priority thread stops, yields or enters blocked state, the low priority thread starts running. However, if any higher priority thread enters, it will preempt the currently running thread forcing it to move to the runnable state. Example 3.4 illustrates how priority is assigned to a thread.

**Example 3.4:** A program to demonstrate the assigning of priority to a thread is as follows:

```
class Thread1 extends Thread
{
public void run () //entrypoint of the Thread1
 {
int i=0;
 while (i<5)
 {
 System.out.println("First Child Thread:" +i);
 i=i+1;
 }
 System.out.println("\tFirst child exited");
}
}
class Thread2 extends Thread
{
 public void run () //entrypoint of the Thread2
 {
 int j=0;
 while (j<5)
 {
 System.out.println("Second Child Thread:" +j);
 j=j+1;
 }
 System.out.println("\tSecond child exited");
 }
}
class ThreadPriority
{
 public static void main (String [] args)
 {
 Thread1 firstthread=new Thread1 ();
 Thread2 secondthread=new Thread2 ();

 // Thread2 assigned highest priority
 secondthread.setPriority (Thread.MAX_PRIORITY);

 // Thread1 assigned lowest priority
 firstthread.setPriority (Thread.MIN_PRIORITY);
 firstthread.start ();
 secondthread.start ();

 System.out.println("\tMain Thread Exiting");
 }
}
}
```

**The output of the program is:**

```
SecondChildThread:0
SecondChildThread:1
SecondChildThread:2
SecondChildThread:3
SecondChildThread:4
Secondchildexited
MainThreadExiting
FirstChildThread:0
FirstChildThread:1
FirstChildThread:2
```

**NOTES**

```
FirstChildThread:3
FirstChildThread:4
Firstchildexited
```

## NOTES

In the program illustrated in Example 3.4, the first child thread, Thread1 has been assigned the minimum priority and the second child thread, Thread2 has been assigned the maximum priority. So despite Thread1 being the first on which start () method is called, its output is printed in the last as it has been preempted by the higher priority thread—Thread2.

*Note:* The amount of CPU time a thread gets depends not only on its priority but also on other factors, such as how an operating system implements multithreading.

### 3.3.2 Synchronization Messaging

When multiple threads need access to a single resource, there must be a way to ensure that only one thread will use the resource at any given point of time, otherwise it may lead to a severe problem. For example, if one thread in a program reads salary from a file and another thread tries to update it, then the program may produce an undesirable output. The solution to this problem can be achieved by using a technique known as synchronization. The objective of synchronization is to control the access to shared resources.

Synchronization uses the concept of monitor. A monitor is an object which is used as a mutually exclusive lock. That is, it can be owned by only one thread at any given point of time. A thread is said to have entered the monitor when it acquires a lock. Any other thread which attempts to acquire the lock has to wait until the first thread comes out of the monitor. There are two ways to implement synchronization, which are as follows:

- (i) Synchronizing Methods
- (ii) Synchronizing Statements

#### Synchronizing Methods

We can synchronize a subset (or all) of the methods of any class by using synchronized keyword. When a method is declared synchronized, Java creates a monitor. To enter the monitor, we need to call a synchronized method. Only one of the synchronized methods in a class object can execute at any given time. Java hands over the monitor to the thread that calls the method first. As long as a thread is inside a synchronized method, other threads trying to call it (or any other synchronized method) on the same instance have to wait. Only when the currently executing thread finishes executing and exits the monitor another waiting thread can enter the monitor.

The syntax to declare a method as synchronized is as follows:

```
synchronizeddata_typemethod_name()
{
 // code for the method
}
```

To understand synchronization, let us consider the program given in Example 3.5, which is not synchronized.

**Example 3.5:** A program to demonstrate the need of synchronization is as follows:

```
class A
{
 void display(String msg)
 {
 System.out.print("(" + msg);
 try
 {
 Thread.sleep(1000);
 } catch (InterruptedException e)
 {
 System.out.println("Interrupted");
 }
 System.out.println(")");
 }
}

class MyThread extends Thread
{
 String str;
 A obj;
 MyThread(A obj1, String s)
 {
 obj = obj1;
 str = new String(s);
 }
 public void run()
 {
 obj.display(str);
 }
}

class UnsynchronizedMethod
{
 public static void main(String[] args)
 {
 A obj = new A();
 MyThread th1 = new MyThread(obj, "THIS");
 MyThread th2 = new MyThread(obj, "IS");
 MyThread th3 = new MyThread(obj, "SYNCHRONIZATION");
 th1.start();
 th2.start();
 th3.start();
 }
}
```

**The output of the program is:**

```
(THIS (IS (SYNCHRONIZATION)
)
)
```

The class A has a method named `display()` which takes a parameter `msg` of `String` type. This method will print the `msg` string enclosed in the first brackets. Note that the `sleep()` method is invoked after the `display()` method prints the opening bracket and the string `msg` which causes the current thread to halt for one second. The constructor of class `MyThread` takes two arguments,

## NOTES

## NOTES

a reference to an instance of class A and a string. When the first thread starts, the object's `run()` method is invoked. The `run()` method invokes the `display()` method on the instance `obj` of A, and passes the string `str`. The call to `sleep()` method allows execution to switch to another thread before the first thread could complete the method. Thus, the output of the program is not as expected and the strings are in mixed up form. This is because, the three threads call the same method `display()` without anything to stop them from competing each other to complete the method.

Now let us modify the program in Example 3.5 by preceding the definition of `display()` method with `synchronized` keyword (Refer Example 3.6). This will serialize access to `display()` method by restricting its access to only one thread at a time, thus producing the correct output.

**Example 3.6:** A program to demonstrate synchronized method is as follows:

```
classA
{
 synchronizedvoiddisplay(Stringmsg)
 {
 System.out.print("(" +msg);
 try
 {
 Thread.sleep(1000);
 }catch(InterruptedException)
 {
 System.out.println("Interrupted");
 }
 System.out.println(")");
 }
}
classMyThreadextendsThread
{
 Stringstr;
 Aobj;
 MyThread (Aobj1, Strings)
 {
 obj=obj1;
 str=newString (s);
 }
 publicvoidrun ()
 {
 obj.display (str);
 }
}

classSynchronizedMethod
{
 publicstaticvoidmain (String[] args)
 {
 Aobj=newA ();
 MyThreadth1=newMyThread (obj, "THIS");
 MyThreadth2=newMyThread (obj, "IS");
 MyThreadth3=newMyThread (obj, "SYNCHRONIZATION");
 th1.start ();
 th2.start ();
 th3.start ();
 }
}
```



```
 }
}
```

### The output of the program is:

```
(THIS)
(IS)
(SYNCHRONIZATION)
```

**Note:** Once a thread is in synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, simultaneous execution of synchronized methods is possible for two different instance, of the same class.

### Synchronizing Statements

Another way of managing the execution of the thread is to synchronize a block of code or statement. This is more powerful. Synchronizing a method does not work in all cases. For example, the class we want to access is created by someone else, which does not have synchronized methods and we do not have access rights to modify it. In this case, the access to objects of this class can be synchronized by placing the call to the methods defined by this class inside a synchronized block. Two block of codes synchronized on the same instance cannot execute at the same time. Example 3.7 illustrates synchronized statements.

The general form to synchronize a block of code is as follows:

```
synchronized(object)
{
 //statements to be synchronized
}
```

where,

object is a reference to the object being synchronized.

**Example 3.7:** A program to demonstrate synchronized statement is as follows:

```
classA
{
 voiddisplay(Stringmsg)
 {
 System.out.print("(" +msg);
 try
 {
 Thread.sleep(1000);
 }catch (InterruptedException)
 {
 System.out.println("Interrupted");
 }
 System.out.println(")");
 }
}
classMyThreadextends Thread
{
 Stringstr;
 Aobj;
 MyThread (Aobj1, String s)
 {
 obj=obj1;
 str=newString (s);
 }
}
```

## NOTES

## NOTES

```
public void run()
{
 synchronized(obj)
 {
 obj.display(str);
 }
}
class SynchronizedStatement
{
 public static void main(String[] args)
 {
 A obj=new A();
 MyThread th1=new MyThread(obj, "THIS");
 MyThread th2=new MyThread(obj, "IS");
 MyThread th3=new MyThread(obj, "SYNCHRONIZATION");
 th1.start();
 th2.start();
 th3.start();
 }
}
```

### The output of the program is:

(THIS)

(IS)

(SYNCHRONIZATION)

By using the synchronized statement inside the run () method, access to the object of the class A is restricted to only one thread at a time, thus producing the same correct output.

### Deadlock

Deadlock is a situation that occurs when two or more threads are in a simultaneous wait state and each of them is waiting for the release of a resource held by one of the other waiting thread. For example, consider the following code segments:

```
run()
{
 synchronized(obj1)
 {
 sleep(1000);
 obj2.method2();
 }
}
run()
{
 synchronized(obj2)
 {
 sleep(1000);
 obj1.method1();
 }
}
```

**Thread X:**

**Thread Y:**

First, Thread X starts and synchronizes on the object obj1, which prevents other threads to call the methods of obj1. Thread X then goes to sleep by calling sleep () method and allows Thread Y to start. Thread Y starts and synchronizes on the object obj2. This prevents method of obj2 to be called by any other thread. Thread Y goes to sleep on the invocation of sleep () method allowing Thread X to wake up. Thread X continues execution and tries to call method2 () on obj2. However, it cannot call the method on obj2 until the code in Thread Y that is synchronized on obj2 finishes execution. As Thread X cannot proceed, Thread Y gets the control and tries to call method1 () on obj1 which is not possible until the code in Thread X that is synchronized on obj1 finishes its execution. Here, neither of the threads can continue because they are deadlocked. Example 3.8 shows the condition of deadlock.

## NOTES

**Example 3.8:** A program to demonstrate deadlock is as follows:

```
classA
{
voiddisplay1 (Aobj2)
{
System.out.println ("First threadwaitingforsecondthreadtorelease
theresource");
synchronized (obj2)
{
System.out.println ("Deadlocked");
}
}
voiddisplay2 (Aobj1)
{
System.out.println ("Secondthreadwaitingforfirstthreadtorelease
theresource");
synchronized (obj1)
{
System.out.println ("Deadlocked");
}
}
}
classThread1extendsThread
{
Aobj1,obj2;
Thread1 (Ai,Aj)
{
obj1=i;
obj2=j;
}
publicvoidrun ()
{
synchronized (obj1)
{
try
{
sleep (1000);
}
catch (Exceptione)
{
```

## NOTES

```
 System.out.println(e);
 }
 obj2.display1(obj2);
}
}
class Thread2 extends Thread
{
 Aobj1, obj2;
 Thread2 (Ap, Aq)
 {
 obj1=p;
 obj2=q;
 }
 public void run()
 {
 synchronized(obj2)
 {
 try
 {
 sleep(1000);
 }
 catch (Exception e)
 {
 System.out.println(e);
 }
 obj1.display2(obj1);
 }
 }
}
class Deadlock
{
 public static void main (String args [])
 {
 Aobj1=new A ();
 Aobj2=new A ();
 Thread1 t1=new Thread1 (obj1, obj2);
 Thread2 t2=new Thread2 (obj1, obj2);
 t1.start ();
 t2.start ();
 }
}
```

### The output of the program is:

```
Second thread waiting for first thread to release the resource
First thread waiting for second thread to release the resource
```

In the program illustrated in Example 3.8, the thread Thread1 owns the monitor on obj1 and waits for the monitor on obj2. Similarly, the thread Thread2 owns the monitor on obj2 and waits for the monitor on obj1. Thread1 will never release obj1 unless it gets hold of obj2 and Thread2 will never release obj2 unless it gets obj1. The program will never complete as the two threads are in the deadlock situation. We need to press CTRL+C to end the program.

### 3.3.3 Thread Class

A thread in Java shows the program's path of execution. Logically, threading means successful execution of a program that shows the required output. Java programming is known as multi-threaded because implementation of Java threading and OS implementation are done accordingly with it. The Java version 1.1 supports green thread that is implemented in Linux OS. Green threads are simulated threads in Java virtual machine. Thread implementation differs as per operating systems. Sometimes, it is believed that Java Threads are really based on Solaris Thread. In essence, it is said that the size of thread local heap, thread stack, the garbage collection in Java thread are important factors that decide the implementation of Java. The thread of execution represents the class `Thread`. It is defined in public class `Thread`, extended to `Object` and implemented to `Runnable`. The thread class resides in the `java.lang` package. They create three integer (`int`) constants that are used to specify the priority of a thread. These are `MAX_PRIORITY`, `MIN_PRIORITY` and `NORM_PRIORITY`. The following Table 3.1 shows the various methods of thread class and their functions:

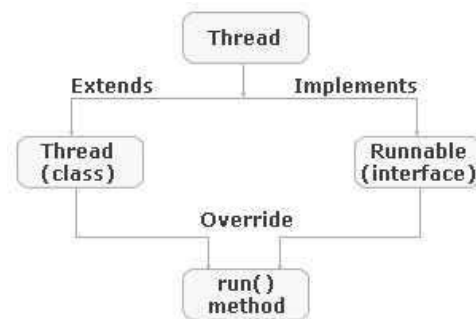
*Table 3.1 Methods used in Thread Class and Their Functions*

| Method                                                                   | Function                                                                       |
|--------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| <code>Thread.currentThread()</code>                                      | Returns a reference to the current thread.                                     |
| <code>Void sleep (long msec) throws InterruptedException</code>          | Causes the current thread to wait for msec milliseconds.                       |
| <code>void sleep(long msec, int nsec) throws InterruptedException</code> | Causes the current thread to wait for msec milliseconds plus nsec nanoseconds. |
| <code>void yield()</code>                                                | Causes the current thread to yield control of the processor to other threads.  |

The multiple threads run concurrently in Java application using Java Virtual Machine (JVM). Threads follow priority. The higher priority threads are executed first than the lower priority threads. The `Thread` keyword creates a new 'Thread' object during run-time of `Thread`. JVM executes thread in the following conditions:

- The class `Runtime` contains the `exit` method and the `exit` method is executed after getting permission from the security manager. Each thread can be marked as daemon or without except daemon.

All threads except daemon threads are dead either by returning from run method or by throwing an exception. A new thread can be executed if a declared class belongs to subclass of `Thread` which overrides run method of `Thread` class.



**Fig. 4.1** Hierarchy Set in Java Thread Class

## NOTES

## NOTES

The following are the two ways of creating a new thread of execution.

### Method I

Either class is to be declared as a subclass of thread. This subclass overrides the run method of class Thread. The subclass's instance is then started. The syntax of the creating class Thread by Method I is as follows:

```
import java.lang.*;
public class counter_Thread extends Thread
{
 public void run() {
 ...
 }
}
```

The following small program of thread computes primes larger than a stated value:

```
class Prime_Thread extends Thread
{
 long val_Prime;
 Prime_Thread(long val_Prime)
 {
 this.Prime_Thread = val_Prime;
 }
 public void run()
 {
 // Computes primes that are larger than minPrime
 ...
 }
}
```

The following code creates pthread:

```
Prime_Thread pthread = new Prime_Thread(21);
pthread.start();
```

### Method II

A class Thread can also be created by declaring a class that implements the Runnable interface. The syntax of the creating class Thread namely counter\_Thread by Method II is as follows:

```
import java.lang.*;
public class counter_Thread implements Runnable
{
 Thread tThread;
 public void run()
 {
 ...
 }
}
```

The above code creates a new class as counter Thread that extends the class Thread to override the Thread.run() method to its own implementation. This class implements the run method. The following code is able to create class by implements Runnable interface and run() method:

```
class Prime_Run implements Runnable
{
 long val_Prime;
```

```
Prime_Run(long val_Prime)
{
 this.val_Prime=val_Prime;
}
public void run()
{
 //Compute primes larger than val_Prime
 ...
}
}
```

The following code creates a thread to run the coding:

```
Prime_Run pRun=new Prime_Run(120);
new Thread(pRun).start();
```

### Creating a Thread

The path of program execution represents a thread. It causes a problem if actions or events occur simultaneously. For example, a program in Java does not draw the pictures while reading the keystrokes. Therefore, a coding part is written for a keyword that cannot handle more than one event at a certain time. The concurrent execution of more than one program is known as implementing an interface and extending a class. The `Runnable` interface forces to run the `run` method. The class `Thread` is also used to check the target class that cannot be equal to `NULL`. Then it is possible to target class to run its own method.

```
public class Thread implements Runnable
{
 ...
 public void run() {
 if (target!=NULL)
 {
 target.run();
 }
 }
 ...
}
```

Once a thread starts to invoke the `run()` method, the process automatically returns from this method and thread dies. The following code namely `'My_Thread'` results the subclass `Thread` that overrides `run()` and prints every 100 milliseconds for two seconds by using loops:

```
public class My_Thread extends Thread
{
 public void run()
 {
 int count_val=0;
 while (true)
 {
 System.out.println("Thread one alive");
 //Prints every 0.10 sec for two seconds
 }
 try
 {
 Thread.sleep(100);
 } catch (InterruptedException e)
 {
 count_val++;
 }
 if (count_val>=20) break;
 }
}
```

### NOTES

```

 }
 System.out.println ("ThreadStopping) ;
 }
 } //ClassMy_Thread

```

**NOTES**

In the above code, the sleep () method prints every 100 milliseconds of the content involved in 'for' loop.

The following code namely My\_Applet.java creates an instance of my\_thread. The start () method of My\_Applet.java program creates an instance of My\_Thread class to invoke the start method. The start method opens the browser page that initiates a thread.

```

public class My_Applet extends java.applet.Applet //Threading with
a Thread subclass
public void start () //Start () method starts a thread.
{
 MyThread m_thr = new MyThread (); //Creates an instance of
 m_thr.start ();
}

```

This coding creates a thread named as MyThread. The start () method launches the thread process.

Figure 3.2 shows how the main and thread processes communicate with each other and in result run in parallel.

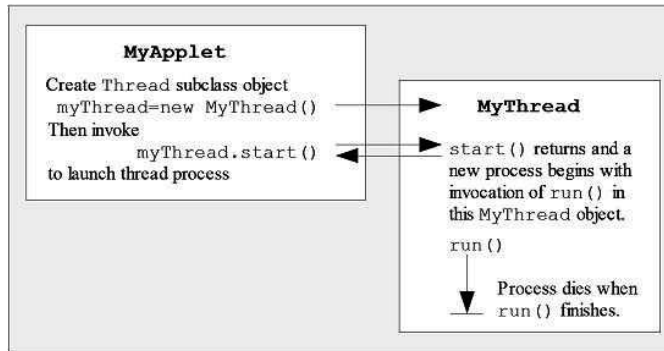


Fig. 3.2 Main and Thread run in Parallel

Once MyThread object is declared, it runs in parallel with the MyApplet thread. The run method executes the thread process till that method finishes and returns.

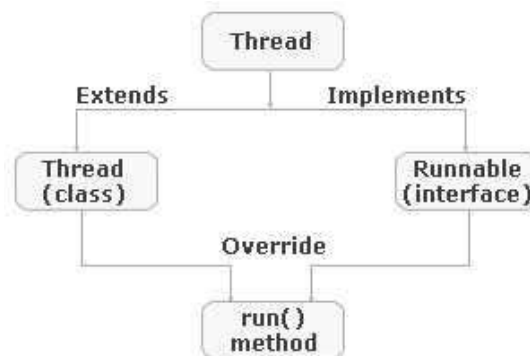


Fig. 3.3 Extending the java.lang.Thread Class



The following program shows a single thread creation that extends the 'Thread' class.

```
class ex_Thread extends Thread
//Extending the Thread class
{
 String str=null;
 //str variable is defined for String type
 ex_Thread(String str_str)
 //ex_Thread constructor is defined
 {
 str=str_str;
 //Assigning the str value as equal to str_str variable
 start();
 //Start () method causes the thread to begin execution
 }
 public void run() //Run method started as data type void
 {
 System.out.println(str);
 //Prints the 'str' value
 }
}
public class str_run_Thread
{
 public static void main (String args [])
 {
 ex_Thread x1=new ex_Thread ("Thread is started.....");
 //Calling the function that prints the message 'Thread is started.....'
 }
}
```

## NOTES

**Output of the program:**

```
C:\javafolder\thread>javac
str_run_Thread.java

C:\javafolder\thread>java str_run_Thread
Thread is started.....
```

### 3.3.4 Runnable Interface

The thread that is ready to run, but waiting for the processor availability is called the runnable thread and the state is known as Runnable state. All the threads in runnable states wait for execution and they can be executed on the basis of various scheduling techniques such as round robin or first-come first-serve, if all the threads have the same priorities. The `yield()` method controls all the runnable threads that have the same priorities. The following figure shows switching among various threads using the `yield()` method.

## NOTES

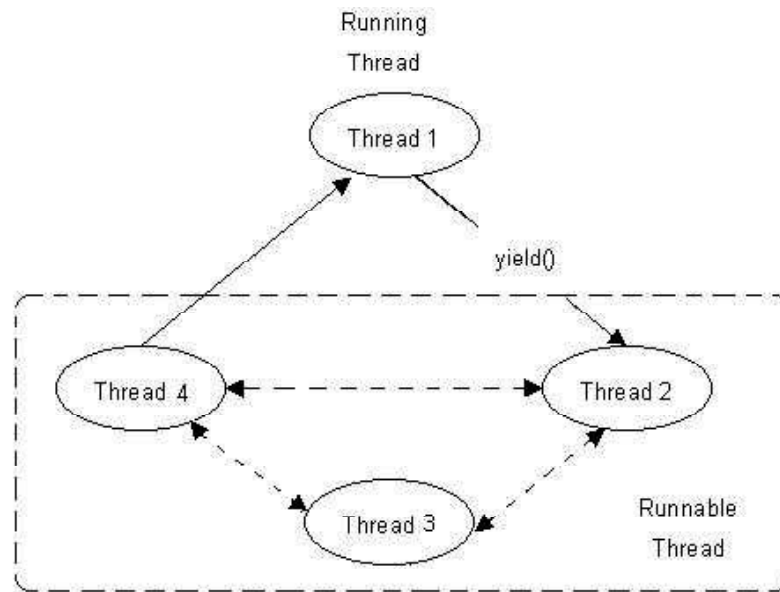


Fig. 3.4 Switching between Threads in Runnable State

### Implementing a Runnable Interface

You need to create a class that implements the runnable interface for creating a new thread. The runnable interface declares the `run()` method that is required for implementing a thread in the program. The steps to create a thread using the runnable interface are:

1. Create a class that implements a runnable interface. The syntax is:

```
class Class-Name implements Runnable
```

2. Implement the `run()` method. Define the code that constitutes the new thread inside this method. The syntax is:

```
public void run()
{
 // Threadbody
}
```

3. Instantiate an object of the thread class within the class to create a new thread. The syntax is:

```
Thread object = new Thread (Runnablethread-object, Stringthread-
name);
```

4. Call the `start()` method to run the thread. The syntax is:

The following program code creates a multithreaded Java program by implementing the runnable interface:

Creating a Thread by implementing runnable Interface

```
class A implements Runnable
{
 public void run()
 {
 for (int i=1; i<=5; i++)
 {
 System.out.println("\tFrom ThreadA: i=" + i);
 }
 }
}
```

## NOTES

```
}
System.out.println("Exit fromA");
}
}
class B implements Runnable
{
public void run ()
{
for (int j=1; j<=5; j++)
{
System.out.println("\tFrom ThreadB: j="+j);
}
System.out.println("Exit fromB");
}
}
class C implements Runnable
{
public void run ()
{
for (int k=1; k<=5; k++)
{
System.out.println("\tFrom ThreadC: k="+k);
}
System.out.println("Exit fromC");
}
}
class Thread_Creation
{
public static void main (String arg[])
{
A Thread_A=new A ();
B Thread_B=new B ();
C Thread_C=new C ();
Thread threadA=new Thread (Thread_A);
Thread threadB=new Thread (Thread_B);
Thread threadC=new Thread (Thread_C);
threadA.start ();
threadB.start ();
threadC.start ();
System.out.println("Exit the main thread.");
}
}
```

This code creates three new threads, A, B and C by implementing the Runnable interface. The main () method consists of the instances of the A, B and C classes. These instances are passed as the argument value to the thread class for creating its objects such as ThreadA, ThreadB and ThreadC, respectively. The three threads are started by the start () method.

## NOTES

The following is the output.

```
C:\java\Chapter5>javacThread_Creation.java
C:\java\Chapter5>javaThread_Creation
Exitthemainthread.
 FromThreadA:i=1
 FromThreadA:i=2
 FromThreadA:i=3
 FromThreadB:j=1
 FromThreadC:k=1
 FromThreadA:i=4
 FromThreadB:j=2
 FromThreadC:k=2
 FromThreadA:i=5
 FromThreadB:j=3
 FromThreadC:k=3
ExitfromA
 FromThreadB:j=4
 FromThreadC:k=4
 FromThreadB:j=5
 FromThreadC:k=5
ExitfromB
ExitfromC
```

### Extending the Thread Class

You can create a thread class by extending a thread superclass and this thread is implemented by creating its instances. The thread class is the member of the `java.lang.Thread` package of Java. The three steps to create a new thread using the thread class are:

1. Declare the class that extends the properties and methods of the thread superclass. The syntax to declare a thread class is:

```
classclass-nameextendsThread
{
//Classbody
}
```

2. Override the `run()` method of the extending class. The `run()` method is the entry point for the thread, because all the threads are started through this `run()` method. The syntax is;

```
publicvoidrun()
{
//Threadbody
}
```

3. Create an object of the thread class that extends the thread super class. The thread object is created and it is called the newborn thread. The syntax for creating a thread object is:

```
class-nameobject=newclass-name();
```

4. Start the new thread by calling the `start()` method. The syntax is:

```
object.start();
```

The created thread is moved from the newborn state to the Runnable state using the `start()` method.

The following program code creates threads by extending the thread class:

### Creating Threads using the Thread Class

```
classAextends Thread
{
publicvoidrun ()
{
for (inti=1;i<=5;i++)
{
System.out.println("\tFromThreadA: i="+i);
}
System.out.println("Exit fromA");
}
}
classBextends Thread
{
publicvoidrun ()
{
for (intj=1;j<=5;j++)
{
System.out.println("\tFromThreadB: j="+j);
}
System.out.println("Exit fromB");
}
}
classCextends Thread
{
publicvoidrun ()
{
for (intk=1;k<=5;k++)
{
System.out.println("\tFromThreadC: k="+k);
}
System.out.println("Exit fromC");
}
}
classThreadCreation
{
publicstaticvoidmain (Stringarg[])
{
newA().start();
newB().start();
newC().start();
}
}
```

This code creates three new threads such as A, B and C by extending the Thread class. Each class overrides the run () method to define a thread. You have to start these threads by using the start () method of the thread class after creating the threads. The output of the above code changes on each run. All the threads have the same priorities and they may start in any order.

The following is the output.

```
C:\java\Chapter5>javac ThreadCreation.java
C:\java\Chapter5>java ThreadCreation
 FromThreadA: i=1
 FromThreadA: i=2
```

### NOTES

## NOTES

```
FromThreadA : i = 3
FromThreadA : i = 4
FromThreadA : i = 5
FromThreadB : j = 1
Exit from A
FromThreadB : i = 2
FromThreadC : k = 1
FromThreadB : j = 3
FromThreadC : k = 2
FromThreadB : j = 4
FromThreadC : k = 3
FromThreadB : j = 5
FromThreadC : k = 4
Exit from B
FromThreadC : k = 5
Exit from C
```

### 3.3.5 Creating Multiple Threads

Thread methods are those methods that control the different types of behaviour, such as running, sleeping or resuming of the various threads. Thread class is the member of java.lang.thread package of Java. The thread class of Java provides various types of thread methods:

- **run () Method:** Executes the code that is enclosed within the run () method. The thread class overrides the run () method that extends the thread class or implements the runnable interface. The syntax is:

```
public void run ()
{
// Thread body
}
```

- **start () Method:** Initializes the thread and calls the run method. Starts the execution of a newborn thread that is created by the run () method of the thread class. The start () method allocates the system resources that are required for the thread execution. The syntax is:

```
thread-object.start ();
```

- **yield () Method:** Relinquishes between the threads that are in runnable state and have the same priorities. The yield () method changes the execution of the thread on the basis of round robin or first-come first-serve scheduling techniques. The syntax is:

```
yield ();
```

- **stop () Method:** Destroys a running or suspended thread. This method is deprecated in Java 2. You need to use the exception handler with this method or the interrupt () method instead of the stop () method. The syntax is:

```
stop ();
```

- **sleep () Method:** Suspends the currently running thread for a specified amount of time. This method does not destroy the thread, rather it suspends the thread for a specified time. This method throws InterruptedException, as a result, you use the sleep () method in between the try-catch block.

```
sleep (t);
```

Here, `t` is the specified amount of time in millisecond.

- **suspend () Method:** Stops the running thread, until it is again resumed by the `resume ()` method of the thread class. The syntax is:

```
suspend();
 //After certain time
resume();
```

- **wait () Method:** Stops the currently running thread, until some event occurs using the `notify ()` method of the thread class. The syntax is:

```
wait();
 //After sometime
notify(); //When any event is occurred.
```

The following program code shows the threads that use the various methods of the thread class:

#### Using Thread Methods

```
class A extends Thread
{
 public void run ()
 {
 for (int i=1; i<=5; i++)
 {
 if (i==1)
 yield();
 System.out.println("\tFrom Thread A: i="+i);
 }
 System.out.println("Exit from A");
 }
}
class B extends Thread
{
 public void run ()
 {
 for (int j=1; j<=5; j++)
 {
 System.out.println("\tFrom Thread B: j="+j);
 if (j==3)
 stop();
 }
 System.out.println("Exit from B");
 }
}
class C extends Thread
{
 public void run ()
 {
 for (int k=1; k<=5; k++)
 {
 System.out.println("\tFrom Thread C: k="+k);
 if (k==1)
 try
 {
 sleep(1000);
 } catch (Exception e) {}
 }
 }
}
```

## NOTES

## NOTES

```
System.out.println("Exit fromC");
}
}
class ThreadMethods
{
public static void main (String arg[])
{
Athread_a=new A();
Bthread_b=new B();
Cthread_c=new C();
System.out.println("Exit fromA");
thread_a.start();
System.out.println("Exit fromB");
thread_b.start();
System.out.println("Exit fromC");
thread_c.start();
System.out.println("Exit the main thread.");
}
}
```

This code shows a multithreaded Java program that uses the various thread methods. In this code, the `yield()` method is invoked, when Thread A is at the iteration `i = 1`. Thread A starts first, but it is relinquished and gives up its control to Thread B. The iteration of Thread B reaches `j = 3` and is destroyed by the `stop()` method. The iteration of Thread C reaches `k = 1`, Thread C is suspended for 1 second and after one second, it is automatically resumed.

The following is the output.

```
C:\java\Chapter5>java ThreadMethods
Exit fromA
Exit fromB
Exit fromC
Exit the main thread
 From Thread C : k = 1
 From Thread B : j = 1
 From Thread B : j = 2
 From Thread B : j = 3
 From Thread A : i = 1
 From Thread A : i = 2
 From Thread A : i = 3
 From Thread A : i = 4
 From Thread A : i = 5
Exit fromA
 From Thread C : k = 2
 From Thread C : k = 3
 From Thread C : k = 4
 From Thread C : k = 5
Exit fromC
```

### 3.3.6 Suspending, Resuming and Stopping Threads

As mentioned earlier, the predefined methods `suspend()`, `resume()` and `stop()` have been deprecated in Java 2 though they are a convenient way for managing the execution of threads. These methods were deprecated as they may cause deadlocks and serious system failures in a multithreaded environment.



However, in the new version of Java, suspending, resuming and stopping a thread can be performed using `boolean` type flags. The `run ()` method of a thread will work based on the current flag values that indicate the execution state of a thread. For example, if the running flag is set to `true`, the `run ()` method must let the thread execute. For the `run ()` method to suspend the execution of the currently running thread, the suspend flag must be set to `true`. Likewise, the thread will die once the stop flag is set to `true` (Refer Example 3.9).

**Example 3.9:** A program to demonstrate suspend, resume and stop operations is as follows:

```
class ChildThread extends Thread
{
 boolean suspend_flag, stop_flag;
 String name;
 ChildThread (String str)
 {
 name = str;
 suspend_flag = false;
 stop_flag = false;
 }
 public void run ()
 {
 try
 {
 int i = 5;
 while (i >= 1)
 {
 System.out.println (name + " " + i);
 sleep (1000);
 i--;
 synchronized (this)
 {
 while (suspend_flag)
 {
 wait ();
 if (stop_flag)
 {
 break;
 }
 }
 }
 }
 } catch (InterruptedException e)
 {
 System.out.println ("Thread interrupted");
 }
 }
 synchronized void my_suspend ()
 {
 suspend_flag = true;
 }
 synchronized void my_resume ()
 {
 suspend_flag = false;
 notify ();
 }
}
```

## NOTES

## NOTES

```
 }
 synchronized void my_stop()
 {
 suspend_flag=false;
 stop_flag=true;
 notify();
 }
}
class SRS
{
 public static void main(String[] args)
 {
 try
 {
 ChildThread obj=new ChildThread("Thread");
 obj.start();
 System.out.println("Thread started");
 Thread.sleep(2000);
 obj.my_suspend();
 System.out.println("Thread is suspended");
 Thread.sleep(2000);
 obj.my_resume();
 System.out.println("Thread is resumed");
 Thread.sleep(2000);
 obj.my_suspend();
 System.out.println("Thread is suspended");
 Thread.sleep(2000);
 obj.my_resume();
 System.out.println("Thread is resumed");
 Thread.sleep(2000);
 obj.my_stop();
 System.out.println("Thread stopped");
 }
 catch (InterruptedException e)
 {
 System.out.println("Thread interrupted");
 }
 }
}
```

### The output of the program is:

```
Thread started
Thread 5
Thread 4
Thread 3
Thread is suspended
Thread is resumed
Thread 2
Thread 1
Thread is suspended
Thread is resumed
Thread stopped
```

### Check Your Progress

1. How will you create thread in java?
2. Why threads are called lightweight processes?
3. What are the constants defined in thread class?
4. Write the default priority of a thread.
5. What are the two ways to implement synchronization?
6. Define the term runnable interface.

### NOTES

## 3.4 BASIC INPUT/OUTPUT

Java manages all input and output in the form of streams. A stream refers to a channel through which data flows from the source to the destination. This data is in the form of sequence of bytes or characters.

Java streams can be broadly categorized into two types, namely input stream and output stream. The streams which help to read data from various sources, namely keyboard, mouse, files, storage devices, etc., in order to supply it to the program are called input streams. The streams which receive data from the program and directly write it to the physical devices or other programs are called output streams. For example, to bring the data from an input device into the program, the program opens an input stream on the input device and reads the data in a serial manner. Conversely, to write data from the program to an output device, the program opens an output stream to the output device and writes data to it serially (Refer Figure 3.5).

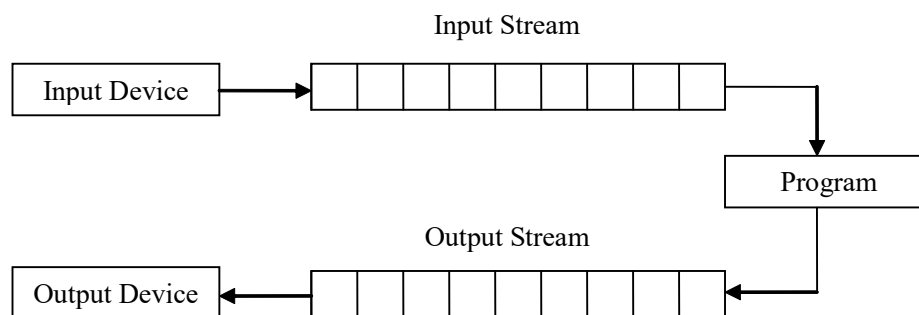


Fig. 3.5 Java Streams

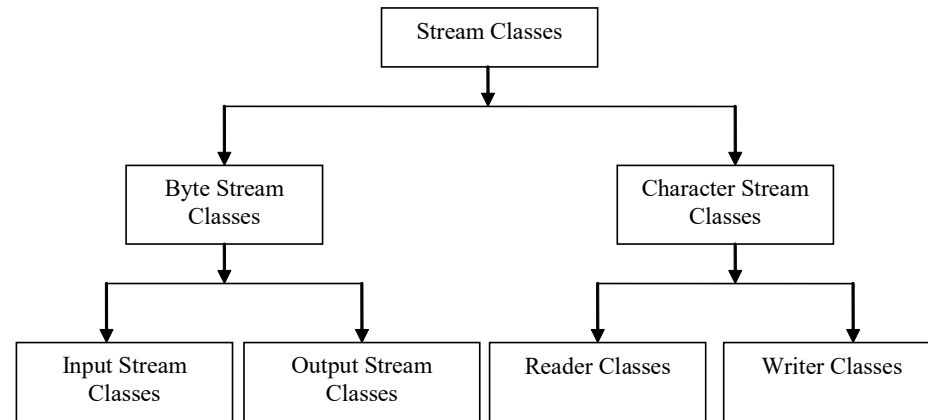
### 3.4.1 Streams (Byte and Character)

Java supports input/output streams through a hierarchy of classes defined in the `java.io` package. On the basis of the type of data on which these classes operate, they can be categorized into two groups: byte stream classes and character stream classes (Refer Figure 3.6).

- **Byte Stream Classes:** These classes support input and output operations on bytes (8-bit bytes). For example, while reading from or writing data to a binary file, byte stream classes are used.

- **Character Stream Classes:** These classes perform input and output operations on characters (16-bit Unicode). For example, while reading from or writing data to a text file, character stream classes are used.

## NOTES



*Fig. 3.6 Classification of Stream Classes*

### 3.4.2 Reading From and Writing To Console

Java 1.0 supports console input using byte streams. Since the use of byte streams for reading/writing console input requires using deprecated methods, this approach is not recommended. The more desired approach used for reading input for Java 2 is to use character streams instead of byte streams. Java 1.1 and higher versions of Java provide `InputStreamReader` to convert byte-oriented data into character-oriented data. In order to read data from the console using character stream, `System.in` is used. `System.in` refers to an object of type `InputStream`, therefore, it can be used as a character-based input stream.

Example 3.10 illustrates the use of `InputStreamReader` class. The statement to wrap `InputStreamReader` inside the `BufferedReader` and create a character-based stream connected to the console through `System.in` is as follows:

```
BufferedReader br = new BufferedReader (new InputStreamReader
(System.in));
```

**Example 3.10:** A program to demonstrate the use of `InputStreamReader` class is as follows:

```
import java.io.*;
class InputStreamReaderExample
{
 public static void main (String args[])
 {
 try
 {
 int j[]=new int[5];
 double sum=0.0;
 System.out.print("Enter your name: ");
 BufferedReader br=new BufferedReader (new InputStreamReader
 (System.in));
```

```
//reading input from the keyboard
 String name=br.readLine();
System.out.println("Enter marks you scored in five subjects");
 for (int i=0; i<j.length; i++)
 {
 System.out.print((i+1)+": ");
 String s=br.readLine();
 j[i]=Integer.parseInt(s);
 sum+=j[i];
 }
System.out.println(name+" your percentage is: "+sum/j.length+"%");

 }
 catch (Exception e) {}
}
}
```

### **The output of the program is:**

```
Enter your name: Kevin
Enter marks you scored in five subjects
1: 89
2: 90
3: 65
4: 78
5: 60
Kevin your percentage is: 76.4%
```

## **3.4.3 Reading and Writing Files**

### **FileReader and FileWriter Classes**

The `FileReader` class is used to read characters from the file. The `FileReader` class creates character stream between the file and the program and reads characters from the file and sends it to the program. Similarly, to write characters to a file, the `FileWriter` class is used. Example 3.11 illustrates the use of `FileReader` and `FileWriter` class.

**Example 3.11:** A program to demonstrate the use of `FileReader` and `FileWriter` class for reading from and writing characters to a file is as follows:

```
import java.io.*;
class ReadWriteFile
{
 public static void main (String args[]) throws IOException
 {
 String w="Hello\nHow\nare\nyou";
 FileWriter fw=new FileWriter("data.txt");
 System.out.println("Writing to the file data.txt...");
 fw.write(w);
 System.out.println("Writing complete");
 fw.close();

 System.out.println();
 FileReader fr=new FileReader("data.txt");
 BufferedReader b=new BufferedReader(fr);
```

## **NOTES**

## NOTES

```
 System.out.println("Reading the file data.txt...");
 while((w=b.readLine())!=null)
 {
 System.out.println(w);
 }
 fr.close();
 System.out.println("Reading ends");
 }
}
```

### The output of the program is:

```
Writing to the file data.txt...
Writing complete
Reading the file data.txt...
Hello
How
are
you
Reading ends
```

In the program illustrated in Example 3.11, when the `FileWriter` class' object is created, a file with the name `data.txt` is automatically created by the operating system and data is written to it. Moreover, the object of `FileReader` class is used to read the contents of the same file.

### File Class

The `File` class allows you to obtain and manipulate the information about a file like permissions, size, time, etc. Unlike other classes of `java.io` package, this class does not operate on the streams; it deals directly with the files and file system. That is, it does not specify how the data is retrieved from or sent to the files. Using this class, you can also make new directories, rename as well as delete the files.

A `File` object can be created using any one of the following constructors:

```
File(String path)
File(String path, String filename)
File(File dir, String filename)
```

where,

`path` is the path name of the file.

`filename` is the name of the file or subdirectory.

`dir` is a `File` object that specifies a directory.

The `File` class defines various methods. Some of them along with their description are listed in Table 3.2. Also, Example 3.12 illustrates the use of some of the methods of `File` class.

**Table 3.2** Some File Class Methods

| Method                  | Description                                                                     |
|-------------------------|---------------------------------------------------------------------------------|
| boolean canExecute()    | It returns true if the file given by the abstract path name can be executed.    |
| boolean canRead()       | It returns true if the file given by the abstract path name can be read.        |
| boolean canWrite()      | It returns true if the file given by the abstract path name can be written.     |
| boolean createNewFile() | It creates a new empty file.                                                    |
| boolean delete()        | It deletes the file or directory; directory can be deleted only if it is empty. |
| boolean exists()        | It returns true if the file given by the abstract pathname exists.              |
| String getName()        | It returns the name of the file or directory.                                   |
| String getParent()      | It returns the name of the parent directory.                                    |
| String getPath()        | It returns the path of the file as a string.                                    |
| boolean isAbsolute()    | It returns true if the abstract file path name is absolute.                     |
| boolean isFile()        | It returns true if invoked on a file and false if invoked on a directory.       |
| boolean isHidden()      | It returns true if the file is a hidden file.                                   |
| long lastModified()     | It returns the time the file was last modified.                                 |
| long length()           | It returns the size of the file.                                                |
| boolean isDirectory()   | It returns true if the file given by the abstract path name is directory.       |
| boolean setReadOnly()   | It sets the file to read-only .                                                 |

**Note:** In Java, a directory is also treated as file. The only difference is that it contains a list of file names which can be obtained using the `list()` method.

**Example 3.12:** A program to demonstrate the use of some of the methods of File class is as follows:

```
import java.io.*;
public class FileClassExample
{
 public static void main (String args [])
 {
 InputStreamReader cin = new InputStreamReader (System.in);
 BufferedReader br = new BufferedReader (cin);
 File fileobj = new File ("Java/Sample.txt");

 if (!fileobj.exists ())
 {
 System.out.println ("File does not exist.");
 System.exit (0);
 }
 System.out.println ("FileName: " + fileobj.getName ());
 System.out.println ("Path: " + fileobj.getPath ());
 System.out.println ("AbsolutePath: " +
fileobj.getAbsolutePath ());
 System.out.println ("Parent: " + fileobj.getParent ());
 System.out.println ("File was last modified at: " +
fileobj.lastModified ());
 System.out.println ("File size: " + fileobj.length () + "
Bytes");
 }
}
```

## NOTES

## NOTES

### The output of the program is:

```
FileName: Sample.txt
Path: Java\Sample.txt
AbsolutePath: E:\JAVA_Book\Java\Sample.txt
Parent: Java
Filewaslastmodifiedat: 1264808454781
Filesize: 31Bytes
```

### 3.4.4 PrintWriter Class

When a program is executed, the input is read from various sources and the output is sent to different destinations. Generally, the keyboard and monitor are used as standard input and output devices, respectively. The data fed by the user is supplied to the program using input streams and the output of the program is supplied to the output device using output streams. This output is displayed to the user using Java `PrintStream` class. Therefore, the two methods, namely `print()` and `println()` that are used for displaying the primitive data type, object, etc. on an output device are defined by the `PrintStream` class. This is a byte stream class which is derived from the `OutputStream` class. Unlike other output streams, the `PrintStream` class does not throw an `IOException` even if an exceptional event occurs.

`PrintStream` class can be connected to the underlying output streams, such as `FileOutputStream`, `ByteArrayOutputStream`, `BufferedOutputStream`, etc.

`PrintStream` class defines following type of constructor:

```
PrintStream(OutputStream os)
```

This constructor creates a `PrintStream` and connects it to the output stream `os`.

In addition to `print()` and `println()` methods, the `PrintStream` class defines some other methods which are listed in Table 3.3.

*Table 3.3 PrintStream Class Methods*

| Method                                           | Description                                                         |
|--------------------------------------------------|---------------------------------------------------------------------|
| <code>PrintStream append(char c)</code>          | Appends the character <code>c</code> to the output stream.          |
| <code>PrintStream append(CharSequence cs)</code> | Appends the character sequence <code>c</code> to the output stream. |
| <code>boolean checkError()</code>                | Checks the error state of the stream.                               |
| <code>protected void setError()</code>           | Sets the error state to <code>True</code> .                         |

**Program 3.1:** A program to demonstrate the use of `PrintStream` class.

```
import java.io.*;
class PrintStreamExample
{
 public static void main (String args[])
 {
 try
 {
```



```
byte b[] = "Java PrintStream class is used to
display result to the user".getBytes();
 CharSequence c = "the end";
FileOutputStream fos = new FileOutputStream
("test.txt"); //creating file output stream
/*creating print stream that wraps file output
stream*/
PrintStream ps = new PrintStream(fos);
ps.write(b); //writing data to the stream
ps.append(c); /*appending data to the print
stream*/
if (ps.checkError()) //checking error state
{
 System.out.println("Error in appending
characters");
}
else
{
 System.out.println("Data written to the
file");
 System.out.println("Character appended
successfully");
 }
 ps.close(); //closing the stream
}
catch (IOException e)
{
 System.out.println(e.getMessage());
}
}
```

### **Output of the program:**

```
Data written to the file
Character appended successfully
```

---

## **3.5 FUNDAMENTALS OF APPLETS**

---

An applet is a small program typically embedded within the Web page which is used to create a dynamic and interactive application. They provide interactive features to a Web page which cannot be provided by HTML (HyperText Markup Language). For example, applets enable capturing user inputs in the form of mouse clicks, text entry, checkbox selection, etc. and generating response to the user's actions.

Each applet that is created must be a subclass of the `Applet` class, contained within the `java.applet` package. This class contains methods which govern the life and behavior of the applets. In addition, applets use various methods of the `Graphics` class contained inside the `java.awt` package. The `Graphics` class is responsible for all the operations related to display (output) of an applet. The Java applets can be executed either through an appletviewer (a tool of Java Development Kit) JDK or any Java-compatible Web browser.

### **NOTES**

## NOTES

### Life Cycle of an Applet

A Java applet enters into various states during its entire life cycle which include born state, running state, idle state and dead state. These states occur when different methods of the `Applet` class are invoked by the Java runtime system. The invocation of these methods makes an applet undergo a series of state change right from the time it is loaded till it is destroyed and frees all the resources held by it. The life cycle of an applet can be depicted as shown in Figure 3.7.

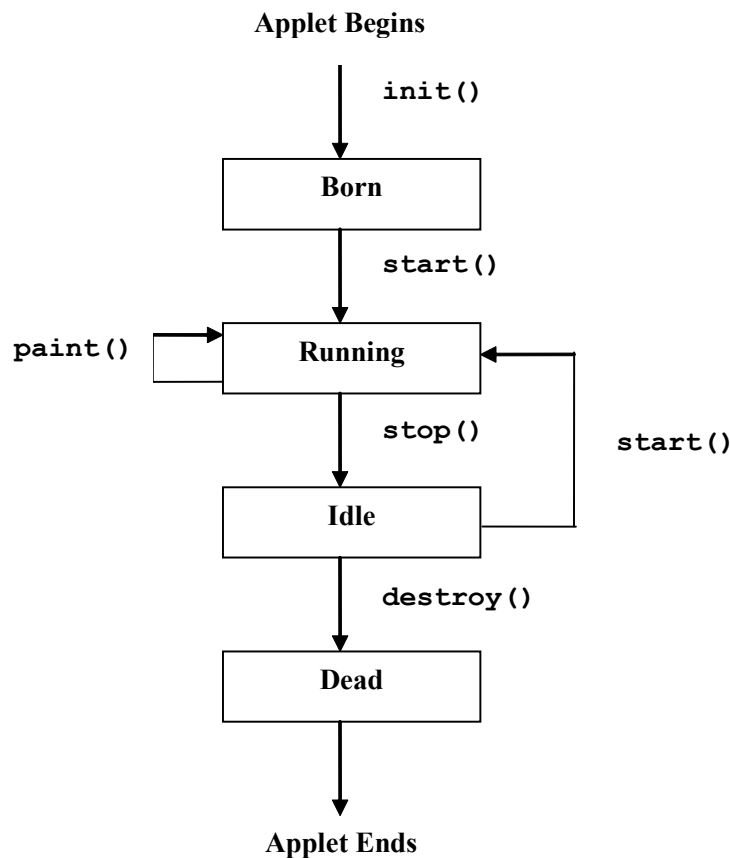


Fig. 3.7 Applet Life Cycle

The order of method invocation when an applet is loaded is as follows:

- **`init()`**: The life cycle of an applet begins when it is first loaded and the `init()` method is invoked. This method is invoked only once during the entire lifetime of an applet. The body of the method includes statements related to variable initialization, object creation, adding components like buttons, textboxes, etc., setting colors of the applet, loading of images or fonts, etc. After the invocation of the `init()` method, the applet enters the *born* state.
- **`start()`**: This method is automatically invoked after the `init()` method. Unlike `init()` method, `start()` method may be invoked more than once. This method is called every time the Web page containing the applet is executed and displayed on the screen. With the invocation of this method, the applet enters the running state.

- **paint ()** : This method is invoked to display the output on the screen in the form of text, graphics, etc. Sometimes, it may happen that the window in which applet is running is covered by another window, or is minimized or resized. In all these cases, `paint ()` method is re-invoked for the output to be redrawn on the screen. As depicted in Figure 3.7 the applet remains in the running state while `paint ()` method is invoked.

The order of method invocation at the time of termination of an applet is as follows:

- **stop ()** : This method is invoked automatically when the Web page containing the running applet is closed or left temporarily and applet enters the idle state. The user can also stop the running applet by invoking the `stop ()` method explicitly.
- **destroy ()** : This method is invoked to remove the applet permanently from the memory. It releases all the resources held by the applet. Like `init ()` method, this method is invoked only once during the entire life cycle of an applet. The applet becomes dead when this method is invoked.

*Note:* The `init ()` and `destroy ()` methods are called only once whereas `start ()`, `paint ()` and `stop ()` methods can be called multiple times in an applet.

### 3.5.1 Transient and Volatile Modifier

The transient and volatile modifiers are two special modifiers provided by Java which are used to handle some specialized situations.

#### **transient**

Object serialization is the process of reading and writing objects. By default, all objects are serializable, i.e., they can be read from and written to the secondary memory so that the value which they hold persists. To make an object non-serializable, the `transient` modifier is used. If an instance variable is declared as transient then the values of that variable will not persist while writing its object to the secondary memory.

To understand the concept of transient keyword, consider the following code segment:

```
class TransientExample
{
 transient double first; // will not persist
 double second; //will persist
}
```

Here, when the object of class `TransientExample` is written to the secondary memory, the values of `first` will not be saved while the value of `second` will be.

#### **volatile**

The `volatile` modifier is used to tell the compiler that the variable declared as `volatile` can be changed at any time by the other parts of the program. This modifier is mainly used in multithreading in which a program (process) is divided into two or more subprograms (subprocesses), each of which runs by a separate

## NOTES

## NOTES

thread and performs different tasks concurrently. In a multithreaded program, various threads share the same instance variable and keep their own copy of the variable in their local cache memory while the master copy remains in the main memory. Whenever a thread changes the value of the variable, it updates the value of the variable only in its local cache memory and not in the main memory. This leads to inconsistency because the thread using the same variable does not know about the change of the value by another thread. So to avoid this problem, the `volatile` modifier is used. When a variable is declared as `volatile`, it is not stored in the cache memory and its value is updated in the main memory so that the other threads can easily access the updated value.

### 3.5.2 Modifier `Strictfp`

In the Java programming language, the `strictfp` is a modifier that restricts floating point calculations to ensure portability. The `strictfp` command was introduced into Java with the Java Virtual Machine (JVM) version 1.2 and is now available for use on all currently updated Java VMs.

The IEEE (Institute of Electrical and Electronics Engineers) standard IEEE 754 specifies a standard method for both floating point calculations and storage of floating point values in various formats, including single (32-bit, used in Java's `float`) or double (64-bit, used in Java's `double`) precision.

When the overflow or underflow is lacking, then there is no difference in results with or without `strictfp`. But when the repeatability is essential, then the `strictfp` modifier is typically used to ensure that overflow and underflow occurs in the same places on all the platforms. Without the `strictfp` modifier, intermediate results may use a larger exponent range. The `strictfp` modifier accomplishes this by representing all intermediate values as IEEE single precision and double precision values, as occurred in earlier versions of the JVM.

Java programmers use the modifier `strictfp` to ensure that calculations are performed as it was in the earlier versions, i.e., only with IEEE single and double precision types used. Additionally, using `strictfp` guarantees that results of floating point calculations are identical on all the platforms.

The modifier `strictfp` can be used on classes, interfaces and non-abstract methods. When applied to a method, it causes all calculations inside the method to use strict floating point mathematics. When applied to a class, all calculations inside the class use strict floating point mathematics. Compile time constant expressions must always use strict floating point behaviour.

Following is the syntax example of `strictfp` modifier:

```
public strictfp class MyFPclass {
 // ... contents of class here ...
}
```

Following are some examples of `strictfp` modifier.

**Example 1:** Keyword `strictfp` modifier usage with classes.

```
strictfp class Test {

 // All concrete methods here are implicitly strictfp.
}
```

**Example 2:** Keyword **strictfp** modifier usage with interfaces.

```
strictfp interface Test {

 // All methods here becomes implicitly
 // strictfp when used during inheritance.
}

class Car {

 // strictfp applied on a concrete method
 strictfp void calculateSpeed() {}
}
```

**Example 3:** Keyword **strictfp** modifier usage with variables.

```
strictfp interface Test {
 double sum();

 // Compile-time error here
 strictfp double mul();
}
```

From the above given example codes, we can define the following conclusions:

- When a class or an interface is declared with **strictfp** modifier, then all methods declared in the class or interface, and all nested types declared in the class, are implicitly **strictfp**.
- The **strictfp** cannot be used with abstract methods. Though, it can be used with abstract classes or interfaces.
- Because the methods of an interface are implicitly abstract, therefore **strictfp** cannot be used with any method inside an interface.

### 3.5.3 Native Interface

In Java, `native` is a modifier. The `native` modifier can only refer to methods. Like the `abstract` keyword, `native` indicates that the body of a method is to be found elsewhere. In the case of abstract methods, the body is in a sub-class, but in the case of `native` methods, the body lies entirely in a library which is outside the JVM. People who port Java to a new platform, implement extensive `native` code to support GUI components, network communication and a broad range of platform specific functionalities. However, it is rare for applet and application programmers to use the `native` codes. One of the vital features of **JNI** (Java Native Interface) is that it never imposes any restriction on the JVM. Therefore, JVM (Java Virtual Machine) vendors can add support for the **JNI** without affecting other parts of the virtual machine. A Java programmer may need the `native` method in the following situations:

- If the client requires platform-dependent features in Java, but the standard Java class library supports platform-independent features. In this situation, the programmer requires JNI for supporting platform-dependent features in Java.
- **JNI** has to be used when the programmer has already developed the code in C and C++ and wishes to make it accessible to Java code.

## NOTES

## NOTES

- **JNI** must be used when the programmer wants to develop a small portion of time-critical code by using a lower-level language, such as assembly.

By programming through **JNI**, the programmer uses `native` methods to:

- Create, inspect, and update Java objects.
- Call Java methods.
- Catch and throw Exception.
- Load classes and collect class information.
- Perform runtime type checking.

**JNI** can be used with **Invocation API** to embed any `native` application in JVM.

Drawback of `native` code is that it violates the platform independent features of Java and this procedure has less security as binary files are generated during `native` implementation.

**JNI** allows Java byte codes to communicate with foreign methods like C/C++. Advantage of such a technique is that byte codes are able to communicate with executable files, which execute faster and hence increase the performance. In Java, `native` code or `native` method accesses JVM features by calling **JNI** functions. In Java, **JNI** functions are available by using an interface pointer. An interface pointer is a pointer to a pointer to a structure. This structure pointer points to a set of pointers to function which is inside the structure of **JNI**. The following example clarifies this concept.

### Program 3.2

```
//Java - C Communication
public class p
{
 int i;
 public static void main (String args[])
 {
 p x=new p ();
 x.i=10;
 System.out.println(x.i);
 x.fun ();
 //Native method calling.
 System.out.println(x.i);
 }
 public native void fun ();
 //Native method declaration.
 static{
 System.loadLibrary("p");
 }
}
```

**Step 1.** Compile it with `javac p.java`

This will generate the class file as `p.class`

**Step 2.** Create the header file by the command `javah p`

This will generate the header file `p.h`.

If the programmer wants to see the header file then the programmer must write `edit p.h` at the Command prompt. The following header file will be displayed.

```
p.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include<jni.h>
/* Header for class p */
#ifndef _Included_p
#define _Included_p
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: p
 * Method: fun
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_p_fun
 (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

The above two steps are same for Windows and Linux.

### Steps For Linux Platform

**Step 3.** Create the `.c` file `p.c`

**Note :** For Linux the C file name should be same as that of the java file name, otherwise run time error will arise.

```
p.c
#include"p.h"
#include"jni.h"
#include"stdio.h"
JNIEXPORT void JNICALL Java_p_fun
 (JNIEnv *env, jobject obj)
{
 jclass cls;
 jfieldID fid;
 jint i;
 printf("Hello");
 cls=(*env)->GetObjectClass(env,obj);
 fid=(*env)->GetFieldID(env,cls,"i","I");
```

## NOTES

## NOTES

```
i=(*env)->GetIntField(env,obj, fid);
printf("%d",i);
}
```

**Step 4.** Create the object file by:

```
gcc -O -fpic -c p.c
```

**Note:** In case one is using multithreading in the programs, then `D_REENTRANT` attribute has to be used along with the command to create the object file.

```
gcc -O -D_REENTRANT -fpic -c p.c
```

**Step 5.** Create the `libp.so`(shared library) file by:

```
gcc -shared -o libp.so p.o
```

**Step 6.** Get the output by:

```
java -Djava.library.path=. p
```

### Steps for Windows Platform

**Step 3.** `CL/LD p.c`

To create dynamic linking library:

```
javap
```

To execute Java program:

The `p.Java` file is a usual Java file. Header file `p.h` is generated by JVM, it contains the prototype declaration of the `native` method. `native` methods are loaded when the programmer calls a `static` method of system class.

```
public static void loadLibrary()
JNIEXPORT void JNICALL Java_p_fun
(JNIEnv *, jobject)
```

Signature: `V()` in the comment indicates that the return type of `native` method is `void`. The signatures of other return types are listed below:

**Table 3.4** Signature of Different Data Types

| Type Signature            | Java Type             |
|---------------------------|-----------------------|
| Z                         | boolean               |
| B                         | byte                  |
| C                         | char                  |
| S                         | short                 |
| I                         | int                   |
| J                         | long                  |
| F                         | float                 |
| D                         | double                |
| L fully-qualified-class ; | fully-qualified-class |
| [ type                    | type[]                |
| ( arg-types ) ret-type    | method type           |



Consider the `p.c` file. It takes the argument `JNIEnv *env` and `jobject obj`.

```
typedef const struct JNINativeInterface *JNIEnv;
```

**JNIEnv** is a pointer to the structure **JNINativeInterface** which contains function pointers. This concept can be understood through an example. Consider the C program below:

### Program 3.3

```
void fun ()
{
 printf("HelloWorld");
}
struct xxx
{
 void(*p) ();
};
typedef struct xxx *struct_ptr;
int main()
{
 struct_ptr *pointer;
 struct xxx a,*ptr1;
 a.p=&fun;
 ptr1=&a;
pointer=&ptr1;
(*pointer)->p();
}
```

The output will be `Hello World`.

It can be seen that `JNIEnv *env` is similar to that of `struct_ptr *pointer` and `struct xxx` is similar to **JNINativeInterface**.

Now on considering the second argument, `jobject obj`. `obj` holds the value of the pointer that points to the current object that is **x** here. `GetObjectClass(JNIEnv *, jobject);` is a function pointer inside the structure **JNINativeInterface**. While Java is communicating with C, both `jobject` and `jclass` are the same. The `jclass` is typedef as **jobject** (`typedef jobject jclass`). **GetObjectClass** returns the reference of the current object which is stored in **cls**. **GetFieldID** returns the reference held by **cls** to access the various class members present in the class. Prototype of **GetFieldID** is:

```
jfieldID GetFieldID(JNIEnv *env, jclass clazz,
const char *name, const char *sig);
```

Here **name** parameter holds the name of the class element to be accessed by the native method and **sig** parameter holds signature of the class element. Both these parameters are pointers to character constant. In the program, **name** holds the name of the class element that is **i** and **sig** holds **I**, which is the signature of **i** to indicate that **i** is of integer type. **GetIntField (env, obj, fid);**

## NOTES

returns the value of `i`. Similarly, if the class contains a `float` variable, then the function will be `GetIFloatField (env, obj, fid)`; to access that variable. Consider the following example:

## NOTES

### Program 3.4

```
//p1.java
//Get Your Native Interface Version.
class p1{
public native void fun();
public static void main (String args[])
{
 new p1().fun();
}
static{
 System.loadLibrary("p1");
}
}
p1.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class p1 */
#ifndef _Included_p1
#define _Included_p1
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: p1
 * Method: fun
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_p1_fun
 (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
p1.c
#include "p1.h"
#include "stdio.h"
JNIEXPORT void JNICALL Java_p1_fun
 (JNIEnv *a, jobject b)
{
 jint i = (*a)->GetVersion(a);
 printf ("%x", i);
}
}
```

Both these examples have data type of **i** as **jint**. In **JNI** **int** is typedef as **jint**. Similarly, the typedef version of other Java data types is listed below:

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

**Table 3.5 Java Datatypes and their Native Types with Description**

| <i>Java Type</i> | <i>Native Type</i> | <i>Description</i> |
|------------------|--------------------|--------------------|
| boolean          | jboolean           | unsigned 8 bits    |
| byte             | jbyte              | signed 8 bits      |
| char             | jchar              | unsigned 16 bits   |
| short            | jshort             | signed 16 bits     |
| int              | jint               | signed 32 bits     |
| long             | jlong              | signed 64 bits     |
| float            | jfloat             | 32 bits            |
| double           | jdouble            | 64 bits            |
| void             | void               | N/A                |

## NOTES

### Program 3.5

```
//static native method
//p2.java
class p2{
public native static void fun ();
static{
System.loadLibrary("p2");
}
public static void main (String args [])
{
p2.fun ();
}}
p2.h
/* DONOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class p2 */
#ifndef _Included_p2
#define _Included_p2
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: p2
 * Method: fun
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_p2_fun
(JNIEnv *, jclass);
```

## NOTES

```
#ifndef __cplusplus
}
#endif
#endif
p2.c
#include "stdio.h"
#include "p2.h"
JNIEXPORT void JNICALL Java_p2_fun
(JNIEnv * a, jclass b)
{
printf("Hello C, From JAVA");
}
```

Since the native method is static, the second argument to the function `JNIEXPORT void JNICALL Java_p2_fun (JNIEnv * a, jclass b)` is of `jclass` type. It has already been stated that when Java communicates with C, `jclass` and `jobject` are exactly the same therefore, `jclass` can be replaced with `jobject`.

---

## 3.6 STRING HANDLING

---

A String is a sequence of characters. Java provides full complement features of string handling by implementing strings as built-in objects.

### String Constructors

The String class supports several constructors. To create an empty String, call the default constructor.

For example:

```
String s = new String ();
```

It will create an instance of String with no characters in it.

```
class MakeString
{
 public static void main (String args [])
 {
 char c [] = { 'H', 'E', 'L', 'L', 'O' }
 String s1 = new String (c);
 String s2 = new String (s1);

 System.out.println (s1);
 System.out.println (s2);
 }
}
```

### Output of the program:

```
HELLO
HELLO
```

### String Length

The length of a string is the number of characters that it contains. The syntax is:

```
int length ()
```

For example:

```
The following prints "5", since it has five characters in the strings.
char chars[]={'H','E','L','L','O'}
String s=newString(chars);
System.out.println(s.length);
```

### String Literals

You can use a string literal to initialize a String object.

```
The following code creates two equivalent strings;
char chars[]={'H','E','L','L','O'}
String s1=newString(chars);
String s2="HELLO"; //use string literal
```

### String Concatenation

You can concatenate two strings using `concat ( )`, shown as follows:

```
String concat (String str)
```

This method creates a new object that contains the invoking string with the contents shown as follows:

```
String s1="Hello";
String s2=s1.concat("World");
puts the string "HelloWorld" into s2.
```

### String Compare

The String compare is used to compare the strings.

```
int compareTo (String str)
```

Here *str* is the String compared with the invoking String.

## 3.6.1 Operations on String and Extract Character Methods

Each quoted string is an object of the String class and thus, it is created inside the heap. Therefore, in the following example, `s1` which is a reference of String class, must have a hash code value. It can be checked through the given program:

### Example 3.13

```
class Demo
{
 public static void main (String args[])
 {
 String s1="Java";
 System.out.println(s1.hashCode());
 }
}
```

### Output of the program:

```
71355168
```

Indeed, `s1` is a reference of string class. However, it is interesting to know how an object is created inside the heap without a new operator! When JVM encounters a `string s1="Java"`; such a statement implicitly invokes a new operator to allocate memory from heap. Java has maintained most of the C/C++ syntax and that is why Java provides this facility to declare a string. String objects are created inside the heap in a string pool. In Java, a

## NOTES

## NOTES

string class is immutable, i.e., strings in Java, once created and initialized, cannot be changed on the same reference. If one changes the content of the original string object, then an entirely new string object is created. A `Java.lang.String` class is final. This implies that no class can extend it.

### Most Frequently used Constructors of String Class

The most frequently used constructors are:

#### *No Argument Constructor*

`String s=new String()` this constructor is used to create an empty string.

#### *String(char chars[])*

This constructor can be used to create a string from an array of characters. Consider the following example:

#### Example 3.14

```
class Demo
{
 public static void main (String args [])
 {
 char mak[]={ 'a', 'b', 'c' };
 String s1=new String (mak);
 System.out.println(s1);
 }
}
```

#### Output of the program:

```
abc
String(char[] c, int start, int length)
```

The start parameter specifies the index from which the characters are used to create the **string** object and the length specifies the number of characters to be used from the character indicated by start. The following example shows this:

#### Example 3.15

```
class Demo
{
 public static void main (String args [])
 {
 char mak[]={ 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
 String s1=new String (mak, 3, 3);
 System.out.println(s1);
 }
}
```

#### Output of the program:

```
def
```

#### Example 3.16

```
String (String stringObj);
class Demo
{
 public static void main (String args [])
 {
 String s1=new String ("Hello, Java!");
 System.out.println(s1);
 }
}
```

```
 }
}
```

**Output of the program:**

Hello, Java!

**Example 3.17**

```
class Demo
{
 public static void main (String args [])
 {
 String s2="HelloWorld!";
 String s1=new String (s2);
 System.out.println (s1);
 }
}
```

**Output of the program:**

HelloWorld!

In Java, strings are set in a unicode character sequence. Unicode characters take two bytes from memory. If the programmer is assured that the user characters in a string are only from keyboard, i.e., only ASCII character set, then it is advisable to use byte data type rather than char. The constructors are:

```
String (byteb[])
String (byteb[], int start, int length)
```

The parameter `b` represents the array of bytes. Here also, the start parameter specifies the index from which the characters are used to create the string object and the length specifies the number of characters to be used from the character indicated by start, i.e., the range. Consider the following example:

**Example 3.18**

```
class SubStringConstruction
{
 public static void main (String args [])
 {
 byte ascii[]={65, 66, 67, 68, 69, 70};
 String s1=new String (ascii);
 System.out.println (s1);
 String s2=new String (ascii, 2, 3);
 System.out.println (s2);
 }
}
```

**Output of the program:**

```
ABCDEF
CDE
```

Extended versions of the byte-to-string constructors are also defined. In these a programmer can specify the character encoding that determines how bytes are converted to characters. However, most of the time, default encoding is used.

**NOTES**

## NOTES

### String Comparison Method

#### Behaviour of == Operator in Case of String

The `==` operator is used to determine whether the content of two variables is same or different. However, when `==` is used in the case of `string` references, some abnormal behaviour is shown, as in the program as follows:

#### Example 3.19

```
class Demo
{
 public static void main (String args [])
 {
 String s1="Java";
 String s2="Java";
 if (s1==s2)
 System.out.println("Surprised!!!");
 }
}
```

#### Output of the program:

```
Surprised!!!
```

`s1` is a reference of `string` class. `s1="Java"`; statement instructs the JVM to call the constructor of a `string` class implicitly through a `new` operator. `s1` is created inside stack and memory is allocated to it from the heap. Similarly, `s2` is created inside the stack and JVM implicitly calls the constructor to allocate memory to it from the heap. Since `s1` and `s2` are two different reference variables, it is quite obvious that two separate chunks of memory are allocated and their starting addresses have to be stored in `s1` and `s2` respectively.

Therefore, `s1` cannot be equal to `s2`. This concept is absolutely correct. However, in case of a `string` object, when the constructor is called implicitly, JVM behaves in different way. `Strings` are immutable, i.e., the content of a `string` object cannot be altered in a given reference. So when JVM implicitly calls the `string` class constructor, first JVM searches the heap area to ascertain whether any memory chunk of the `string` object that has the same character set in the same sequence, is available or not. If such a memory chunk is available then, instead of allocating a new memory chunk, it uses the existing one.

In the above program, `s1` and `s2` have the same character set in the same sequence. JVM first allocates memory to `s1` from heap, say 1000. When `s2` is encountered, JVM searches the heap to find whether the same character set in the same sequence is present or not. If it is found, instead of allocating a new memory chunk, JVM assigns to `s2`, the starting address of the memory chunk that has been allocated to `s1`. So `s2` also contains 1000. When a programmer explicitly calls the constructor through `new` operator; a new separate memory chunk is allocated. So `s1` is now not equal to `s2`.

#### Example 3.20

```
class Demo
{
 public static void main (String args [])
 {
 String s1="Java";
```



```
String s2=newString("Java");
if(s1==s2)
System.out.println("hello");
}
}
```

### Output of the program:

no output

### A Closer Look at equals () and hashCode () Method

In case of a string class, the equals () method and the hashCode () has been overridden. The hashCode () method returns the same unique integral value for any two objects that are compared as equal and a different integral value for any two objects that are not compared as equal. Equality is inspected by the equals () method. Except for a string class, the equals method checks the content of reference variables in all cases. If the content is same, then it returns as true.

### Example 3.21

```
class X
{
 int x;
}
class Demo
{
 public static void main (String args[])
 {
 X d1=new X ();
 d1.x=9;
 X d2=new X ();
 d2.x=9;
 if (d1.equals (d2))
 System.out.println ("Hello! We are same.");
 else
 System.out.println ("No, We are different.");
 if (d1==d2)
 System.out.println ("Same same dear.");
 else
 System.out.println ("Believe dear we are different.");
 }
}
```

### Output of the program:

No, We are different.  
Believe dear we are different.

In the (Refer Example 3.21) program, JVM has allocated two different chunks of memory to d1 and d2. Since both d1 and d2 are reference variables, they contain the starting address of the memory chunk allocated to them. Therefore, the contents of d1 and d2 are entirely different. The == operator and equals method checks the content of reference variables and since the content is different, the output is quite obvious. However, the story is different in case of string objects, as can be checked in the following program:

### Example 3.22

```
class Demo
```

## NOTES

## NOTES

```
{
publicstatic voidmain (Stringargts[])
{
Stringd1=newString ("Java");
Stringd2=newString ("Java");
if(d1.equals (d2))
System.out.println ("Hello! Weare same.");
else
System.out.println ("No, We aredifferent.");
if(d1==d2)
System.out.println ("Same samedear.");
else
System.out.println ("Believe dear we aredifferent.");
}
}
```

### Output of the program:

```
Hello! Wearesame.
Believedear we aredifferent.
```

The different output is surprising. However, the program can be explained from the very beginning. `d1` and `d2` are the reference variables of `string` class. The constructor of `string` class has been explicitly invoked through a `new` operator. JVM allocates memory from the heap to `d1` and `d2`. Definitely, `d1` and `d2` have different addresses. Therefore, `d1` is not equal to `d2`. The `==` operator checks the contents of `d1` and `d2`. Since these are different, the output is as usual. However, in the case of `equals` method, as mentioned earlier, these behave in a different way as compared to `string` class. The `equals()` method checks whether the character sequence and character case (upper or lower) is same or not, when invoked by `string` objects. If the character sequence and character case are the same, then the `equals` method returns boolean true, else it returns false. The `hashCode()` method returns the same integral value if the `equals()` method returns boolean true value for two objects. Consider the following example:

### Example 3.23

```
class Demo
{
publicstatic voidmain (Stringargts[])
{
Stringd1=newString ("Java");
Stringd2=newString ("Java");
if(d1.equals (d2))
System.out.println ("Hello! Weare same.");
else
System.out.println ("No, We aredifferent.");
if(d1==d2)
System.out.println ("Same samedear.");
else
System.out.println ("Believe dear we aredifferent.");
System.out.println (d1.hashCode ());
System.out.println (d2.hashCode ());
}
}
```

### Output of the program:

```
Hello! We are same.
Believe dear we are different.
2301506
2301506
```

Since the character set and character sequence are same in both the string objects, equals () method returns the same boolean true value. Therefore, hashCode () method returns the same value for two objects. This happens only because of the fact that strings are immutable.

### Some Important Methods of String Class

Some important methods of string class are:

```
public int length()
```

This method is used to determine the length of the string. This can be explained by the following example:

#### Example 3.24

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 int i=s1.length();
 System.out.println(i);
 }
}
```

### Output of the program:

```
11
public char charAt(int index)
```

This method is used to extract a specified character from the string by the particular index supplied by the programmer. The index supplied must be within the length of the string. The given example clarifies this:

#### Example 3.25

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 char ch=s1.charAt (2);
 System.out.println(ch);
 }
}
```

### Output of the program:

```
l
```

Suppose, the programmer supplies the index which is greater than the length of the string, then string out of bound exception will be generated. This is shown in the following example:

#### Example 3.26

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
```

## NOTES

```
 char ch=s1.charAt(12);
 System.out.println(ch);
 }
}
```

## NOTES

Upon execution, this program throws string out of bound index exception.

```
Public Void GetChars(int Start,int end, char c[], int index_1)
```

This method is used to copy the set of unicode characters from the string from the index supplied through the start variable up to the index represented by the end variable into a character array c. The last variable index\_1 represents the index number of the array from which the characters, copied from the string, have to be stored.

### Example 3.27

```
class Demo{
 public static void main (String args [])
 {
 String s1="Hello world";
 char ch[]=new char[5];
 s1.getChars(1,3,ch,2);
 for (int i=0;i<5;i++)
 System.out.println(ch[i]);
 }
}
```

### Output of the program:

```
e
l
Public byte[] getBytes ()
```

This method is used to convert a unicode string into an array of bytes. Consider the following example:

### Example 3.28

```
class Demo{
 public static void main (String args [])
 {
 String s1="Hello";
 int i=s1.length();
 byte b[]=new byte[i];
 b=s1.getBytes();
 for (int j=0;j<i;j++)
 System.out.println(b[j]);
 }
}
```

### Output of the program:

```
72
101
108
108
111
```

### Public boolean equalsIgnoreCase(Strings)

This method is used to compare two strings ignoring the upper or lower case. This can be seen in the following example.

### Example 3.29

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 String s2="hello world";
 boolean b=s1.equalsIgnoreCase (s2);
 System.out.println (b);
 }
}
```

#### Output of the program:

```
true
Public int compareTo (String sobj)
```

This method is used to compare two strings. This can be seen in the following example.

### Example 3.30

```
class Demo {
 public static void main (String [] args)
 {
 String s1="Java";
 String s2="C++";
 int i=s2.compareTo (s1);
 System.out.println (i);
 }
}
```

#### Output of the program:

```
-7
On checking these two programs:
```

```
class Demo {
 public static void main (String [] args)
 {
 String s1="Java";
 String s2="C++";
 int i=s2.compareTo (s2);
 System.out.println (i);
 }
}
```

#### Output of the program:

```
0
class Demo {
 public static void main (String [] args)
 {
 String s1="Java";
 String s2="C++";
 int i=s1.compareTo (s2);
 System.out.println (i);
 }
}
```

#### Output of the program:

```
7
Therefore, the conclusion is:
i<0 means invoking string is less than the string taken as argument.
```

## NOTES

## NOTES

$i=0$  both the **string** objects are the same.

$i>0$  means the **string** is greater than the **string** taken as argument.

It is interesting to see how this comparison is made. Is it made only through the length? No, absolutely not. In fact it is done through a dictionary order and upper case letter comes first, as can be seen from the following example.

### Example 3.31

```
class Demo {
 public static void main (String [] args)
 {
 String s1="Java";
 String s2="Java";
 int i=s1.compareTo(s2);
 System.out.println(i);
 }
}
```

#### Output of the program:

```
-32
Public Boolean startsWith(String prefixValue, int index)
```

It checks whether the **string** begins with the specified **prefixValue** from the specified index.

### Example 3.32

```
class Demo {
 public static void main (String [] args)
 {
 String s1="Java is cool";
 boolean i=s1.startsWith("cool",8);
 System.out.println(i);
 }
}
```

#### Output of the program:

```
true
Public Boolean startsWith(String prefixValue)
```

It checks if the **string** starts with the specified **prefixValue**. Therefore, a programmer can feel that **startsWith method** of **string** class is overloaded. The following example shows this:

### Example 3.33

```
class Demo {
 public static void main (String [] args)
 {
 String s1="Java is cool";
 boolean i=s1.startsWith("Java");
 System.out.println(i);
 }
}
```

#### Output of the program:

```
true
Public boolean endsWith (String suffixValue)
It checks whether the string ends with the specified suffixValue or not.
```

```
Public int indexOf(int ch)
```

This method is used to return the place value of the specified character in the **string**.

### Example 3.34

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 int b=s1.indexOf('w');
 System.out.println(b);
 }
}
```

### Output of the program:

```
6
Public int indexOf(int ch,int fromIndex)
```

This method returns the place value of the specified character within the **string**. If the **string** does not have the character within the **string**, then 1 is returned. This checking is done with respect to the integer value from **Index**. This can be seen in the following example.

### Example 3.35

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 int b=s1.indexOf('l',4);
 System.out.println(b);
 }
}
```

Output 9

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 int b=s1.indexOf('l',4);
 System.out.println(b);
 }
}
```

Output:2

```
class Demo {
 public static void main (String args [])
 {
 String s1="Hello world";
 int b=s1.indexOf('H',4);
 System.out.println(b);
 }
}
```

Output:

-1

```
Public String concat (String str)
```

This adds the **string** sent in the argument at the end of the **string** through which it is invoked. Consider the following example:

### Example 3.36

```
class Demo {
 public static void main (String args [])
 {
```

## NOTES

```
String s1="Hello";
String s2=s1.concat("World");
System.out.println(s2);
}}
```

## NOTES

### Output of the program:

```
HelloWorld
Public String toLowerCase()
```

This method is used to convert all the characters present in the **string** to lower case and assigns it to a new **string** object.

### Example 3.37

```
class Demo{
 public static void main (String args[])
 {
 String s1="HelloWorld";
 String s2=s1.toLowerCase();
 System.out.println(s2);
 }
}}
```

### Output of the program:

```
helloworld
Public String toUpperCase()
```

This method is used to convert all the characters present in the **string** to upper case and assigns it to a new **string** object.

```
Public String trim()
```

This method is used to eliminate the white space from the beginning of the **string**.

### Example 3.38

```
class Demo{
 public static void main (String args[])
 {
 String s1=" HelloWorld";
 String s2=s1.trim();
 System.out.println(s1);
 System.out.println(s2);
 }
}}
```

### Output of the program:

```
HelloWorld
HelloWorld
Public char[] toCharArray()
```

This method is used to convert a **string** to character array.

### Example 3.39

```
class Demo{
 public static void main (String args[])
 {
 String s1="HELLOWORLD";
 int i=s1.length();
 char ch[]=new char[i];
 ch=s1.toCharArray();
 for (int j=0;j<i;j++)
 System.out.println(ch[j]);
 }
}}
```



### Output of the program:

```
H
E
L
L
O

W
O
R
L
D
```

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

### NOTES

### 3.6.2 StringBuffer

This involves the following:

**StringBuffer ()**

This constructs an empty **StringBuffer** .

**StringBuffer (int capacity)**

This constructs an empty **StringBuffer** with the specified initial capacity.

**StringBuffer (String s)**

This constructs a **StringBuffer** that initially contains the special string.

#### Difference Between String and StringBuffer

1. In Java `String` is a class that represents an immutable string, which represents a sequence of character that can never change. Any modification to the `String` will have to create a new `String` object. But a `StringBuffer` is a mutable `String` object that can be modified at runtime.
2. The significant performance difference between these two classes is that `StringBuffer` is faster than `String` when performing simple concatenations.
3. When `String` object is constructed through `new` keyword, two objects are constructed whereas when `StringBuffer` object is constructed through `new` keyword one object is constructed.
4. `String` never auto-flush the memory whereas `StringBuffer` auto-flush the memory.

#### Program 3.6

```
public class Buffer1
{
 public static void main (String args[])
 {
 StringBuffer sb1=new StringBuffer ("Java") ;
 StringBuffer sb2=new StringBuffer ("Java") ;
```

## NOTES

```
 if(sb1.equals(sb2))
 {
 System.out.println("Equals");
 }else{
 System.out.println("Not Equals");
 }
 }
}
```

### Output of the program:

Not Equals

Here equals () is the method of base class or Object class. StringBuffer class not overrides this method. So output of equals () depends on the return type of hashCode () of Object class. As two objects are constructed, hashCode () returns different hash value and equals () returns false.

### Program 3.7

```
public class Buffer1
{
 public static void main (String args [])
 {
 StringBuffer sb1=new StringBuffer ("Java");
 StringBuffer sb2=new StringBuffer ("Java");
 if (sb1==sb2)
 {
 System.out.println ("Equals");
 }else{
 System.out.println ("Not Equals");
 }
 }
}
```

### Output of the program:

Not Equals

Here equals operator checks the contents but sb1 and sb2 are two reference of StringBuffer class. In Java reference never hold data but they hold base address. As two objects are constructed both the reference hold different address. So here equals operator returns false.

### Methods of StringBuffer

The various methods of **StringBuffer** are:

```
public synchronized int length ()
```

This method returns the length of the **StringBuffer** .

```
public synchronized int capacity ()
```

This method returns the capacity of the **StringBuffer**. A **StringBuffer** has a capacity, which is equal to its longest string. It can represent a string without needing to allocate more memory.

```
public synchronized void setLength(int length)
```

This method is used to set the length of the **StringBuffer**.

```
public synchronized void ensureCapacity(int capacity)
```

This method is used to set the capacity of the **StringBuffer**.

```
public synchronized char charAt(int index)
```

This method returns a character from the **StringBuffer**.

```
public synchronized void getChars(int start, int end, char c[], int
index)
```

This method extracts more than one character from the **StringBuffer**.

```
public synchronized void setCharAt(int index, char ch)
```

This method sets a character in the **StringBuffer**.

```
public synchronized StringBuffer append(Object o)
```

This method calls `toString()` on Object `o` and appends the result to the current **StringBuffer**.

```
public synchronized StringBuffer append(String s)
```

This method appends a string in the **StringBuffer**.

```
public synchronized StringBuffer append(StringBuffer sb)
```

This method appends a **StringBuffer** object to the existing **StringBuffer**.

```
public synchronized StringBuffer append(char c)
```

This method appends a character to the existing **StringBuffer**.

```
public synchronized StringBuffer delete(int index, int length)
```

This method is used to delete more than one character from the **StringBuffer**.

```
public synchronized StringBuffer deleteCharAt(int index)
```

This method is used to delete a character from the **StringBuffer**.

```
public synchronized StringBuffer replace(int index, int length,
String s)
```

This method is used to replace a string in the **StringBuffer**.

```
public synchronized StringBuffer insert(int index, String s)
```

This method is used to insert a string in the **StringBuffer**.

```
public synchronized StringBuffer reverse()
```

This method is used to reverse the **StringBuffer**.

```
public String toString()
```

This method is used to convert a string to a **StringBuffer**.

### Program 3.8

```
public class Text
{
 public static void main(String args[])
 {
 StringBuffer sb1=new StringBuffer("I Java");
 System.out.println(sb1.insert(2,"Like"));
 }
}
```

## NOTES

## NOTES

```
System.out.println(sb1);
String s1=sb1.toString();
System.out.println(s1);
System.out.println(sb1.deleteCharAt(0));
System.out.println(sb1.delete(0,6));
System.out.println(sb1.replace(2,4,"pan"));
System.out.println(sb1.length());
sb1.setLength(7);
System.out.println(sb1.reverse());
}
}
```

### Output of the program:

```
I Like Java
I Like Java
I Like Java
 Like Java
Java
Japan
5
napaJ
```

In **StringBuffer** class all the methods are non-static so they are called through **StringBuffer** object.

### Check Your Progress

7. What are the two categories of stream classes?
8. Differentiate between byte stream classes and character stream classes.
9. Which class is used to read input from the console input device?
10. State about the FileReader.
11. Define the term applet.
12. What are the various states of Java applet?
13. What do you understand by the term volatile modifier?
14. Write the use of modifier strictfp.
15. Write the one vital features of Java Native Interface (JNI).
16. Name the various string classes.

## 3.7 WRAPPER CLASSES

Each Java primitive data type has a corresponding **wrapper class**. When an object of the **wrapper class** is created, it contains a field where primitive data types are stored.

## Significance of Wrapper Classes

The **wrapper classes** are required because of the following reasons:

- Vector, ArrayList, LinkedList, classes present in java.util package cannot handle primitive data types like int, char, float, etc. Hence primitive data types may be converted into object types by using **wrapper classes** present in java.lang package.
- **Wrapper classes** convert primitive data types into objects.

*Table 3.6 Primitive Data Types and Corresponding Wrapper Classes*

| Primitive data types | Wrapper class |
|----------------------|---------------|
| boolean              | Boolean       |
| byte                 | Byte          |
| short                | Short         |
| char                 | Character     |
| int                  | Integer       |
| Float                | Float         |
| long                 | Long          |
| double               | Double        |

## NOTES

### Character Class

**Character** class object is a wrap around a char.

The constructor is:

```
Character (char ch)
```

### Methods

The various methods of the **Character** class are:

```
public static Character valueOf (char c)
```

This method converts a single character into a **Character** class object.

```
public char charValue ()
```

This method is useful to convert a **Character** class object into a primitive char value.

```
public int hashCode ()
```

Returns the hash value of a **Character** class object.

```
public static String toString (char c)
```

This method converts char data types into String.

### Boolean CLASS

**Boolean** is wrapper around a **boolean** value. The constructor of the **Boolean** class is overloaded.

```
Boolean (boolean b)
```

```
Boolean (String s)
```

If **s** contains the String true (in uppercase and in lowercase), then the **Boolean** class object holds true values. Otherwise, the **Boolean** object will hold false values.

## NOTES

To obtain a **Boolean** value contained by the **Boolean** object **boolean** **booleanValue ()** is used.

### Methods

The various methods are as follows:

```
public static boolean parseBoolean (String s)
```

It converts a **String** to **boolean**.

```
public boolean booleanValue ()
```

It extract a **boolean value** from a **Boolean** object.

```
public static String toString (boolean b)
```

This method converts **boolean data types** into **String**.

```
public int hashCode ()
```

It returns the hash value of a **Character** class object.

```
public static Boolean valueOf (String s)
```

It converts a **String** that contains a **boolean** value, into a **Boolean** object.

```
public static Boolean valueOf (boolean b)
```

It converts a **boolean value** into **Boolean** object.

### Number CLASS

**Number** is an abstract class whose sub-classes are **Byte**, **Short**, **Integer**, **Float**, **Long** and **Double**.

The methods of the **Number** class are overridden in the child classes.

Methods of the **Number** class are:

- `byte byteValue ()`
- `short shortValue ()`
- `int intValue ()`
- `float floatValue ()`
- `long longValue ()`
- `double doubleValue ()`

### Byte CLASS

This class wraps a value of the primitive type **byte** in an object. The **Byte** class object contains a **byte** value.

### Constructor

Constructor of a **Byte** class is overloaded, i.e.,

```
Byte (byte b)
```

### Syntax

The following is the syntax of **Byte** class:

```
Byte b1=new Byte ((byte) 12);
Byte (String s)
```

```
Byte b2=new Byte ("45");
Byte b3=new Byte ("Java"); //when we extract the value from
the b3 object, then the program is terminated at runtime by throwing an
Exception "NumberFormatException".
```

### Methods

The various methods are:

```
public static byte parseByte (String s)
```

It converts a String to **byte** data type.

```
public static Byte valueOf (String s)
```

It converts a String to **Byte** class object.

```
public static Byte valueOf (byte b)
```

It converts byte value to **Byte** class object.

```
public int hashCode () :
```

It returns the hash value of **Byte** class object.

```
public static String toString (byte b)
```

It converts **byte** value to String.

### Short CLASS

This class wraps a value of the primitive type **short** in an object. The **Short** class object contains a **short** value.

### Constructor

Constructor of **Short** class is overloaded, i.e.,

```
Short (short s)
```

### Syntax

The following is the syntax of **Short** class:

```
Short s1=new Short ((short) 12);
```

```
Short (String s)
```

```
Short s2=new Short ("45");
```

Short s3=new Short ("Java"); //when we extract the value from the **s3** object then the program is terminated at runtime by throwing an exception, "NumberFormatException".

### Methods

The various methods are:

```
public static short parseShort (String s)
```

It converts a String to **short** data types.

```
public static String toString (short s)
```

It converts **short** data types to String.

```
public static Short valueOf (String s)
```

### NOTES

## NOTES

It converts a `String` to a **Short** class object.

```
public static Short valueOf(short s)
```

It converts **short** data types to `Short` class object.

```
public int hashCode()
```

It returns the hash value of a **Short** class object.

### Integer CLASS

This class wraps a value of the primitive type **int** in an object. The **Integer** class object contains **int** value.

#### Constructor

The constructor of **Integer** class is overloaded, e.g.,

```
Integer(int b)
```

#### Syntax

The following is the syntax of `Integer` class:

```
Integer i1=new Integer(12);
```

```
Integer(String s)
```

```
Integer i2=new Integer("45");
```

```
Integer i3=new Integer("Java"); //when the value from
the i3 object is extracted, the program is terminated at runtime by throwing an
Exception 'NumberFormatException'.
```

#### Methods

The various methods are:

```
public static int parseInt(String s)
```

It converts a `String` to an **int** data type.

```
public static String toString(int s)
```

It converts an **int** data type into a `String`.

```
public static Integer valueOf(String s)
```

It converts a `String` class object into an `Integer` class object.

```
public static Integer valueOf(int s)
```

It converts an **int** data type into an `Integer` class object.

```
public int hashCode()
```

It returns the hash value of an `Integer` class object.

### Long CLASS

This class wraps a value of the primitive type **long** in an object. The **Long** class object contains **long** value.

#### Constructor

The constructor of **Long** class is overloaded, e.g.,

```
Long(long l)
```



## Syntax

The following is the syntax of Long class:

```
Long l1=new Long (12);
Long (String s)
```

```
Long l2=new Long ("45");
```

Long l3=new Long ("Java"); //When the value from the **l3** object is extracted, then the program is terminated at runtime by throwing an Exception, 'NumberFormatException'.

## Methods

The various methods are:

```
public static long parseLong (String s)
```

It converts a String class object into **long** data types.

```
public static String toString (long s)
```

It converts **long** data types into a String.

```
public static Long valueOf (String s)
```

It converts a String class object into a **Long** class object.

```
public static Long valueOf (Long l)
```

It converts **long** data types into **Long** class objects.

```
public int hashCode ()
```

It returns the hash value of a **Long** class object.

## Float CLASS

It wraps a value of the primitive type **float** in an object. The **Float** class object contains a **float** value.

## Constructor

The constructor of **Float** class is overloaded, e.g.,

```
Float (float b)
```

## Syntax

The following is the syntax of **Float** class:

```
Float f1=new Float ((float) 12.5);
Float (String s)
```

```
Float f2=new Float ("45");
```

Float f3=new Float ("Java"); //when one extracts the value from the **f3** object, the program is terminated at runtime by throwing an Exception 'NumberFormatException'.

## Methods

The various methods are:

```
public static float parseFloat (String s)
```

## NOTES

## NOTES

It converts a String class object to **float** data type.

```
public static String toString(float s)
```

It converts **float** data types into String.

```
public static Float valueOf(String s)
```

It converts a **String** class object into a **Float** class object.

```
public static Float valueOf(float s)
```

It converts **float** data types into a **Float** class object.

```
public int hashCode()
```

It returns the hash value of a **Float** class object.

## Double CLASS

This class wraps a value of the primitive type **double** in an object. The **Double** class object contains a **double** value.

### Constructor

The constructor of **Double** class is overloaded, e.g.,

```
Double(double b)
```

### Syntax

The following is the syntax for **Double** class:

```
Double s1=new Double (23.09);
```

```
Double (String s)
```

```
Double d2=new Double ("5.9");
```

Double d3=new Double ("Java"); //when a programmer extracts the value from the **d3** object, the program is terminated at runtime by throwing an Exception 'NumberFormatException'.

### Methods

The various methods are:

```
public static double parseDouble (String s)
```

It converts a String class object to **double** data type.

```
public static String toString(double s)
```

It converts **double** data types into String.

```
public static Double valueOf (String s)
```

It converts a String class object into a **Double** class object.

```
public static Double valueOf(double s)
```

It converts **double** data types into **Double** class objects.

```
public int hashCode()
```

It returns the hash value of a **Double** class object.

## Program 3.9

```
public class Wrap1
{
 public static void main (String args[])
 {
 String s="22";
 int i=Integer.parseInt(s);
 i++;
 System.out.println(i);
 double d=Double.parseDouble(s);
 d+=5;
 System.out.println(d);
 short s1=Short.parseShort(s);
 s1+=10;
 System.out.println(s1);
 s=Integer.toString(i);
 s+=1;
 System.out.println(s);
 s=Double.toString(d);
 s+=12;
 System.out.println(s);
 s=Short.toString(s1);
 s+=2;
 System.out.println(s);
 }
}
```

### Output of the program:

```
23
27.0
32
231
27.012
322
```

### Autoboxing and Unboxing

J2SE 5 supports the **autoboxing** process by which a primitive type is automatically encapsulated into its equivalent type **wrapper class** object. There is no need to explicitly construct a **wrapper class** object. This technique is popularly known as **autoboxing** in Java. Conversely, **unboxing** is required to convert a **wrapper class** object into subsequent primitive data types. In general, **autoboxing** and **unboxing** take place whenever a conversion into an object or from an object is required.

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

## NOTES

## NOTES

### Program 3.1

```
public class Auto
{
 Boolean b1=new Boolean("yes");
 boolean b=b1;
 void show()
 {
 if(b){
 System.out.println("YouNeedMoney");
 }else{
 System.out.println("YouNeedKnowledge");
 }
 }
}

public static void main (String args [])
{
 Auto a=new Auto ();
 a.show ();
}
}
```

#### Output of the program:

You Need Knowledge

#### Programs of wrapper class

### Program 3.11

```
public class Boxing
{
 public static void main (String args [])
 {
 Boolean b1=new Boolean("java");
 boolean b=true;
 b1=b;//autoboxing
 System.out.println(b1);

 Byte bb1=new Byte ((byte) 10);
 byte bb=100;
 bb1=bb;//autoboxing
 System.out.println(bb1);
 }
}
```

#### Output of the program:

true  
100

### Program 3.12

```
public class Unbox
{
 public static void main (String args[])
 {
 Boolean b1=new Boolean("java");
 boolean b=b1;//unboxing
 System.out.println(b);
 Byte bb1=new Byte ((byte) 45);
 byte bb=bb1;//unboxing
 System.out.println(bb);
 }
}
```

#### Output of the program:

```
false
45
```

### 3.7.1 Memory Management

Does Java has a **memory** pointer? This is a very important question. It can be said that Java has a **memory** pointer but it is not as prominent as in the case of C/C++. Reference variables behave like pointers. In computer terminology, reference variables mean variables which can hold the address.

File Name p.java

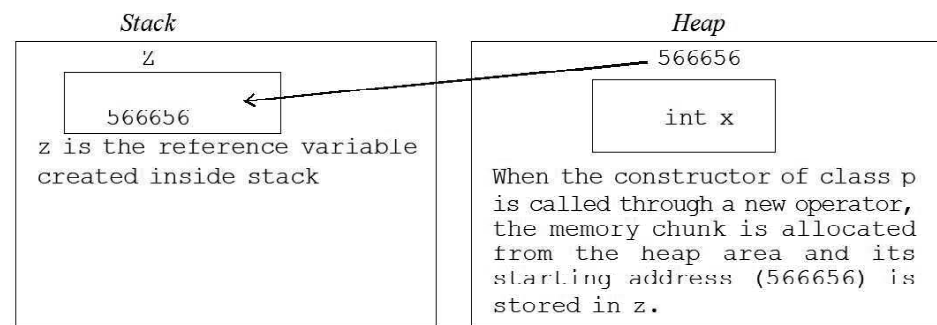
```
class p
{
 int x;
 public static void main (String
args[])
 {
 p z=new p();
 z.x=3;
 System.out.println(z.x);
 }
}
```

This is a simple Java program. Here **p** is a class having an instance variable **x**. To create an object of class **p** one has to call the default constructor of class **p** through a new operator. The job of a new operator is to dynamically allocate **memory** during runtime. **z** is known as a reference variable of class **p**. As explained before, reference variable holds the address of a **memory** location. In Java, **memory** management is completely done by (Java Virtual Memory) JVM. Java code p.class file is created after compiling the above. During the execution of p.class file, first it is loaded in the **memory** by the boot strap class loader,

### NOTES

## NOTES

which is a component of JVM. **z** is a reference variable which is created inside the stack area. When the constructor of class **p** is called through a **new** operator, a chunk of **memory** is allocated from the heap area and its starting address is stored in **z**. Therefore, it can be said that the object of class **p** is created. However, the amount of **memory** that is to be allocated depends upon the sum of the size of instance variables belonging to that class. If the class contains a reference variable as an instance variable, then 4 byte **memory** will be allocated for it. **static** variables and methods are not the parts of an object, so they are stored in a different part of the **memory** known as the method area. Throughout its life **z** will hold the starting address of the **memory** chunk that has been allocated to it from the heap area unless a new **memory** chunk is allocated to it by calling the constructor through a **new** operator. When **z** dies, the **memory** chunk allocated to it would be freed by garbage collector. This is shown in the following Figure:



*Fig. 3.7 The Mapping of Memory between Stack and Heap in Java*

### The Swapping Problem

This problem is meant to test the depth of one's understanding of the concept of reference. The following examples clarify this:

#### Program 3.13

File Name: p.java

```
class p
{
 int x;
 public static void main(String args[])
 {
 p a=new p();
 a.x=1;
 p b=new p();
 b.x=2;
 valueSwap(a,b);
 System.out.println(a.x);
 System.out.println(b.x);
 }
}
```

```

static void valueSwap(p k,p l)
{
int i=k.x;
k.x=l.x;
l.x=i;
}
}

```

**Output of the program:**

2  
1

**Program 3.14**

File Name: q.java

```

class q
{
int x;
public static void main(String args[])
{
q a=new q();
a.x=1;
q b=new q();
b.x=2;
valueSwap(a,b);
System.out.println(a.x);
System.out.println(b.x);
}
static void valueSwap(q k,q l)
{
q temp;
temp=k;
k=l;
l=temp;
}
}

```

**Output of the program:**

1  
2

**NOTES**

## NOTES

### Difference in the Output of `p.java` and `q.java`

Consider `p.java` program. `a` and `b` are the reference variables created inside the stack area. When we call the constructor through a `new` operator, memory is allocated from the heap. Assume that 1000 is the starting address of the memory chunk allocated to `a` and 5000 is starting address of the memory chunk allocated to `b` as shown in the Figure.

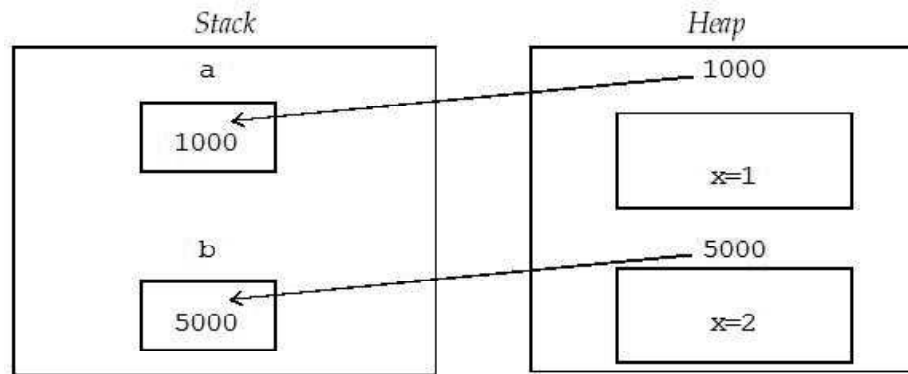


Fig. 3.8 Representation of Two Different Objects in Memory

The `valueSwap()` method takes `a` and `b` as its argument. In the `valueSwap()` method content of `a` is copied to `k` and content of `b` is copied to `l`. Now `k=1000` and `l=5000`. `k` and `l` are the local reference variables of `valueSwap()` method created inside stack.

Inside the method `valueSwap()`, the code `int i=k.x` implies that the value of `i` is 1. Code `k.x=1` means that the content of `x` present in the memory chunk whose starting address is 1000 is changed to 2. Similarly, the code `l.x=i`, which means that the content of `x` present in the memory chunk whose starting address is 5000 is changed to 1. This feature is described below.

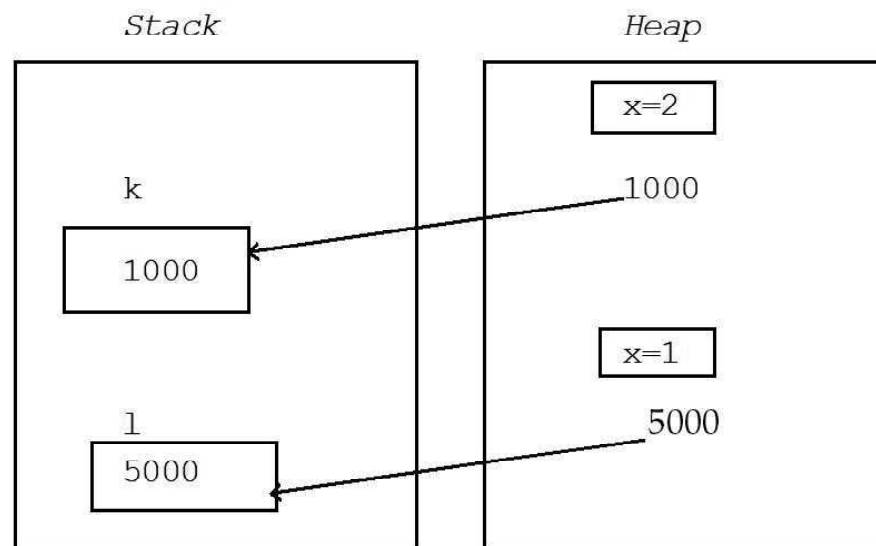


Fig. 3.9 Changing the Reference of Objects



When control goes out of the method `valueSwap()`, although the reference variables `k` and `l` become dead but the changes that they have already made in the heap area are permanent. When control goes back to the `main()` method, the changes reflected there are shown in the following Figure.

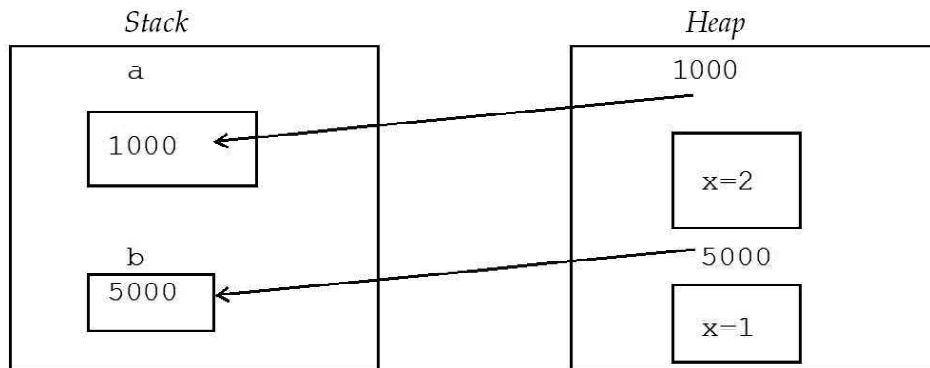


Fig 3.10 Persistence of Heap Allocation

Considering the second program `q.java` up to the method call `valueSwap()`, it is clear that everything is similar to that of `p.java`. Inside the method `valueSwap()` of the program `q.java` things are different. The `temp` is a reference of class `q`. `temp=k;` means now `temp=1000; k=1;` means `k=5000` and `l=temp;` means `l=1000`. When control goes out of the method, `valueSwap()`, `k` and `l` become dead because they are local reference variables created inside stack. Hence, the changes are not reflected.

It is to be understood that local variables always die when control goes out of a method. However, if the local variables are of reference type, then they make permanent changes in the memory locations that they point towards.

### Use of `hashCode()` Method

When an object is created through a `new` operator by calling the constructor of the corresponding class, a unique identifier is assigned to the reference variable, known as `hashCode`. `hashCode` is allotted by JVM. This can be seen in the following example:

#### Program 3.15

```
class p
{
 public static void main(String[]
args)
 {
 p x=new p();

 System.out.println(x.hashCode());
 }
}
```

## NOTES

## NOTES

```
 }
```

The method **hashCode ()** returns the **hashCode** of the corresponding reference when an object is created. **Hash code** is assigned to a reference variable only when memory is allocated from the heap area.

The local variables of primitive data types are created inside the stack area and the memory is also allocated from there. However, all the above complexities are not involved here. The local variables of primitive data types do not have any **hash code** because they do not acquire memory from the heap area. To get a **hash code**, the variable must be of reference type. This is shown by the following example:

### Program 3.16

```
class p
{
 public static void main(String[] args)
 {
 int mak=5;
 System.out.println(mak.hashCode());
 }
}
```

The above program will result in a compilation error because **mak** is not a reference variable and memory is not allocated to it from heap. The error is 'int can not be dereferenced'. The next question that arises is whether it is possible to allocate memory to variables of primitive data type from heap or not. The answer to this is yes. This will be explained in the next chapter.

## 3.7.2 Java.lang Environment Properties

The **java.lang** package has the following classes:

Boolean, Byte, Character, Character.Subset, Character.UnicodeBlock, Class, ClassLoader, Compiler, Double, Enum, Float, InheritableThreadLocal, Integer, Long, Math, Number, Object, Package, Process, ProcessBuilder, Runtime, RuntimePermission, SecurityManager, Short, StackTraceElement, StrictMath, String, StringBuffer, StringBuilder, System, Thread, ThreadGroup, ThreadLocal, Throwable, Void.

The various interfaces present in **java.lang** package are:

Appendable, Comparable, Runnable, CharSequence, Iterable, Cloneable, Readable.

Of these, some of the important and most frequently used classes and their methods are explained below.

## Boolean

This is a wrapper class. It is used to create the object of primitive data type **boolean**.

## Byte

This is another wrapper class. This class is used to create the object of primitive data type **byte**.

## Character

This is another wrapper class. This class is used to create the object of primitive data type **char**.

## Character.Subset

This class is `static` inner class. This class extends the `Object` class. This class is enclosed by the wrapper class `Character`. The signature of this class is, `public static class Character.Subset extends Object`.

## Character.Unicode

This class is also a `static` inner class. Its outer class is `Character`. This class cannot be extended since it is declared with keyword `final`. This class extends the `Character.Subset` class.

## Class

This class extends the `Object` class and implements `Serializable` interface. This class is a `final` class, hence cannot be extended. It does not contain any `public` constructor. In Java, all the arrays, including the arrays of primitive data types, are created by invocation of the `new` operator, just like the creation of an object. Java implements these arrays as the reflection of the object of class `Class`. This class is instantiated only by Java Virtual Machine during the process of class loading by invocation of the `defineClass()` method. This class is generic in nature. One can create the object of this class by the invocation of `getClass()` method which originally belongs to the `Object` class and is overridden in this class, `Class`.

### The `getClass()` and `getName()`

These two methods are invoked together to determine the class name of an object. The following example clarifies this.

## Program 3.17

```
class Demo
{
 public static void main(String[] args)
 {
 Class cl="HelloWorld".getClass();
 System.out.println("The string is an object of
"+cl.getName());
```

## NOTES

```
 cl=System.out.getClass();
 System.out.println("out is an object of "+cl.getName());
 }
}
```

## NOTES

### Output of the program:

```
The string is an object of java.lang.String
out is an object of java.io.PrintStream
```

In the above program, **cl** is a reference of class **Class**. As said earlier, the object of the class **Class** cannot be created directly.

The `getClass()` method returns the name of the class, whose object has invoked it. First the `getClass()` method is invoked by the `String` object. Hence **cl** holds the **String** class. Then the `getName()` method simply shows the name of the class that is held by **cl**. There is another method `getSuperClass()`, which is used to determine the current super class of the class held by the reference variable of class **Class**.

Now the class name to which arrays of primitive data types in Java belongs has to be found. This is shown by the following example.

### Program 3.18

```
class Demo
{
 public static void main (String[] args)
 {
 float mak[]=new float[8];
 Class cl=mak.getClass();
 System.out.println(cl.getName());
 }
}
```

### Output of the program:

```
[F
```

The output is indeed surprising. The `[` indicates that **mak** is an array. If **mak** is a two-dimensional array, then its initial symbol would be `[ [`. The next symbol **F** indicates that **mak** is an array of `float` of single dimension. For other primitive data types the symbols are:

|                    |            |
|--------------------|------------|
| boolean            | Z          |
| byte               | B          |
| char               | C          |
| class or interface | Lclassname |
| double             | D          |
| float              | F          |
| int                | I          |
| long               | J          |
| short              | S          |

## Use of `forName()` Method and the `newInstance()` Method

These can be clarified with the following example:

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

### Program 3.19

```
class X
{
 int i=10;
}
class Demo
{
 int i=10;
 public static void main(String[] args) throws
 InstantiationException, IllegalAccessException,
 ClassNotFoundException
 {
 X b=new X();
 Class cl=Class.forName("demo.X");
 X a=(X)cl.newInstance();
 System.out.println(a.i);
 }
}
```

#### Output of the program:

10

The `forName()` method is a `static` method, therefore it has to be invoked by its class name `Class`. Its signature is:

```
public static Class forName(String className)
throws ClassNotFoundException
```

Hence, it has to be either invoked inside the `try` block or keyword `throws` has to be used. This method returns the class object that holds the class supplied as a `String` argument to this method. It is already known that to create an object of a class, one needs to call the constructor of this class through a `new` operator. However, the `newInstance()` method can be invoked to create the object of the class that is held by `cl`. The signature of `newInstance()` method is:

```
public Object newInstance()
throws InstantiationException,
 IllegalAccessException
```

Therefore, this method has to be invoked inside a `try` block, followed by a `catch` block or by the use of keyword `throws`.

### ClassLoader Class

It is an abstract class which extends the `Object` class. In Java, a class is loaded in the memory either by a boot strap class loader or by user defined class loader. Boot strap class loader is a component of Java Virtual Machine. When a class file

### NOTES

## NOTES

is executed by **javac** command, the boot strap class loader is responsible for loading the class in the memory. The class **ClassLoader** is used to create the user defined class loader. Rarely, a Java programmer requires a user defined class loader.

### Compiler Class

When one compiles a Java source file, a class file is created. By using a **Compiler class**, native executable files can be generated from Java source file. This class cannot be extended. It extends the **Object** class. However, this class is rarely used by a Java programmer.

### Double Class

This is another wrapper class and is used to create the object of primitive data type `double`. This class is explained in the Chapter 17 on Wrapper Class.

### Enum Class

This class is used to create the **Enum** object. **Enum** members are simple constants.

### Float Class

This is another wrapper class and is used to create the object of primitive data type `float`.

### Integer Class

This wrapper class is used to create the object of primitive data type `int`.

### Long Class

This wrapper class is used to create the object of primitive data type `long`.

### Process Class and Runtime Class

These two classes are closely entangled with each other. A Java programmer frequently uses these two classes to develop system level applications.

**Process** is an abstract class. All the methods present in this class are abstract. The signature of the **Process** class is: `public abstract class Process extends Object`. Since it is an abstract class, one cannot directly create the object of **Process** class. For this purpose, normally the `exec()` method of **Runtime** is used. A **Process** object embeds a process into it.

The object of a **Runtime class** contains the current system environment. To instantiate a **Runtime class**, one has to invoke the `getRuntime()` method. It is a `static` method and is invoked by the class name **Runtime**.

### Program 3.20

```
class X
{
 int i;
}
class Demo
{
```

```
public static void main (String[] args)
{
 Runtime rt=Runtime.getRuntime ();
 long Initial, Final;
 Initial=rt.freeMemory ();
 //This method returns the amount of free memory available.
 System.out.println ("Available memory initially: "+Initial);
 Xmak[]=new X[10000];
 Final=rt.freeMemory ();
 System.out.println ("Available memory after the creation of object:
 "+Final);
 long s=(Initial-Final)/10000;
 System.out.println ("Available free memory: "+s);
 rt.gc ();
 //gc () method is used to invoke the garbage collector.
 System.out.println ("Available free memory:"+rt.freeMemory ());
}
}
```

## NOTES

### Output of the program:

```
Available memory initially: 1872280
Available memory after the creation of object: 1832264
Available free memory: 40016
Available free memory: 1875168
```

One question that may arise in a student's mind is the need to create such a large array. This is because by creation of a large array, an appreciable change in the free memory is reflected.

A usual question that newcomers to Java ask is about measuring the size of an object in Java. Having migrated from C/C++, the new students do miss having the size of operator. Java does not provide the size of operator, but one can determine the size of an object through various methods available in **Runtime** class.

### Determining the Size of Primitive Data Type char

This can be clarified with the following example:

#### Programme 3.21

```
class Demo
{
 public static void main (String[] args)
 {
 Runtime rt=Runtime.getRuntime ();
 long Initial, Final;
 Initial=rt.freeMemory ();
 char mak[]=new char[10000];
 Final=rt.freeMemory ();
```

```
long s=(Initial-Final)/10000;
System.out.println("Size of the object is "+s);
}
```

## NOTES

```
}
```

### Output of the program:

Size of the object is 2

## 22.4 Determining the Size of int

This can be clarified with the following example:

### Program 3.22

```
class Demo
{
public static void main (String [] args)
{

 Runtime rt=Runtime.getRuntime ();
 long Initial, Final;
 Initial=rt.freeMemory ();
 int mak[]=new int [10000];
 Final=rt.freeMemory ();
 long s=(Initial-Final)/10000;
 System.out.println("Size of the object is "+s);
 }
}
```

### Output of the program:

Size of the object is 4

## Determining the Size of double

This can be clarified with the following example:

### Program 3.23

```
class Demo
{
public static void main (String [] args)
{

 Runtime rt=Runtime.getRuntime ();
 long Initial, Final;
 Initial=rt.freeMemory ();
 double mak[]=new double [10000];
 Final=rt.freeMemory ();
 long s=(Initial-Final)/10000;
 System.out.println("Size of the object is "+s);
 }
}
```

### Output of the program:

Size of the object is 8



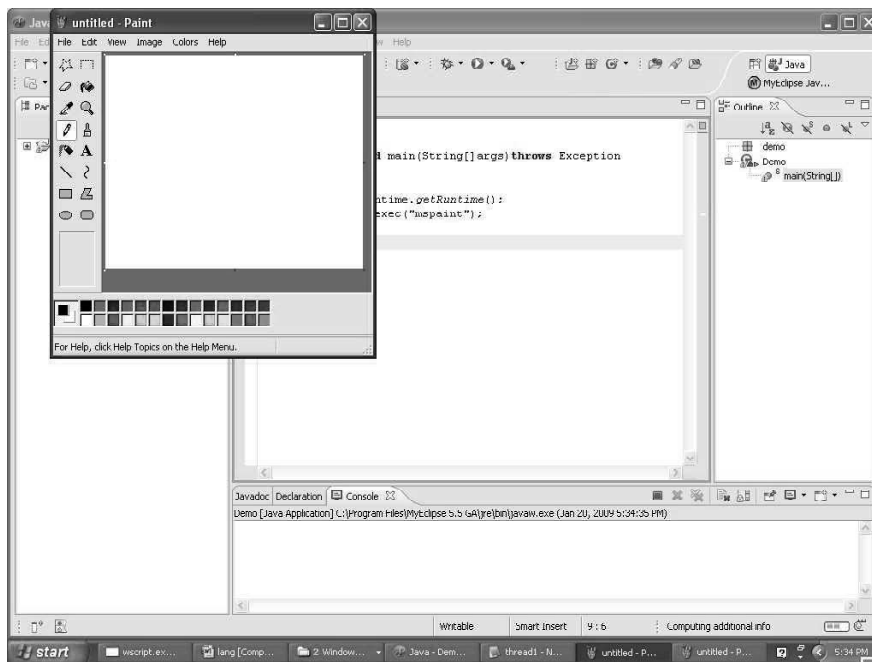
## Use of `exec()` Method to Execute System Dependent Application

In a Windows system the below codes will open MS Paint. This can be seen in the following example:

### Program 3.24

```
class Demo
{
public static void main (String [] args) throws Exception
{
 Runtime rt=Runtime.getRuntime ();
 Process p=rt.exec ("mspaint");
}
}
```

### Output of the program:



First of all, one has to create an object of **Runtime** class. **Runtime** class does not provide any constructor to create an object. Rather `getRuntime()`, which is a static method, is invoked which returns the object of **Runtime**. Then, `exec()` is called through `rt` which returns the object of **Process** class. From the output, it is clear that when `exec()` is successfully executed, MS Paint gets opened on the screen.

### The System Class

This class contains a large number of static methods and variables. One can neither create the object of a **System** class, nor a sub-class of this class.

### The `getProperty()` Method

This method takes the various types of environment properties defined by the **java.lang** package. This method is a static method and hence can only be invoked through a class name.

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

## NOTES

## NOTES

The way to determine the OS in which one is working can be seen in the following example:

### Program 3.25

```
class Demo
{
public static void main (String [] args)
{
 String S=System.getProperty("os.name");
 System.out.println(S);
}
}
```

#### Output of the program:

Windows XP

The `getProperty()` method takes the **os.name** as its argument. This is a pre-defined field. When the `getProperty()` method takes this field as its argument, upon execution it returns the name of the OS where the corresponding class file is executed.

This particular method can be used to create a single class file which will operate differently in different platforms.

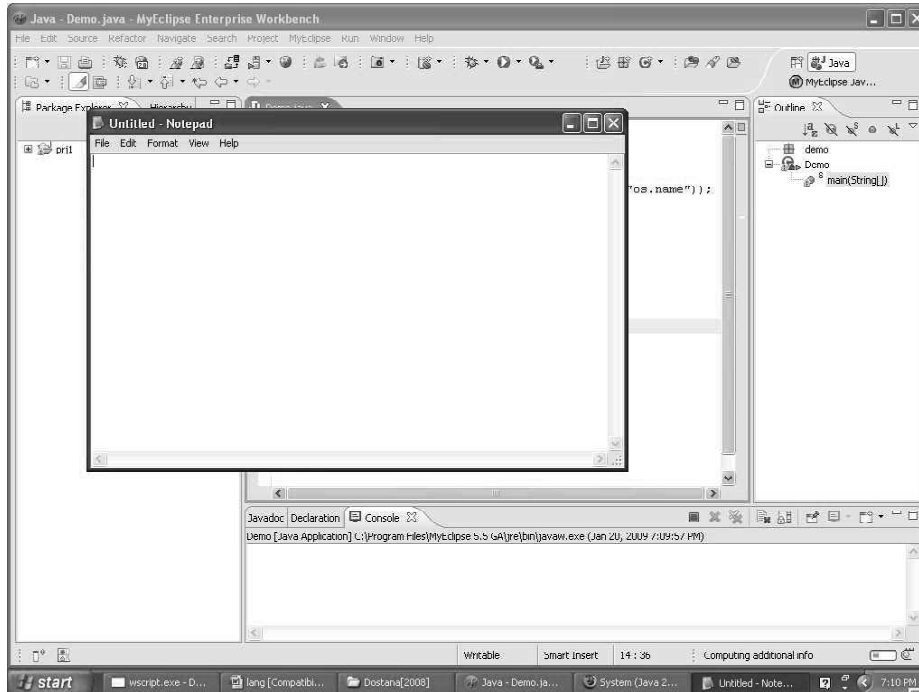
### Program 3.26

```
public class Os
{
 public static void main (String args [])
 {
 String s=new String (System.getProperty ("os.name"));
 Runtime r=Runtime.getRuntime ();
 Process p=null;
 try
 {
 if (s.equals ("Linux"))
 {
 p=r.exec ("gedit");
 }
 if (s.equals ("Windows XP"))
 {
 p=r.exec ("notepad");
 }
 }
 catch (Exception ie)
 {
 }
 }
}
```

On running this program in Windows platform, it will open the Notepad and on running it in Linux platform, it will open the gedit.

In the example, this program has been run in the Windows platform and the output can be seen below:

### Output of the program:



### NOTES

### The arrayCopy () Method

Generally, an array is copied by a loop. This process can be completed by the use of **arrayCopy ()** method in a more efficient way.

### Program 3.27

```
class Demo
{
 public static void main (String args [])
 {
 char mak[]={ 'r','o','o','t',' ','l','e','s','s',' ','a','g','g','r','e','s','s','i','o','n' };
 char arr[]=new char [mak.length];
 System.arrayCopy(mak, 0, arr, 0, mak.length);
 for (int i=0; i<arr.length; i++)
 System.out.print (arr[i]);
 }
}
```

### Output of the program:

root less aggression

This method takes the source array as its first argument. The next argument is the position from which the source has to be copied, third is the name of the destination array, fourth is the position of the destination array from which the

## NOTES

copy process begins and the final is the length of the source array up to which the data has to be copied.

### The ThreadLocal Class

The **ThreadLocal** instances are typically declared as `private` and `static`. Whenever a programmer feels that some data should be unique for the thread and the data should not be shared, then the **ThreadLocal** variables are used. Every individual thread has a separate copy of the variable. This means that any change done to the **ThreadLocal** variable by a thread is totally private to it. It is not going to reflect in any other thread.

Given below is the structure of `ThreadLocal<T>` class defined in `java.lang`:

```
public class ThreadLocal<T>
{
 protected T initialValue();
 public T get();
 public void set(T value);
 public void remove();
}
```

#### **protected T initialValue()**

It returns the **initial value** for the `ThreadLocal` variable of the current thread. Whenever the variable with the `get()` method is accessed for the first time, the `initialValue()` method is called implicitly. If the `set()` method is invoked prior to the `get()` method, then the **initial value** method will not be invoked.

#### **public T get()**

It returns the value of the copy of the `ThreadLocal` variable present in the current thread and creates and initializes the copy in case it is called for the first time by the thread. Then it creates and initializes the copy of the variable.

#### **public void set(T value)**

It sets the copy of the `ThreadLocal` variable of the current thread to the specified value.

#### **public void remove()**

It removes the value of the `ThreadLocal` variable. After removing the value, if one again invokes the `get()` method, then it will again call the `initialValue()` method and will initialize the value of the `ThreadLocal` variable.

### Program 3.28

```
class ThreadLocalDemo1 extends Thread
{
 private static int number = 0;
 private static ThreadLocal threadnumber = new ThreadLocal()
 {
 protected synchronized Object initialValue() {
 return number++;
 }
 };
 public void run()
 {
```

```
System.out.println("Thread " +
Thread.currentThread().getName() +
 "has thread number " + threadnumber.get());
 number++;
System.out.println("Thread " +
Thread.currentThread().getName() +
 "has thread number " + threadnumber.get());
 threadnumber.set(5);
System.out.println("Thread " +
Thread.currentThread().getName() +
 "has thread number " + threadnumber.get());
 threadnumber.remove();
System.out.println("Thread " +
Thread.currentThread().getName() +
 "has thread number " + threadnumber.get());
 threadnumber.remove();
System.out.println("Thread " +
Thread.currentThread().getName() +
 "has thread number " + threadnumber.get());
 }
 public static void main(String[] args) {
 Thread t1 = new ThreadLocalDemol();
 Thread t2 = new ThreadLocalDemol();
 t1.start();
 t2.start();
 }
}
```

## NOTES

### Output of the program:

```
Thread Thread-0 has thread number 0
Thread Thread-0 has thread number 0
Thread Thread-0 has thread number 5
Thread Thread-0 has thread number 2
Thread Thread-0 has thread number 3
Thread Thread-1 has thread number 4
Thread Thread-1 has thread number 4
Thread Thread-1 has thread number 5
Thread Thread-1 has thread number 6
Thread Thread-1 has thread number 7
```

### Cloneable Interface

The **Cloneable** interface, present inside the **java.lang** package, is used to create the clone of an object. The signature of **Cloneable** interface is:

```
public interface Cloneable
{
}
```

The **Cloneable** interface does not contain any method of its own. To clone an object, a class must implement the interface **Cloneable** and then invoke the `clone()` method of **Object** class. The signature of the `clone()` method is:

```
protected native Object clone() throws
CloneNotSupportedException
```

According to the specification, the `clone()` method returns the reference of the **Object** class.

## NOTES

### Program 3.29

```
public class F implements Cloneable
{
 int roll;
 String name=new String();
 F(int i, String c)
 {
 roll=i;
 name=c;
 }
 public static void main(String args[]) throws
 CloneNotSupportedException
 {
 F obj=new F(10,"pinku");
 F ob=(F) obj.clone();
 System.out.println(ob.roll);
 System.out.println(ob.name);
 }
}
```

#### Output of the program:

```
10
pinku
```

The **Cloneable** interface is a **marker interface**. A **marker interface** is an interface, which does not have its own method. However, for certain operations, their implementation is a must. Actually, while implementing the **marker interface**, a programmer implicitly lets the JVM to know that a specific operation is going to be performed.

The **marker interfaces** present inside Java are:

- java.lang.Cloneable
- java.io.Serializable
- java.util.EventListener

To clone one object, the clone () method of the Object class is needed. However, it is required to implement the **Cloneable** interface, i.e., a marker interface to facilitate the class with the ability for cloning its objects.

#### Object Class

In Java, **Object** class is the super class of all the classes. Many of its methods are overridden in different Java inbuilt classes according to their purpose.

#### Methods

The various methods are:

**Object clone ()**

The method is used in creating a clone of an object and has been explained in detail in the Chapter 15 on Strings. This belongs to an **Object** class.

**boolean equals (Object eq)**

This method compares a **String** to the specified **Object**.

**void finalize ()**

This method is called before the garbage collector releases the memory occupied by an object. This has been explained in detail in Chapter 6 on Class Fundamentals.

```
Class getClass ()
```

This has already been explained earlier in this chapter.

```
int hashCode ()
```

This has been explained in detail in object reference and Chapter 15 on Strings.

### Math Class

This class contains the various methods that are useful in scientific and engineering applications. It contains two double constants: **E(the exponential e constant) (~2.72)** and **PI (~3.14)**.

### Methods

The various methods are:

```
public static double sin (double dbl)
```

This method is a static method; hence can be invoked by the class name. It takes a double variable as its argument. This argument is the measurement of an angle in radian. It returns the sine value of the supplied angle.

```
public static double cos (double dbl)
```

This method is a static method; hence can be invoked by the class name. It takes a double variable as its argument. This argument is the measurement of an angle in radian. It returns the cosine value of the supplied angle.

```
public static double tan (double dbl)
```

This method is a static method; hence can be invoked by the class name. It takes a double variable as its argument. This argument is the measurement of an angle in radian. It returns the tangent value of the supplied angle.

This can be seen from the example given below:

### Program 3.30

```
public class Arithml
{
 public static void main(String args[])
 {
 double dbl1= 30;
 double dbl2 = Math.toRadians (dbl1);
 System.out.println(" The angle in radians is : " + dbl1);
 System.out.println(" sine of "+dbl1+" is : "+ Math.
sin(dbl2)
);
 System.out.println(" cosine of "+dbl1+" is : "+ Math.
cos(dbl2));
 System.out.println(" tangent of "+dbl1+" is :"+
Math.tan(dbl2));
 }
}
```

### Output of the program:

```
The angle in radians is : 30.0
sine of "+dbl1+" is : 0.49999999999999994
```

### NOTES

## NOTES

```
cosine of "+dbl1+" is : 0.8660254037844387
tangent of "+dbl1+" is :0.5773502691896257
```

```
public static double asin (double dbl)
```

This method is a `static` method; hence can be invoked by the class name. It takes a `double` variable as its argument. This argument is the sine value of an

angle. It returns the angle in radian. Range is in between  $-\frac{\pi}{2}$  to  $\frac{\pi}{2}$ .

```
public static double acos (double dbl)
```

This method is a `static` method; hence can be invoked by the class name. It takes a `double` variable as its argument. This argument is the cosine value of an angle. It returns the angle in radian. Its range is in between 0 to  $\pi$ .

```
public static double atan (double dbl)
```

This method is a `static` method; hence can be invoked by the class name. It takes a `double` variable as its argument. This argument is the tangent value of an angle. It returns the angle in radian. Its range is in between 0 to  $\pi$ .

### Program 3.31

```
public class Arithm2{
 public static void main (String args [])
 {
 double dbl=0.5;
 System.out.println ("Enter Value is :"+dbl);
 System.out.println ("The angle for which the sine value is 0.5 is
 :"+Math.asin (dbl));
 System.out.println ("The angle for which the cosine value is : " +
 Math. acos (dbl));
 System.out.println ("The angle for which the tangent value is : "
 + Math. atan (dbl));
 }
}
```

### Output of the program:

```
Enter Value is :0.5
The angle for which the sine value is 0.5 is: 0.5235987755982989
The angle for which the cosine value is: 1.0471975511965979
The angle for which the tangent value is: 0.4636476090008061
```

```
public static double toRadians (double dbl)
```

This method is a `static` method; hence can be invoked by the class name. It takes a `double` variable as its argument. This argument is the measurement of an angle in degree. It returns the equivalent angle in radian.

```
public static double toDegrees (double dbl)
```

This method is a `static` method; hence can be invoked by the class name. It takes a `double` variable as its argument. This argument is the measurement of an angle in degree. It returns the equivalent angle in degree.

### Program 3.32

```
class Demo {
 public static void main (String args [])
 {
 System.out.println ("The radian value is :"+Math.
 toRadians (90.0));
 System.out.println ("Degree value
```



```
is:"+Math.toDegrees(1.571));
 System.out.println("Tangent value of two parameters is:
"+Math. atan2(30.00,30.00));
 System.out.println(" The log value is :"+Math.log(35.0));
 System.out.println("The exponent value is
:"+Math.exp(30.00));
 }
}
```

### Output of the program:

```
The radian value is :1.5707963267948966
Degree value is: 90.01166961505233
Tangent value of two parameters is: 0.7853981633974483
The log value is: 3.5553480614894135
The exponent value is: 1.0686474581524463E13
```

**public static double exp (double dbl)**

This method is a `static` method; hence can only be invoked by a class name. It takes a `double` variable as its argument. This argument is the tangent value of an angle. It returns the angle in radian. Range is in between 0 to  $\pi$ .

**public static double log (double dbl)**

The `log` method has only one `double` value, i.e., **dbl** as the parameter. This method returns the natural logarithm (base `e`) of a `double` value.

**public static double sqrt (double dbl)**

Here, the `sqrt` method has only one `double` value **dbl** as the parameter and it returns the square root of the given value **dbl**. When the argument is **NaN** or less than zero, the result is **NaN**.

**public static double IEEEremainder (double db11, double db12)**

This method is used to calculate the remainder operation on two arguments and has two parameters, i.e., **db11**(the dividend) and **db12**(the divisor). Here, it returns the remainder when **db11** is divided by **db12**.

**public static double ceil (double dbl)**

This method has only one `double` value, i.e., **dbl** which is taken as parameter. According to this method, the given `double` value returns the smallest `double` value which is not less than the argument and is equal to a mathematical integer.

**public static double floor (double dbl)**

This `floor` method has only one `double` value **dbl** as the parameter and returns the largest `double` value which is not greater than the argument and is equal to a mathematical integer.

**public static double rint (double dbl)**

This `rint` method contains only one `double` value **dbl** as the parameter. It returns the closest `double` value to that **dbl** and is equal to a mathematical integer. If there are two `double` values that are equally close to the value of the argument, it returns the integer value that is even.

**public static double atan2 (double db11, double db12)**

Here there are two `double` values **db11** and **db12**. These are the parameters for this method. The given rectangular coordinates **db12** and **db11** are converted to polar (`r`, `theta`) by this method. It also computes the phase `theta` by computing an arc tangent of **db11/db12** in the range of  $-\pi$  to  $\pi$ .

## NOTES

## NOTES

```
public static double pow(double db11, double db12)
```

Here, two double values **db11** and **db12** are taken as the parameters for this method. This method returns the value of the first argument which is raised to the power of the second argument. If **db11**==0.0, then **db12** must be greater than 0.0, otherwise it will throw an exception. An exception can also arise if (**db11**<=0.0), **db12** is not equal to a whole number.

### Program 3.33

```
class Arithm4 {
 public static void main (String args[]) {
 System.out.println("The square root value is :"+Math.sqrt(25));
 System.out.println("\n The remainder of 5 divided by 2
is:"+Math.IEEEremainder(5,2));
 System.out.println("\n The ceil value is : "+Math.ceil(5.6));
 System.out.println("\n The floor value is:"+Math.floor(5.6));
 System.out.println("\n The power value is :"+Math.pow(5.0,2.0));
 System.out.println("\n "+Math rint value is :"+Math.rint(30.6));
 }
}
```

Most of the methods of `Math` class are `static` in nature. Their names clearly indicate their tasks. One can easily search them on Google.

Thus the **java.lang** package is the only package that is available to the programmer by default. It provides a lot of utility tools to the programmer. Recently added interfaces like instrumentation have increased the scope of this package by adding new dimension for the development of user-defined class loader through Java agents and `premain` method. This package is the premier package of Java programming language forever.

### 3.7.3 Security Manager and SecurityManager Class

A security manager is referred as an object that specifically defines a security policy for an application. This policy typically specifies those actions that are unsafe or sensitive. Any of the action that is not allowed by the security policy can cause a **SecurityException** to be thrown. An application can also query its security manager to discover which actions are allowed.

Typically, in Java, a web applet runs with a security manager provided by the browser or Java Web Start plugin. Other types of applications generally run without a security manager, unless the application itself defines one. If no security manager is present, the application has no security policy and acts without restrictions.

#### Interacting with the Security Manager

The security manager is an object of type **SecurityManager**; to obtain a reference to this object, invoke **System.getSecurityManager**.  
**SecurityManager appsm = System.getSecurityManager  
( );**

If there is no security manager, this method returns **null**.

Once an application has a reference to the security manager object, it can request permission to do specific things. Many classes in the standard

libraries do this. For example, **System.exit**, which terminates the Java Virtual Machine (JVM) with an exit status, invokes **SecurityManager.checkExit** to ensure that the current thread has permission to shut down the application.

The **SecurityManager** class defines many other methods used to verify other kinds of operations, for example, **SecurityManager.checkAccess** verifies thread accesses, and **SecurityManager.checkPropertyAccess** verifies access to the specified property. Each operation or group of operations has its own **checkXXX()** method.

Additionally, the set of **checkXXX()** methods represents the set of operations that are already subject to the protection of the security manager. Typically, an application does not have to directly invoke any **checkXXX()** methods.

### Recognizing a Security Violation

Many actions that are repetitive or routine without a security manager can throw a **SecurityException** whenever run with a security manager. This is true even when invoking a method that is not documented as throwing **SecurityException**. For example, consider the following code used to write to a file:

```
reader = new FileWriter("vikas.txt");
```

In the absence of a security manager, this statement executes without error, provided **vikas.txt** exists and is writable. But suppose this statement is inserted in a 'Web Applet', which typically runs under a security manager that does not allow file output. The following error messages might result:

```
appletviewer fileApplet.html
Exception in thread "AWT-EventQueue-1"
java.security.AccessControlException: access denied
(java.io.FilePermission xanadu.txt write)
at
java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
at
java.security.AccessController.checkPermission(AccessController.java:546)
at
java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
at
java.lang.SecurityManager.checkWrite(SecurityManager.java:962)
at
java.io.FileOutputStream.<init>(FileOutputStream.java:169)
at java.io.FileOutputStream.<init>(FileOutputStream.java:70)
at java.io.FileWriter.<init>(FileWriter.java:46)
...
```

Remember that the specific exception **java.security.AccessControlException** thrown in this case is a subclass of **SecurityException**.

Therefore, the security manager is a class that allows applications to implement a security policy. It allows an application to determine, before performing a possibly unsafe or sensitive operation, what the operation is and whether it is

## NOTES

## NOTES

being attempted in a security context that allows the operation to be performed. The application can allow or disallow the operation.

```
public class SecurityManager
```

```
extends Object
```

The **SecurityManager** class contains many methods with names that begin with the word **check**. These methods are called by various different methods in the Java libraries before those methods perform some certain hypothetically sensitive operations. The invocation of such a **check** method typically looks as follows:

```
SecurityManager security =
System.getSecurityManager();
if (security != null) {
 security.checkXXX(argument, . . .);
}
```

The security manager is thus given an opportunity to check completion of the operation by throwing an exception. A security manager routine simply returns if the operation is permitted, but throws a **SecurityException** if the operation is not permitted. The only exception to this convention is **checkTopLevelWindow**, which returns a **Boolean** value.

The current security manager is set by the **setSecurityManager** method in class **System**. The current security manager is obtained by the **getSecurityManager** method.

The special method **checkPermission(java.security.Permission)** determines whether an access request indicated by a specified permission should be granted or denied. The default implementation calls

```
AccessController.checkPermission(perm);
```

If a requested access is allowed, **checkPermission** returns. If denied, a **SecurityException** is thrown.

---

## 3.8 JAVA UTILITY CLASS

---

- Java collections framework provides a well designed set of interfaces and classes that support operations on a collection of objects.
- The **Locale** class is used to tailor program output to the conventions of a particular geographic, political or cultural region.
- The **GregorianCalendar** provides support for traditional Western calendars.

### **StringTokenizer CLASS**

Parsing is a familiar term in compiler design. However, what exactly is meant by parsing? When one writes a program in any high-level language, say Java, it is the task of the compiler to convert the source to some intermediate language like byte code or a machine readable language. During this process, the compiler first checks the syntactical correctness of the program. This checking is done by dividing the text entered by the programmer into a number of sub-strings, according to some

parsing protocol or rule and this process is called parsing. Java provides the **StringTokenizer** class to divide the entered text into a number of sub-strings or tokens. **StringTokenizer** implements the Enumeration interface for which one can traverse through the various sub-strings present in the entered text.

### Program 3.34

```
import java.util.*;
class Demo {
 static String str1="Past, Present and future is nothing special,
 they are just Clock Time!!";
 public static void main (String args[])
 {
 StringTokenizer sTokenizer=new StringTokenizer (str1,",");
 while (sTokenizer.hasMoreTokens ())
 {
 String keyused= sTokenizer.nextToken ();
 System.out.println (keyused);
 }
 }
}
```

#### Output of the program:

```
Past
Present and future is nothing special
they are just Clock Time!!
```

In the above example, **sTokenizer** is an object of the **StringTokenizer** class. The constructor of the **StringTokenizer** class takes two arguments. The first argument is the string, which is meant for parsing. The second argument is the string (here it is the comma, “,”), according to which the first argument has to be divided into a number of sub-strings.

#### Methods

The various methods are:

**int countTokens ()**

This method determines the number of tokens left to be parsed and returns the result.

**boolean hasMoreElements ()**

This method returns **true** if one or more tokens remain in the string and returns **false** if there is none.

**boolean hasMoreTokens ()**

This method returns **true** if one or more tokens remain in the string and returns **false** if there are none.

**Object nextElement ()**

This method returns the next token as an object.

**String nextToken (String delimiters)**

This method returns the next token as a **String** and sets the **delimiters** of the **String**.

## NOTES

## NOTES

The StringTokenizer class provides another two constructors. These are:

**StringTokenizer (String string)**

**StringTokenizer (String string, String delimiters, boolean delim)**

Another program is given below:

### Program 3.35

```
import java.util.*;
class Demo {
 static String str1="Past, Present and future is nothing special,
 they are just Clock Time!!";
 public static void main (String args[])
 {
 StringTokenizer sTokenizer=new StringTokenizer (str1);
 String arr[]={",","&","is",",",","};
 int i=0;
 while (i<4)
 {
 String keyused = sTokenizer.nextToken (arr[i]);
 System.out.println (keyused);
 i++;
 }
 }
}
```

### Output of the program:

```
Past
Present and future is nothing special
```

The output here is quite simple and straight forward.

### BitSet Class

Java provides the **BitSet** class to store the bit values. The array created by the **BitSet** class is dynamic in nature, i.e., it can grow and shrink according to the given input. The constructors available in **BitSet** class are:

- **BitSet ( )**
- **BitSet (int capacity)** This constructor is used to initialize the capacity of the **BitSet** class object.

The next example illustrate, the use of the **BitSet** class.

### Program 3.36

```
import java.util.*;
class Demo {
 static String str1="Past, Present and future is nothing special,
 they are just Clock Time!!";
 public static void main (String args[])
 {
 BitSet bSet1=new BitSet (32);
 BitSet bSet2=new BitSet (32);
 for (int i=1; i<=32; i++)
```

```
{
 if(i%2==0)
 bSet1.set(i);
 if(i%3==0)
 bSet2.set(i);
}
System.out.println(bSet1);
System.out.println(bSet2);
//Anding of Bits
bSet2.and(bSet1);
System.out.println("After ANDING:"+bSet2);
// The OR operation
bSet1.or(bSet2);
System.out.println("After OR:"+bSet1);
//XOR operation
bSet2.xor(bSet1);
System.out.println("After XOR:"+bSet2);
}
```

## NOTES

### Output of the program:

```
{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32}
{3, 6, 9, 12, 15, 18, 21, 24, 27, 30}
After ANDING: {6, 12, 18, 24, 30}
After OR: {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32}
After XOR: {2, 4, 8, 10, 14, 16, 20, 22, 26, 28, 32}
```

The `set()` method sets the particular bit on for the **BitSet** object through which it is invoked. For example, when this statement `if (i % 2 == 0) bSet1.set(i);` is encountered, second bit, fourth bit, sixth bit, eighth bit and so on of **bSet1** is turned on. The **and()** method performs the **and** operation and the result is stored in the invoking object **bSet2**.

### Date

Java provides `Date` class to show the current date and time. By using this class one can get system defined date and time. `Date` class also implements the `Comparable` interface.

The constructors available in `Date` class are:

#### **Date ()**

It initializes the object with the current date and time.

#### **Date (long millisec)**

It accepts one argument that equals the number of milliseconds.

### Program 3.37

```
import java.util.Date;
public class date7
{
```

## NOTES

```
public static void main (String args [])
{
 Date date = new Date ();
 System.out.println ("Date is : " + date);
 System.out.println ("Milliseconds since January 1, 1970,
00:00:00 GMT : " + date.getTime ());
}
}
```

### Output of the program:

```
Date is : Tue Jan 01 03:09:10 IST 2002
Milliseconds since January 1, 1970, 00:00:00 GMT :1009834750484.
```

### Methods

The various methods are:

**void setTime (long time)**

It sets the time and date, which is specified by `time`.

**long getTime ()**

It returns the number of milliseconds, that have elapsed since January 1970.

**String toString ()**

It converts the invoking date object into a `String`.

**boolean equals (Object date)**

It returns `true` value, if invoking the `Date` object contains the same time and date as one specified by `date`, otherwise it returns `false`.

### Calendar class

It is an abstract class which is present in `java.util` package. There is no constructor provided by `Calendar` class.

### Methods

The various methods are:

**static Calendar getInstance ()**

It returns the `Calendar` object for the default time zone.

To set month, year, date, hour, minute, second we have to call a method, i.e., **set ()** of `Calendar` class.

**set ()** method having corresponding `final` variables like

**final void set (int year, int month, int dayOfMonth)**

**final void set (int year, int month, int dayOfMonth, int  
hours, int minutes, seconds)**

To get month, year, date, hour, minute, second one has to call a method, i.e., **get ()** of `Calendar` class.

Inside this **get ()** method called corresponding `static` variables like

`Calendar.MONTH`

`Calendar.YEAR`

`Calendar.DATE`

`Calendar.HOUR`

`Calendar.MINUTE`

`Calendar.SECOND`



### Program 3.38

```
import java.util.Calendar;
public class DMY{
 public static void main (String[] args)
 {
 Calendar calendar = Calendar.getInstance ();
 int year = 2009;
 int month = Calendar.MARCH;
 int date = 1;
 calendar.set (year, month, date);
 int days =
calendar.getActualMaximum (Calendar.DAY_OF_MONTH);
 System.out.println ("Number of Days: " + days);
 year = calendar.get (Calendar.YEAR);
 month = calendar.get (Calendar.MONTH);
 System.out.println ("Current month: " + month);
 System.out.println ("Current Year: " + year);
 }
}
```

#### Output of the program:

Number of Days-31

Current month-2

Current Year-2009

#### GregorianCalendar Class

It is a class present in `java.util` package and it is a child class of `Calendar` class. All the methods of the `Calendar` class are also present in this class along with some additional methods such as `isLeapYear ()`. This method shows whether the current year is leap year or not.

There are also several constructors for `GregorianCalendar` objects. These are:

```
GregorianCalendar ()
```

It initialize the object with the current date and time in the default locale and time zone.

```
GregorianCalendar (int year, int month, int dayOfMonth)
```

It sets the day, month and year.

```
GregorianCalendar (int year, int month, int dayOfMonth, int
hours, int minutes)
```

It sets the day, month, year, hours and minutes.

```
GregorianCalendar (int year, int month, int dayOfMonth, int
hours, int minutes, int seconds)
```

It sets the day, month, year, hours, minutes and seconds.

All the above three constructors set the day, month, and year. Here, `year` specifies the number of years that have elapsed since 1900. The month is specified by `month`, with zero indicating January. The day of the month is specified by `dayOfMonth`.

#### NOTES

## NOTES

### Program 3.39

```
import java.util.*;
class Demo1 {
public static void main (String args[]) {
String mth[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
int year=0;
GregorianCalendar cal = new GregorianCalendar();
System.out.print ("Date is:" + cal.get (Calendar.DATE));
System.out.print (" , " + mth[cal.get (Calendar.MONTH)] + " , ");
System.out.println (cal.get (Calendar.YEAR));
System.out.print ("Time is:" + cal.get (Calendar.HOUR) + ":");
System.out.print (cal.get (Calendar.MINUTE) + ":");
System.out.println (cal.get (Calendar.SECOND));
cal.set (Calendar.HOUR, 12);
cal.set (Calendar.MINUTE, 25);
cal.set (Calendar.SECOND, 52);
System.out.print ("Updated time is: ");
System.out.print (cal.get (Calendar.HOUR) + ":");
System.out.print (cal.get (Calendar.MINUTE) + ":");
System.out.println (cal.get (Calendar.SECOND));
if (cal.isLeapYear (year))
{
System.out.println ("The current year is a leap year");
}
else {
System.out.println ("The current year is not a leap year");
}
}
}
```

#### Output of the program:

```
Date is: 1, Jan, 2002
Time is: 4:41:22
Updated time is: 0:10:52
The current year is a leap year
```

#### Random Class

This class is present in `java.util` package. It is used for giving random numbers like `Random int`, `Random double`, `Random float`, etc.

**Random class** provides the following constructors:

**Random ()**

It creates a new random number generator seed.

**Random (long l)**

It creates a new random number generator using a single long seed.

## Methods

**boolean nextInt()**

It returns the next integer random number.

**boolean nextFloat()**

It returns the next float random number.

**boolean nextDouble()**

It returns the next double random number.

**boolean nextInt(int n)**

It returns the next integer random number within the range, i.e., zero to n.

## Program 3.40

```
import java.util.Random;
public final class ran1
{
 public static final void main (String args [])
 {
 show ("Generating 10 random integers in range 0 to 50.");
 Random r = new Random ();
 for (int i=1; i<=10; ++i)
 {
 int randomInt=r.nextInt (50);
 show ("Generated:" + randomInt);
 }
 show ("Done.");
 }
 private static void show (String str) {
 System.out.println (str);
 }
}
```

### Output of the program:

```
Generating 10 random integers in range 0 to 50.
Generated:4
Generated:2
Generated:3
Generated:33
Generated:12
Generated:28
Generated:43
Generated:43
Generated:20
Generated:33
Done.
```

## Locale Class

This class is used to describe a geographical, political or cultural region. In this class the date, time, numbers format is used to display it according to the customs of the user's native country, region or culture.

## NOTES

## NOTES

### Constructors

**Locale (String language)**

It returns the specific language.

**Locale (String language, String country)**

It returns the specific language and country codes.

**Locale (String language, String country, String data)**

It returns the specific language, country codes and data according to customer specified requirement.

### Methods

**final String getDisplayCountry ()**

It is used to display country name.

**final String getDisplayLanguage ()**

It is used to display language name.

**final String getDisplayName ()**

It is used to describe the **Locale** completely. **Locale** means some constants that are defined by **Locale** class. These are:

CANADA, CHINA, CHINESE, ENGLISH, FRANCE, FRENCH, GERMAN, GERMANY,  
ITALIAN, ITALY, JAPAN, JAPANESE, KOREA, KOREAN, PRC, TAIWAN, UK, US.

**static void setDefault (Locale obj)**

It sets the default **Locale** according to the given object of **Locale** class.

**static Locale setDefault (Locale obj)**

It gets the default **Locale**.

### Program 3.41

```
import java.util.Locale;
public class loc {
 public static void main (String[] args) {
 System.out.println(Locale.getDefault());
 Locale l = new Locale ("de", "DE");
 Locale l1 = new Locale ("fr", "FR");
 System.out.println ("Default language name (default) : "+
 l.getDisplayLanguage());
 System.out.println ("German language name (German) : "+
 l.getDisplayLanguage (l));
 System.out.println ("German language name (French) : "+
 l.getDisplayLanguage (l1));
 }
}
```

### Output of the program:

```
en_US
Default language name (default) : German
Default language name (German) : Deutsch
Default language name (French) : allemand
```

## The Observable class and the Observer Interface

These two are closely entangled with each other. One will seldom find a programmer using only one of the classes to develop an application in Java. Java provides a unique tool through this class and interface to keep track of the changes in the various objects during the course of execution of the program.

The class which keeps a track of the changes has to implement **Observer interface** and the class which is under focus must extend the **Observable class**. The following program makes it clear.

### Program 3.42

```
import java.util.*;
class X implements Observer{
 public void update (Observable obs, Object obj)
 {
 System.out.print ("Object of Y is changed:");
 }
}
class Y extends Observable {
 int i;
 void fun ()
 {
 i++;
 this.setChanged();
 notifyObservers (this);
 try{
 Thread.sleep (100);
 }catch (InterruptedException e) {}
 }
 show ();
}
void show ()
{
 System.out.println (i);
}
}
class Demo {
 public static void main (String args [])
 {
 Y a=new Y ();
 X b=new X ();
 a.addObserver (b);
 for (int i=0; i<10; i++)
 {
 a.fun ();
 }
 }
}
```

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

## NOTES

## NOTES

```
 }
}
Output of the program:
Object of Y is changed:1
Object of Y is changed:2
Object of Y is changed:3
Object of Y is changed:4
Object of Y is changed:5
Object of Y is changed:6
Object of Y is changed:7
Object of Y is changed:8
Object of Y is changed:9
Object of Y is changed:10
```

Class **X** implements an **Observer interface**. This tells to JVM that the object of **X** will behave as an Observer. Class **Y** extends the **Observable class**; that means changes made to the object of **Y** are going to be monitored. Class **Y** has an instance variable **i**. Inside the `main` method, the `fun` method is invoked by the object of **Y**. The `fun` method manipulates the object **a** by performing `i++`. The statement `a.addObserver(b)`; registers the object **b** as an observer to observe the changes made in object **a**. Now, inside the `fun` method, the `setChanged()` method has been invoked. The `setChanged()` method contains a flag bit which turns on when it is invoked. If one does not invoke this method, the flag bit remains turned off. The `setChanged()` method has to be invoked by the object which is to be observed. The `notifyObservers(this)` method informs the Observer when the object is manipulated by passing that object to the `update` method. This method actually implicitly invokes the `update()` method if the flag bit is turned on by the `setChanged()` method and sends the object **a** to `update()` method implicitly. Once the control is transferred, the change is monitored and the message inside the `System.out.println()` is displayed. This is shown below.

### Program 3.43

```
import java.util.*;
class X implements Observer{
 public void update (Observable obs, Object obj)
 {
 System.out.println ("Object of Y is changed:"+(Integer) obj);
 }
}
class Y extends Observable {
 Integer i;
 void fun ()
 {
 i++;
 this.setChanged();
 }
}
```

```
 notifyObservers(this.i);
 try{
 Thread.sleep(100);
 }catch (InterruptedException){}
 }
}
class Demo {
 public static void main(String args[])
 {
 Y a=new Y();
 a.i=new Integer("0");
 X b=new X();
 a.addObserver(b);
 for(int i=0;i<10;i++)
 {
 a.fun();
 }
 }
}
```

### **Output of the program:**

```
Object of Y is changed:1
Object of Y is changed:2
Object of Y is changed:3
Object of Y is changed:4
Object of Y is changed:5
Object of Y is changed:6
Object of Y is changed:7
Object of Y is changed:8
Object of Y is changed:9
Object of Y is changed:10
```

The next example shows an object having multiple observers:

### **Program 3.44**

```
import java.util.*;
class X implements Observer{
 public void update (Observable obs, Object obj)
 {
 System.out.println ("Object of Y is changed:"+(Integer) obj);
 }
}
class Y extends Observable {
 Integer i;
 void fun ()
 {
 i++;
 this.setChanged();
 }
}
```

## **NOTES**

## NOTES

```
 notifyObservers(this.i);
 }
 try{
 Thread.sleep(100);
 }catch (InterruptedException) {}
}
}
class Demo {
 public static void main(String args[])
 {
 Y a=new Y();
 a.i=new Integer("0");
 Y c=new Y();
 c.i=new Integer("100");
 X b=new X();
 a.addObserver(b);
 c.addObserver(b);
 for(int i=0;i<10;i++)
 {
 a.fun();
 c.fun();
 }
 }
}
```

### Output of the program:

```
Object of Y is changed:1
Object of Y is changed:101
Object of Y is changed:2
Object of Y is changed:102
Object of Y is changed:3
Object of Y is changed:103
Object of Y is changed:4
Object of Y is changed:104
Object of Y is changed:5
Object of Y is changed:105
Object of Y is changed:6
Object of Y is changed:106
Object of Y is changed:7
Object of Y is changed:107
Object of Y is changed:8
Object of Y is changed:108
Object of Y is changed:9
Object of Y is changed:109
Object of Y is changed:10
Object of Y is changed:110
```



The next example shows two objects monitored by two different observers:

### Program 3.45

```
import java.util.*;
class X implements Observer{
 public void update (Observable obs, Object obj)
 {
 System.out.println("Object of Y is changed:"+(Integer) obj);
 }
}
class Z implements Observer{
 public void update (Observable obs, Object obj)
 {
 System.out.println("Object of Y is changed:"+(Integer) obj);
 }
}
class Y extends Observable {
 Integer i;
 void fun ()
 {
 i++;
 this.setChanged();
 notifyObservers(this.i);
 try{
 Thread.sleep(100);
 }catch (InterruptedException){}
 }
}
class Demo {
 public static void main (String args[])
 {
 Y a=new Y();
 a.i=new Integer ("0");
 Y c=new Y();
 c.i=new Integer ("100");
 X b=new X();
 Z d=new Z();
 a.addObserver (b);
 c.addObserver (d);
 for (int i=0;i<10;i++)
 {
 a.fun();
 c.fun();
 }
 }
}
```

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

### NOTES

## NOTES

### Output of the program:

```
Object of Y is changed:1
Object of Y is changed:101
Object of Y is changed:2
Object of Y is changed:102
Object of Y is changed:3
Object of Y is changed:103
Object of Y is changed:4
Object of Y is changed:104
Object of Y is changed:5
Object of Y is changed:105
Object of Y is changed:6
Object of Y is changed:106
Object of Y is changed:7
Object of Y is changed:107
Object of Y is changed:8
Object of Y is changed:108
Object of Y is changed:9
Object of Y is changed:109
Object of Y is changed:10
Object of Y is changed:110
```

### Task Scheduling

It is another excellent feature provided by this premier programming language. A programmer can schedule his job at any future time and set the time interval after which the task is required to be repeated. All these things can be achieved by the **Timer** and **TimerTask** class. The given example would make it clear.

#### Program 3.46

```
import java.util.*;
class X extends TimerTask{
 public void run ()
 {
 System.out.println("HelloWorld!!!");
 }
}
class Demo {
 public static void main (String args [])
 {
 X a=new X ();
 Timer tm=new Timer ();
 tm.scheduleAtFixedRate (a, 1000, 250);
 try{
 Thread.sleep (5000);
 } catch (InterruptedException e) {}
 tm.cancel ();
 }
}
```



## NOTES

never removes an element from the collection object. This interface has two methods.

### Methods

---

|                            |                                                            |
|----------------------------|------------------------------------------------------------|
| boolean hasMoreElements () | Checks if the Enumeration has any more elements or not     |
| Object nextElement ()      | Returns the next element that is available in Enumeration. |

---

### Program 3.47

```
import java.util.*;
class X{
 int i;
}
class Demo {
 public static void main(String args[])
 {
 ArrayList<X> arrX=new ArrayList<X>();
 for (int j=0;j<5;j++)
 {
 X a=new X();
 a.i=j;
 arrX.add(a);
 }
 System.out.println("Forward traversing");
 ListIterator<X> LiteR=arrX.listIterator();
 while(LiteR.hasNext()){
 X a=LiteR.next();
 System.out.println(a.i);
 }
 System.out.println("Backward traversing");
 while(LiteR.hasPrevious()){
 X a=LiteR.previous();
 System.out.println(a.i);
 }
 }
}
```

### Output of the program:

```
Forward traversing
0
1
2
3
4
Backward traversing
4
```

3  
2  
1  
0

The `hasPrevious()` and the `previous()` method behave in the opposite way to the `hasNext()` and the `next()` methods. **ListIterator** can also be used in `LinkedList`, `HashSet`, `TreeSet` to perform a similar task as that of an `ArrayList`.

### Map Interface

Map interface is not inherited from the `Collection` interface. Instead, the interface starts off its own interface hierarchy, for maintaining key-value associations. Map interface helps to establish a mapping between the keys to the corresponding element. A map cannot contain duplicate keys. In the process of mapping, each key can map at most one value.

Map interface has three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, and set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection view return their elements.

### Methods

---

|                                               |                                                                                                                       |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>void clear()</code>                     | Removes all the mappings from the current Map                                                                         |
| <code>boolean containsKey (Object key)</code> | Returns true or false accordingly, whether the current Map is mapping one or more keys to the value or not            |
| <code>boolean containsValue (Object v)</code> | Returns true when the current Map maps one or more keys to the value v, otherwise it returns false.                   |
| <code>Set entrySet()</code>                   | Returns a Set interface reference which specifies a view of the mappings established in the current Map.              |
| <code>boolean equals (Object mp)</code>       | Returns true when the object mp is equal to the current Map, or else it returns false.                                |
| <code>Object get (Object mp)</code>           | Returns value to which the current Map maps the key.                                                                  |
| <code>int hashCode()</code>                   | Returns an int value representing the hash-code value of the Map.                                                     |
| <code>boolean isEmpty()</code>                | Checks the current key value mappings. If the Map does not contain any mappings, then it returns true, or else false. |
| <code>Set keySet()</code>                     | Returns the Set interface reference which gives representation of keys in the Map.                                    |

### NOTES

## NOTES

|                                         |                                                                                              |
|-----------------------------------------|----------------------------------------------------------------------------------------------|
| <code>Object put (Object mapkey,</code> | Used to associate the mapvalue with the mapkey in the current Map.                           |
| <code>Object mapvalue)</code>           |                                                                                              |
| <code>Object remove (Object</code>      | Removes the mapping which is present in the Map for the key keyused.                         |
| <code>keyused)</code>                   |                                                                                              |
| <code>int size()</code>                 | Returns the total number of mappings established inside the Map.                             |
| <code>Collection values ()</code>       | Returns a collection reference which represents the view of the values contained in the Map. |

---

### SortedMap Interface

SortedMap is an interface which inherits from Map interface. This interface maintains its entries in ascending order. The sorting may be done in two ways. These are:

- According to the default ordering principle provided by Java.
- According to a user defined comparator explicitly provided by the programmer.

### Methods

---

|                                             |                                                                                                                                 |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>Comparator comparator ()</code>       | Returns the Comparator associated with the current SortedMap.                                                                   |
| <code>Object firstKey ()</code>             | Returns the currently lowest key in the SortedMap.                                                                              |
| <code>SortedMap headMap</code>              | Returns the reference of the current SortedMap whose keys are less than keyused                                                 |
| <code>(Object keyused)</code>               |                                                                                                                                 |
| <code>Object lastKey ()</code>              | Returns the currently highest key in the SortedMap                                                                              |
| <code>SortedMap subMap</code>               | Returns a reference of the portion of the SortedMap whose keys ranges from the key specified by sm1 to the key specified by sm2 |
| <code>(Object sm1, Object sm2)</code>       |                                                                                                                                 |
| <code>SortedMap tailMap (Object sm1)</code> | Returns the reference of the portion of the SortedMap whose keys are greater than or equal to the key as specified by sm1.      |

---

### HashMap Class

HashMap is a class that implements Map interface. It is a collection that stores the elements in the form of key value pair. The HashMap class holds only unique keys. This mean we cannot use duplicate data for keys in the HashMap. This class allows both null values and null key. HashMap is not synchronized and hence while using multiple threads on HashMap object, we get unreliable results.

There are two factors that affect the instance of the HashMap. They are the initial capacity and the load factor. The initial capacity determines the capacity

of the hashtable when it is created. The load factor determines how full the hashtable can become before its capacity is automatically increased. `HashMap` does not guarantee the order of its elements. This means the order in which the iterator reads the `HashMap` is not a constant.

We can write `HashMap` class as:

```
Class HashMap<K, V>
```

Where `K` represents the type of the key element and `V` represents the type of value element. For example, to store a `String` as key and `Integer` object as its value, we can create `HashMap` as,

```
HashMap<String, Integer> hm=new HashMap<String, Integer> ();
```

## Constructor

The constructor of the `HashMap` class is overloaded.

- `HashMap ()` : Constructs an empty map with default capacity and load factor.
- `HashMap (int cap)` : Constructs an empty map with the capacity specified by `cap` and default load factor.
- `HashMap (int cap, float load)` : Constructs an empty map with capacity specified by `cap`, and load factor specified by `load`.
- `HashMap (Map m)` : Constructs a map with the mappings specified `Map`.

## Methods

---

|                                            |                                                                                                                                   |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <code>void clear ()</code>                 | Removes all the key value pairs from the <code>Map</code> .                                                                       |
| <code>value get (Object key)</code>        | Returns the corresponding value when the key is given. If the key does not have a value associated with it, then it returns null. |
| <code>value put (key, value)</code>        | Stores <code>key-value</code> pair into the <code>HashMap</code> .                                                                |
| <code>Set&lt;k&gt; keySet ()</code>        | Converts <code>HashMap</code> into a <code>Set</code> where only keys will be stored.                                             |
| <code>Collection&lt;v&gt; values ()</code> | Returns all the values of the <code>HashMap</code> into a <code>Collection</code> object.                                         |
| <code>value remove (Object key)</code>     | Removes the key and corresponding value from the <code>HashMap</code> .                                                           |
| <code>boolean isEmpty ()</code>            | Returns <code>true</code> if there are no key-value pairs in the <code>HashMap</code> .                                           |
| <code>int size ()</code>                   | Returns number of key-value pairs in the <code>HashMap</code> .                                                                   |

---

## Program 3.48

```
import java.util.*;
class K {
 int key;
}
class V {
```

## NOTES

## NOTES

```
int data;
}
class Demo {
 public static void main (String args [])
 {
 HashMap<K, V> hashM = new HashMap<K, V> ();
 K keyArr [] = new K [5];
 for (int i = 0; i < 5; i++)
 {
 K key1 = new K ();
 key1.key = i;
 keyArr [i] = key1;
 V val = new V ();
 val.data = i + 5;
 hashM.put (key1, val);
 }
 for (int i = 0; i < 5; i++) {
 V a = hashM.get (keyArr [i]);
 System.out.println (a.data);
 }
 }
}
```

### Output of the program:

```
5
6
7
8
9
```

Data is stored in the **HashMap** along with the key value. If one looks at the `put` method, it keeps inserting data in the **HashMap** with the key value. Key is required to extract data from the **HashMap**. Therefore, an array is declared to store the key value. Then, each key value stored in the array is used to extract data from the **HashMap**. The `get ()` method only requires the key to extract the data.

### Hashtable Class

Hashtable is a predefined class present in `java.util` package. This class implements Hashtable datastructure to store element within the Hashtable object. We must specify the key and the value to be mapped to that key. The key is hashed and the hashcode is used as the index of the position in which the value is stored. Hashtable is same as HashMap but Hashtable is synchronized assuring proper results even if multiple threads act on it simultaneously.

We can write Hashtable class as:

```
class Hashtable<K, V>
```



Where  $K$  represents the type of key element and  $V$  represents the type of value element. For example, to store a `String` as key and `Integer` object as its value, we can create the `Hashtable` as:

```
Hashtable<String, Integer> hm=new
Hashtable<String, Integer>();
```

## Constructor

The constructor of the `HashTable` class is overloaded.

- `Hashtable()` : Constructs an empty hash table with default capacity and load factor.
- `Hashtable(int cap)` : Constructs an empty hash table with the capacity specified by `cap` and default load factor.
- `Hashtable(int cap, float load)` : Constructs an empty hash table with capacity specified by `cap`, and load factor specified by `load`.
- `Hashtable(Map m)` : Constructs a hash table with the mappings specified by `Map`.

## Methods

|                                            |                                                                                                                                                             |
|--------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void clear()</code>                  | Removes all the key value pairs from the <code>Hashtable</code> .                                                                                           |
| <code>value get (Object key)</code>        | Returns the corresponding value when the <code>key</code> is given. If the <code>key</code> does not have a value associated with it, then it returns null. |
| <code>value put (key, value)</code>        | Stores <code>key-value</code> pair into the <code>Hashtable</code> .                                                                                        |
| <code>Set&lt;k&gt; keySet ()</code>        | Converts <code>Hashtable</code> into a <code>Set</code> where only keys will be stored.                                                                     |
| <code>Collection&lt;v&gt; values ()</code> | Returns all the values of the <code>Hashtable</code> into a <code>Collection</code> object.                                                                 |
| <code>value remove (Object key)</code>     | Removes the <code>key</code> and corresponding value from the <code>Hashtable</code> .                                                                      |
| <code>boolean isEmpty()</code>             | Returns <code>true</code> if there are no <code>key-value</code> pairs in the <code>Hashtable</code> .                                                      |
| <code>int size ()</code>                   | Returns number of <code>key-value</code> pairs in the <code>Hashtable</code> .                                                                              |

## Program 3.49

```
import java.util.*;
class K {
 int key;
}
class V {
 int data;
}
class Demo {
 public static void main (String args[])
 {
```

## NOTES

## NOTES

```

HashMap<K, V> hashM = new HashMap<K, V> ();
 for (int i = 0; i < 5; i++)
 {
 K key1 = new K ();
 key1.key = i;
 V val = new V ();
 val.data = i + 5;
 hashM.put (key1, val);
 }

 Set<Map.Entry<K, V>> s = hashM.entrySet ();
 for (Map.Entry<K, V> x : s)
 {
 K z = x.getKey ();
 System.out.print ("For key: " + z.key + " Value:");
 V c = x.getValue ();
 System.out.println (c.data);
 }
}

```

### Output of the program:

```

For key: 2 Value:7
For key: 4 Value:9
For key: 3 Value:8
For key: 1 Value:6
For key: 0 Value:5

```

Here the object of Set **s** is going to hold the entries of HashMap in set format by the invocation of **entrySet () method** through hashM, the object of HashMap. The **entrySet () method** return the object of Set containing all the stored mapping elements in HashMap. It must be remembered that each element present in s is a reference of Map.Entry.

Entry is an inner class of Map. That is why, it is a reference of a Map.Entry. The getKey () method returns the reference of class **K** and the getValue () method returns the reference of class **V**. Now the output is straightforward.

### Difference between HashMap and Hashtable

| <i>HashMap</i>                                                                                                                                        | <i>Hashtable</i>                                                                                                                             |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• HashMap is not Synchronized in nature. In case of single thread HashMap is faster than HashTable.</li> </ul> | <ul style="list-style-type: none"> <li>• Hashtable is synchronized. In case of multiple threads Hashtable is better than HashMap.</li> </ul> |
| <ul style="list-style-type: none"> <li>• HashMap allows to store null and keys.</li> </ul>                                                            | <ul style="list-style-type: none"> <li>• Hashtable does not permit null values and keys.</li> </ul>                                          |
| <ul style="list-style-type: none"> <li>• Iterator in the HashMap is fail safe.</li> </ul>                                                             | <ul style="list-style-type: none"> <li>• Enumeration for the Hashtable is not fail safe.</li> </ul>                                          |
| <ul style="list-style-type: none"> <li>• It means Iterator will produce an exception if concurrent modification is made to the HashMap.</li> </ul>    | <ul style="list-style-type: none"> <li>• Enumeration never produces an exception to concurrent modification to Hashtable.</li> </ul>         |

## Vector Class

This is a predefined class present in `java.util` package. This class implements the `ArrayList`. Unlike `ArrayList` class, `Vector` class also supports dynamic array but it is synchronized. It means even if several threads act on `Vector` object simultaneously, the results will be reliable. Each `Vector` maintains a capacity and capacity increment. The capacity is always greater than or equal to the vector size.

## NOTES

### Constructor

The constructor of the `Vector` class is overloaded.

- `Vector()` : Constructs an empty vector. Its size is 10 and capacity increment is 0.
- `Vector(Collection c)` : Constructs a vector containing the elements of the `Collection c`. The elements are accessed in the order returned by the `Collection`'s iterator.
- `Vector(int cap)` : Constructs an empty vector with the capacity specified by `cap`.
- `Vector(int cap, int inc)` : Constructs an empty vector with the capacity specified by `cap`, capacity increment specified by `inc`.

### Method

Same methods of `ArrayList` class.

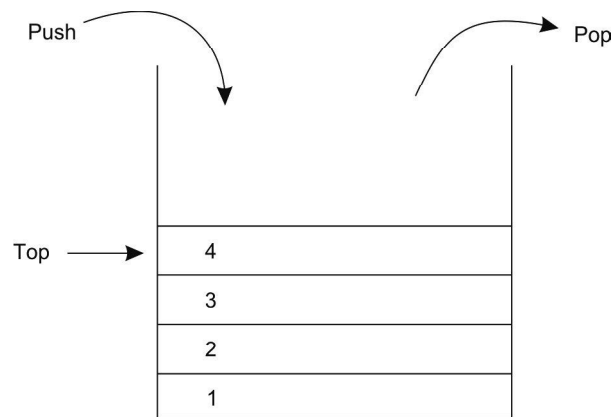
### Difference between ArrayList and Vector

| <i>ArrayList class</i>                                                                                                                                                                       | <i>Vector class</i>                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• <code>ArrayList</code> is not synchronized by default. In case of single thread <code>ArrayList</code> is faster than <code>Vector</code>.</li></ul> | <ul style="list-style-type: none"><li>• <code>Vector</code> is synchronized by default. In case of multiple threads <code>Vector</code> is better than <code>ArrayList</code>.</li></ul> |
| <ul style="list-style-type: none"><li>• <code>ArrayList</code> increases its size by 50 percent.</li></ul>                                                                                   | <ul style="list-style-type: none"><li>• A <code>Vector</code> increases its array size by doubling the size.</li></ul>                                                                   |
| <ul style="list-style-type: none"><li>• <code>ArrayList</code> has no default size.</li></ul>                                                                                                | <ul style="list-style-type: none"><li>• <code>Vector</code> has a default size of 10.</li></ul>                                                                                          |
| <ul style="list-style-type: none"><li>• <code>ArrayList</code> does not require any <code>Iterator</code> to display its contents.</li></ul>                                                 | <ul style="list-style-type: none"><li>• But <code>Vector</code> requires an <code>Iterator</code> to display its contents.</li></ul>                                                     |

### Stack Class

`Stack` class inherits from `Vector` class. A `Stack` represents a group of elements stored in LIFO (Last In First Out). It means that the element which is stored as a last element into the stack will be the first element to be removed from the stack. Inserting the elements (objects) into the stack is called 'Push Operation' and removing the elements from the stack is called 'Pop Operation'. Insertion and deletion of elements take place only from one side of the stack, called 'top' of the stack, as shown in the following figure.

## NOTES



**Fig. 3.11** Stack with Some Elements

## Methods

|                                         |                                                                                                                                                           |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean empty ()</code>           | Checks whether the stack is empty or not. if the stack is empty then it returns <code>true</code> , otherwise returns <code>false</code>                  |
| <code>element peek ()</code>            | Returns the top most object from the stack without removing it.                                                                                           |
| <code>element pop ()</code>             | Pops the top most element from the stack and returns it.                                                                                                  |
| <code>element push (element obj)</code> | Pushes an element <code>obj</code> onto the top of the stack and returns the element.                                                                     |
| <code>int search (Object ob)</code>     | Returns the position of the element <code>ob</code> from the top of the stack, if the element is not found in the stack then it returns <code>-1</code> . |

## Program 3.50

```
import java.util.*;
class X {
 int i;
}
class userComp implements Comparator<demo.X>
{
 public int compare(X a, X b) {
 return a.i-b.i;
 }
}
class Demo {
 public static void main (String args [])
 {
 TreeSet<X> treeS=new TreeSet<X> (new userComp ());
 for (int i=0;i<5;i++)
 {
 X a=new X ();
```

```
 a.i=i;
 treeS.add(a);
 }
 for(Xa:treeS)
 {
 System.out.println(a.i);
 }
}
}
```

### **Output of the program:**

```
0
1
2
3
4
```

`TreeSet<X> treeS=new TreeSet<X>(new userComp());`  
This statement tells the compiler to use the user-defined comparator rather than the default **Comparator**. Java implicitly invokes the `compare()` method of `userComp` class. The object for which the `compare()` method returns a positive value is stored next to the previously stored object. How to reverse this user-defined comparator? This can be shown by the following example:

### **Program 3.51**

```
import java.util.*;
class X {
 int i;
}
class userComp implements Comparator<demo.X>
{
 public int compare(X a, X b) {
 return -a.i+b.i;
 }
}
class Demo {
 public static void main(String args[])
 {
 TreeSet<X> treeS=new TreeSet<X>(new userComp());
 for(int i=0;i<5;i++)
 {
 Xa=new X();
 a.i=i;
 treeS.add(a);
 }
 for(Xa:treeS)
 {
```

## **NOTES**

```
 System.out.println(a.i);
 }
}
```

## NOTES

### Output of the program:

```
4
3
2
1
0
```

### Program 3.52

```
import java.util.*;
class Demo {
 public static void main (String args [])
 {
 LinkedList<Integer> linkedL=new LinkedList<Integer>();
 linkedL.add(-100);
 linkedL.add(99);
 linkedL.add(-99);
 linkedL.add(100);
 Comparator<Integer> revC=Collections.reverseOrder();
 // creation of a reverse comparator
 Collections.sort(linkedL, revC);
 // sorting the list according to reverse comparator
 System.out.println("Descending order:");
 for(Integer i:linkedL)
 System.out.println(i);
 Collections.shuffle(linkedL);
 // shuffling the elements present in the list
 System.out.println("After shuffling");
 for(Integer i:linkedL)
 System.out.println(i);
 System.out.println("Maximum
element:"+Collections.max(linkedL));
 System.out.println("Minimum
element:"+Collections.min(linkedL));
 }
}
```

### Output of the program:

```
Descending order:
100
99
-99
-100
```

```
After shuffling
99
100
-100
-99
Maximum element: 100
Minimum element: -100
```

The output is straightforward and simple. All the method names used in this program define their task.

### **Program 3.53**

```
import java.util.*;
class X {
 int i;
}
class Demo {
 public static void main (String args[])
 {
 Vector<X> vect=new Vector<X> ();
 for (int i=0;i<5;i++)
 {
 X a=new X ();
 a.i=i;
 vect.add(a);
 }
 X a=new X ();
 a.i=100;
 vect.addElement(a);
 for (X x:vect)
 System.out.println(x.i);
 }
}
```

#### **Output of the program:**

```
0
1
2
3
4
100
```

The `addElement ()` method behaves similar to the `add ()` method. It can be seen that a **Vector** is much similar to an `ArrayList`. Here, one can use `Iterator` to traverse the various elements present in a `Vector` object.

## **NOTES**

## NOTES

### Program 3.54

```
import java.util.*;
class X{
 int i;
}
class Demo {
 public static void main(String args[])
 {
 Vector<X> vect=new ArrayList<X>();
 for (int j=0;j<5;j++)
 {
 X a=new X();
 a.i=j;
 vect.add(a);
 }
 Iterator<X> iteR=vect.iterator();
 while(iteR.hasNext()){
 X a=iteR.next();
 System.out.println(a.i);
 }
 }
}
```

#### Output of the program:

```
0
1
2
3
4
```

### Program 3.55

```
import java.util.*;
class X{
 int i;
}
class Demo {
 public static void main(String args[])
 {
 Vector<X> vect=new Vector<X>();
 for (int i=0;i<5;i++)
 {
 X a=new X();
 a.i=i;
 vect.add(a);
 }
 Enumeration en=vect.elements();
 while(en.hasMoreElements()){
```



```
 Xa=(X)en.nextElement();
 System.out.println(a.i);
 }
 }
}
```

### **Output of the program:**

```
0
1
2
3
4
```

### **Program 3.56**

```
import java.util.*;
class Demo {
 public static void main (String args[])
 {
 Stack<Byte> s=new Stack<Byte> ();
 System.out.println("Initially the Stack:"+s);
 for (int i=0;i<10;i++) {
 byte b=(byte) i;
 s.push(b);
 System.out.println("Initially the Stack:"+s);
 }
 while (!s.empty()) {
 System.out.println("popped:"+s.pop());
 System.out.println(s);
 }
 }
}
```

### **Output of the program:**

```
Initially the Stack: []
Initially the Stack: [0]
Initially the Stack: [0, 1]
Initially the Stack: [0, 1, 2]
Initially the Stack: [0, 1, 2, 3]
Initially the Stack: [0, 1, 2, 3, 4]
Initially the Stack: [0, 1, 2, 3, 4, 5]
Initially the Stack: [0, 1, 2, 3, 4, 5, 6]
Initially the Stack: [0, 1, 2, 3, 4, 5, 6, 7]
Initially the Stack: [0, 1, 2, 3, 4, 5, 6, 7, 8]
Initially the Stack: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
popped: 9
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

## **NOTES**

*Multithreaded  
Programming, Applets,  
Handling String, java.  
lang and Utility Classes*

## NOTES

```
pooped: 8
[0, 1, 2, 3, 4, 5, 6, 7]
pooped: 7
[0, 1, 2, 3, 4, 5, 6]
pooped: 6
[0, 1, 2, 3, 4, 5]
pooped: 5
[0, 1, 2, 3, 4]
pooped: 4
[0, 1, 2, 3]
pooped: 3
[0, 1, 2]
pooped: 2
[0, 1]
pooped: 1
[0]
pooped: 0
[]
```

### Program 3.57

```
import java.util.*;
class K {
 int key;
}
class V {
 int data;
}
class Demo {
 public static void main (String args[])
 {
 Hashtable<K,V>hashT=new Hashtable<K,V>();
 K keyArr[]=new K[5];
 for (int i=0;i<5;i++)
 {
 K key1=new K();
 key1.key=i;
 keyArr[i]=key1;
 V val=new V();
 val.data=i+5;
 hashT.put (key1, val);
 }
 for (int i=0;i<5;i++) {
 V a=hashT.get (keyArr[i]);
 System.out.println(a.data);
 }
 }
}
```

```
 }
 }
}
```

### Output of the program:

```
5
6
7
8
9
```

### 3.9.1 Using Store () and Load ()

The **Properties** class of Java is used to maintain one or more properties that can be easily streamed into Text or Binary. The application properties contained in the **Properties** class instance can be persisted to a text file.

One of the most useful aspects of Java properties is that the information contained in a **Properties** object can be easily stored to or loaded from disk with the **store ()** and **load ()** methods. At any time, the user can write a **Properties** object to a stream or read it back. This makes property lists especially convenient for implementing simple databases.

The **store ()** method of the **Properties** class is used to save the application properties to a text file. This method takes an **OutputStream** or **Writer** object to store the information. Since it accepts **OutputStream** as well as **Writer**, in place of a text file, one can write the properties in a binary file as well. The most preferred way is writing it to a text file and preferred extension for the property file is **‘.properties’**.

The **java.util.Properties.store(OutputStream out,String comments)** method writes this property list (key and element pairs) in this **Properties** table to the output stream in a format suitable for loading into a **Properties** table using the **load(InputStream)** method.

#### Declaration

Following is the declaration for **java.util.Properties.store()** method:

```
public void store(OutputStream out,String comments)
```

#### Parameters

Following are the parameters defined in the above declaration.

- **out** – An Output Stream.
- **comments** – A Description of the Property List.

#### Return Value

This method returns the previous value of the specified key in this property list, or **null** if it did not have one.

### NOTES

## NOTES

### Exception

- **IOException** – If writing this property list to the specified output stream throws an **IOException**.
- **ClassCastException** – If this **Properties** object contains any keys or values that are not **Strings**.
- **NullPointerException** – If out is **null**.

### Check Your Progress

17. Why the wrapper classes are required?
18. What is autoboxing in java?
19. How the memory management is done?
20. Define the term class loader.
21. State about the `Locale` class.
22. What is the use of enumeration interface?
23. Write the definition of term stack.

## 3.10 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. There are two ways to create a thread in Java. One method is used exclusively in Java applications. The other method can be used in applications as well as in applets.
2. Threads are called lightweight processes because all the threads in a main application program share the same address space in the memory.
3. `MIN_PRIORITY`, `MAX_PRIORITY` and `NORM_PRIORITY` are the constants defined in `Thread` class.
4. The default priority of a thread is 5.
5. There are two ways to implement synchronization, which are as follows:
  - i. Synchronizing Methods
  - ii. Synchronizing Statements
6. The thread that is ready to run, but waiting for the processor availability is called runnable thread and the state is known as Runnable State.
7. Stream classes can be categorized into byte stream classes and character stream classes.
8. Byte stream classes support input and output operations on bytes (8-bit bytes). Whereas, character stream classes perform input and output operations on characters (16-bit Unicode).
9. `InputStreamReader` class is used to read input from the console input device.
10. `FileReader` class creates character stream between the file and the program and reads characters from the file and sends it to the program.

11. An applet is a small program typically embedded within the Web page which is used to create a dynamic and interactive application. They provide interactive features to a Web page which cannot be provided by HTML.
12. A Java applet enters into various states during its entire life cycle which include born state, running state, idle state and dead state.
13. The `volatile` modifier is used to tell the compiler that the variable declared as `volatile` can be changed at any time by the other parts of the program. This modifier is mainly used in multithreading in which a program (process) is divided into two or more subprograms (subprocesses), each of which runs by a separate thread and performs different tasks concurrently.
14. The modifier `strictfp` can be used on classes, interfaces and non-abstract methods. When applied to a method, it causes all calculations inside the method to use strict floating point mathematics. When applied to a class, all calculations inside the class use strict floating point mathematics. Compile time constant expressions must always use strict floating point behaviour.
15. One of the vital features of JNI (Java Native Interface) is that it never imposes any restriction on the JVM. Therefore, JVM vendors can add support for the JNI without affecting other parts of the virtual machine.
16. The string class supports several constructors. The various string classes are:
  - String Constructors
  - String Length
  - String Literals
  - String Concatenation
  - String Compare
17. The wrapper classes are required because of the following reasons:
  - `Vector`, `ArrayList`, `LinkedList`, classes present in `java.util` package cannot handle primitive data types like `int`, `char`, `float`, etc. Hence primitive data types may be converted into object types by using wrapper classes present in `java.lang` package.
  - Wrapper classes convert primitive data types into objects.
18. J2SE 5 supports the autoboxing process by which a primitive type is automatically encapsulated into its equivalent type wrapper class object. There is no need to explicitly construct a wrapper class object. This technique is popularly known as autoboxing in Java.
19. Memory management is completely done by JVM. Java code `p.class` file is created after compiling the above. During the execution of `p.class` file, first it is loaded in the memory by the boot strap class loader, which is a component of (Java Virtual Memory) JVM. `Z` is a reference variable which is created inside the stack area. When the constructor of class `p` is called through a `new` operator, a chunk of memory is allocated from the

## NOTES

## NOTES

heap area and its starting address is stored in z. Therefore, it can be said that the object of class p is created.

20. Class loader is an abstract class which extends the Object class. In Java, a class is loaded in the memory either by a boot strap class loader or by user defined class loader. Boot strap class loader is a component of Java Virtual Machine. When a class file is executed by javac command, the boot strap class loader is responsible for loading the class in the memory.
21. Local class is used to describe a geographical, political or cultural region. In this class the date, time, numbers format is used to display it according to the customs of the user's native country, region or culture.
22. Enumeration interface is used to retrieve the elements one by one Iterator but this interface never removes an elements from the collection object.
23. A Stack represents a group of elements stored in LIFO (Last In First Out). It means that the element which is stored as a last element into the stack will be the first element to be removed from the stack. Inserting the elements (objects) into the stack is called 'Push Operation' and removing the elements from the stack is called 'Pop Operation'. Insertion and deletion of elements take place only from one side of the stack, called 'Top' of the stack.

---

## 3.11 SUMMARY

---

- The concept of threads and the idea of multithreading was introduced a decade back, however, it has been accepted into the mainstream programming only recently.
- The concept of threads and the idea of multithreading was introduced a decade back, however, it has been accepted into the mainstream programming only recently.
- Thread scheduling specifies exactly in what order, your thread will be run. There are two strategies: non pre-emptive scheduling and pre-emptive time slicing.
- A thread is just like a program which has a single flow of control. It also has a starting point, the execution part and an end. The main programs in the preceding examples can be called single-threaded programs. Java also allows us to use multiple flows of control in a program and such a program is known as multithreaded program.
- Threads are called lightweight processes because all the threads in a main application program share the same address space in the memory.
- Java program always contains at least one thread, even if we do not create one. This thread is called main thread and it is the one which immediately starts executing when we start a program.
- MIN\_PRIORITY, MAX\_PRIORITY and NORM\_PRIORITY are the constants defined in Thread class.

- When multiple threads need access to a single resource, there must be a way to ensure that only one thread will use the resource at any given point of time, otherwise it may lead to a severe problem.
- The objective of synchronization is to control the access to shared resources.
- Synchronization uses the concept of monitor. A monitor is an object which is used as a mutually exclusive lock. That is, it can be owned by only one thread at any given point of time.
- A thread in Java shows the program's path of execution. Logically, threading means successfully execution of a program that shows the required output. Java programming is known as multi-threaded because implementation of Java threading and OS implementation are done accordingly with it.
- The thread that is ready to run, but waiting for the processor availability is called runnable thread and the state is known as Runnable State.
- The predefined methods `suspend ()`, `resume ()` and `stop ()` have been deprecated in Java 2 though they are a convenient way for managing the execution of threads.
- Java manages all input and output in the form of streams. A stream refers to a channel through which data flows from the source to the destination. This data is in the form of sequence of bytes or characters.
- Stream classes can be categorized into byte stream classes and character stream classes.
- Byte stream classes support input and output operations on bytes (8-bit bytes). Whereas, character stream classes perform input and output operations on characters (16-bit Unicode).
- `InputStreamReader` class is used to read input from the console input device.
- `FileReader` class creates character stream between the file and the program and reads characters from the file and sends it to the program.
- An applet is a small program typically embedded within the Web page which is used to create a dynamic and interactive application. They provide interactive features to a Web page which cannot be provided by HTML.
- A Java applet enters into various states during its entire life cycle which include born state, running state, idle state and dead state.
- Object serialization is the process of reading and writing objects. By default, all objects are serializable, i.e., they can be read from and written to the secondary memory so that the value which they hold persists. To make an object non-serializable, the `transient` modifier is used.
- The `volatile` modifier is used to tell the compiler that the variable declared as `volatile` can be changed at any time by the other parts of the program. This modifier is mainly used in multithreading in which a program (process) is divided into two or more subprograms (subprocesses), each of which runs by a separate thread and performs different tasks concurrently.

## NOTES

## NOTES

- The modifier `strictfp` can be used on classes, interfaces and non-abstract methods. When applied to a method, it causes all calculations inside the method to use strict floating point mathematics. When applied to a class, all calculations inside the class use strict floating point mathematics. Compile time constant expressions must always use strict floating point behaviour.
- One of the vital features of JNI (Java Native Interface) is that it never imposes any restriction on the JVM. Therefore, JVM vendors can add support for the JNI without affecting other parts of the virtual machine.
- A `String` is a sequence of characters. Java provides full complement features of string handling by implementing strings as built-in objects.
- In Java `String` is a class that represents an immutable string, which represents a sequence of character that can never change. Any modification to the `String` will have to create a new `String` object. But a `StringBuffer` is a mutable `String` object that can be modified at runtime.
- J2SE 5 supports the autoboxing process by which a primitive type is automatically encapsulated into its equivalent type wrapper class object. There is no need to explicitly construct a wrapper class object. This technique is popularly known as autoboxing in Java.
- Memory management is completely done by JVM. Java code `p.class` file is created after compiling the above. During the execution of `p.class` file, first it is loaded in the memory by the boot strap class loader, which is a component of (Java Virtual Memory) JVM. `Z` is a reference variable which is created inside the stack area. When the constructor of class `p` is called through a `new` operator, a chunk of memory is allocated from the heap area and its starting address is stored in `z`. Therefore, it can be said that the object of class `p` is created.
- Class loader is an abstract class which extends the `Object` class. In Java, a class is loaded in the memory either by a boot strap class loader or by user defined class loader. Boot strap class loader is a component of Java Virtual Machine. When a class file is executed by `javas` command, the boot strap class loader is responsible for loading the class in the memory.
- A security manager is referred as an object that specifically defines a security policy for an application. This policy typically specifies those actions that are unsafe or sensitive.
- `Local` class is used to describe a geographical, political or cultural region. In this class the date, time, numbers format is used to display it according to the customs of the user's native country, region or culture.
- Enumeration interface is used to retrieve the elements one by one `Iterator` but this interface never removes an elements from the collection object.
- A `Stack` represents a group of elements stored in LIFO (Last In First Out). It means that the element which is stored as a last element into the stack will be the first element to be removed from the stack. Inserting the elements (objects) into the stack is called 'Push Operation' and removing the elements



from the stack is called 'Pop Operation'. Insertion and deletion of elements take place only from one side of the stack, called 'Top' of the stack.

- The `Properties` class of Java is used to maintain one or more properties that can be easily streamed into Text or Binary. The application properties contained in the `Properties` class instance can be persisted to a text file.

## NOTES

---

### 3.12 KEY TERMS

---

- **Multithread programs:** Programs that use multiple flows of control in a program in Java.
- **Priority:** Integers which specify the relative priority of one thread over another.
- **Synchronization:** A technique to ensure that only one thread will use the resource at any given point of time, when multiple threads try to access a single resource.
- **Runnable state:** The thread that is ready to run, but waiting for the processor availability is called runnable thread and the state is known as Runnable State.
- **Stream:** Refers to a channel through which data in the form of sequence of bytes or characters flows from source to destination.
- **FileReader class:** Creates character stream between the file and the program and reads characters from the file and sends it to the program.
- **FileWriter class:** Used to write characters from the.
- **Applet:** A small program typically embedded within the Web page which is used to create a dynamic and interactive application.
- **Exception handling:** Diverting the processing to a part of the program when an exception occurs.
- **Memory management:** Memory management is completely done by (Java Virtual Machine) JVM.

---

### 3.13 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

#### Short-Answer Questions

1. Define the term thread scheduling.
2. What is the use of runnable interface?
3. How do you set priorities for threads?
4. Which keyword is used to synchronize a method or a block of statement?
5. How do you implement a runnable interface?

## NOTES

6. How will you define the term `suspend ()` , `resume ()` and `stop ()` .?
7. What are stream classes?
8. Differentiate between `FileReader` and `FileWriter` .
9. Define the term `PrintWriter` .
10. What is the order of method invocation when an applet is loaded?
11. Differentiate between the `transient` and the `volatile` modifier.
12. Define the term Native method.
13. Write the some important methods of `String` class.
14. Differentiate between `String` and `StringBuffer` .
15. Write the difference between autoboxing and unboxing.
16. Give difference between output of `p.java` and `q.java` .
17. Write the process class and runtime class.
18. How the interacting with security manager is done?
19. Differentiate between observable class and observer interface.
20. State about the `Math` class.
21. Define the term `Map` interface.
22. what do you understand by the term `Store ()` and `Load ()` in Java.

### Long-Answer Questions

1. Briefly explain the multithreaded programming giving appropriate examples.
2. Discuss briefly java thread model with the help of example.
3. Analyse the thread priorities with the help of example.
4. What is synchronization? When do we use it? Explain by giving examples of different ways to implement synchronization.
5. Describe the `Thread` class and `Runnable` interface giving appropriate example programs.
6. Briefly explain the multiple threads and its types with the help of examples.
7. Write a program to demonstrate `suspend` , `resume ()` and `stop ()` operations.
8. Discuss briefly streams (byte and character) with the help of relevant examples.
9. Java supports reading input provided by the user with the help of a console input device, such as keyboard. Explain with the help of example programs.
10. Analyse the `FileReader` and `FileWriter` class by giving appropriate example programs.
11. Describe the fundamentals of an applet with the help of diagram relevant examples.
12. Discuss about the `transient` and `volatile` modifier.

13. Discuss about the modifier `strictfp` with the help of example programs.
14. Briefly explain the native method and its problem with the help of example programs.
15. Discuss about the String handling with the help of example programs.
16. Describe the operation and String and String comparison method.
17. Analyse the methods of `StringBuffer` giving appropriate examples.
18. Briefly explain the significance of wrapper classes.
19. Describe the memory management in Java with the help of relevant examples.
20. Analyse the system class giving appropriate examples.
21. Discuss about the `ThreadLocal` class with the help of example programs.
22. Describe security manager and `SecurityManager` class with the help of example programs.
23. Explain the `StringTokenizer` class with the help of example.
24. Briefly explain the hash table class and vector class giving appropriate example programs.

## NOTES

---

### 3.14 FURTHER READING

---

- Balagurusamy, E. 2007. *Programming with Java*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Naughton, Patrick and Herbert Schildt. 1999. *Java 2: The Complete Reference*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Das, Rashmi Kanta. 2013. *Core Java for Beginners*, 3rd Edition. New Delhi: Vikas Publishing House Pvt. Ltd.
- Schildt, Herbert. 2006. *Java: The Complete Reference*, 7th Edition. New Delhi: Tata McGraw-Hill.
- Hunter, Jason and William Crawford. 2001. *Java Servlet Programming*, 2nd Edition. California: O'Reilly Media.
- Arnold, Ken, James Gosling and David Holmes. 2005. *The Java Programming Language*, 4th Edition. Boston: Addison-Wesley.
- Wigglesworth, Joe and Paula Lumby. 1999. *Java Programming Advanced Topics*, 2 Edition. Boston: Course Technology.
- Deitel, Paul and Harvey Deitel. 2011. *Java: How to Program*, 9th Edition. New Delhi: Prentice-Hall of India.



---

# UNIT 4 INPUT/OUTPUT CLASSES, NETWORKING, AWT GRAPHICS AND TEXT, CONTROLS, LAYOUTS AND MENUS

---

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

## NOTES

### Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 File and File Name in Java
  - 4.2.1 Directory and Creating Directory
- 4.3 Stream Classes
- 4.4 Basic of Networking
  - 4.4.1 Proxy Server
  - 4.4.2 Domain Naming Services
  - 4.4.3 Networking Classes and Interfaces
  - 4.4.4 InetAddress Class
  - 4.4.5 Datagram Packet Network
- 4.5 Applet Basic
  - 4.5.1 Applet Life Cycle
  - 4.5.2 Simple Banner Applet
  - 4.5.3 Handling Events
  - 4.5.4 AudioClip
- 4.6 AWT Classes
  - 4.6.1 Window Fundamentals
  - 4.6.2 Working With Frame Windows
  - 4.6.3 Frame Window and Event Handling in a Frame Window
  - 4.6.4 Display Information While Working with Graphics and Color
  - 4.6.5 Working with Fonts
- 4.7 AWT Controls and Layout Managers
  - 4.7.1 AWT Menus
  - 4.7.2 Dialog Class
- 4.8 Answers to 'Check Your Progress'
- 4.9 Summary
- 4.10 Key Terms
- 4.11 Self Assessment Questions and Exercises
- 4.12 Further Reading

---

## 4.0 INTRODUCTION

---

File handling is one of the basic operations provided by different programming languages. Java treats all the standard input devices and output devices as files. It supports file handling through streams and objects of File class. Files and streams are required to perform I/O operation. A file is a container which contains data or information. Files are located in the secondary storage of the system. When a file is opened for any purpose, first it is loaded into the primary memory, then the

## NOTES

operation is performed. After any write operation, changes are reflected according to the implementation of either write through or write back protocol. Files are normally of two types; binary files and text files. Java provides two ways to handle files. One is through the various streams provided in java.io package and other is through File class which does not require streams to operate.

Network programming and AWT (Abstract Window Toolkit) in Java. At the core of the network programming is the concept of a socket. Basically, a socket is referred as a single endpoint for a two way communication connection between the two different programs operating/running on the network. Java supports creation of sockets and exchange information using different protocols through the classes defined in java.net package. Java applets are used to create graphical and user interactive applications. The applets designed earlier, were graphical in nature. However, they did very little when it came to demonstrating their user interactive capability. User interaction means a specific action is generated and an appropriate result is displayed to the user when he interacts with an application by clicking the mouse or entering a character using the keyboard. In Java, all the activities that occur between the user and the application are termed as events. Events play a significant role in applet programming as they facilitate the inclusion of user interactivity in applets.

An applet is a Java program that is compiled on one computer and can be run on other computers through Java-enabled web browsers or Java tools, such as applet viewer. It uses the platform-independent features of the Java programming language. Applets bring live content, such as news, animation or scorecard to the static web pages that can be loaded from the Internet or from the local disk.

AWT, an abbreviation for Abstract Window Toolkit, provides several graphics, windowing and user interface tools which are used to develop GUI of applets as well as stand-alone applications running in GUI environment.

In this unit, you will study about the file and file name filter interface, directory, creating directory, stream classes, input stream and output stream, file input stream and file output stream, byte array input stream and byte array output stream, filtered byte stream and buffered Bytestream, print stream, access file stream, stream tokenizer and benefits, basic of networking, proxy server, domain naming services, networking classes and interface, InetAddress class and TCP/IP sockets, datagram packet, applet, applet life cycle, simple banner applet, handling events, getDocumentBase (), getCodeBase (), showDocumentBase (), AWT classes, window fundamentals, working with frame windows, frame window in an applet, event handling in a frame window, window program, displaying information while working with graphics and color, working with fonts and managing text output, exploring text and graphics, control fundamentals and layouts, menus, dialog class.

---

## 4.1 OBJECTIVES

---

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

- Discuss the file and file name filter interface
- Analyse the directory and creating directory

- Understand the stream classes
- Describe the input stream and output stream
- Explain the file input stream and file output stream
- Analyse the byte array input stream and byte array output stream
- Understand the filtered byte stream and buffered byte stream
- Elaborate on the print stream and access file stream
- Describe the stream tokenizer and benefits
- Explain the basic of networking
- Define the proxy server and domain naming services
- Understand the networking classes and interface
- Discuss the InetAddress class and TCP/IP sockets
- Describe the datagram packet
- Define applet and applet life cycle
- Explain the simple banner applet and handling events
- Analyse the getDocumentBase (), getCodeBase (), showDocumentBase ()
- Understand the AWT classes and window fundamentals
- Explain the working with frame windows and frame window in an applet
- Define the event handling in a frame window and window program
- Elaborate one the displaying information while working with graphics and color
- Understand the working with fonts and managing text output
- Analyse the exploring text and graphics
- Explain the control fundamentals and layouts
- Define the menus and dialog class

## NOTES

---

## 4.2 FILE AND FILE NAME IN JAVA

---

By the use of **File** class, one can directly deal with the **files**, directories and **file** system of the platform. Actually, Java does not provide a crystal clear view of how things are done in the background when the programmer uses the **File** class. A programmer has to create the **File** object through the constructors provided by the **File** class. Then, using those objects, he/she can perform manipulation with the **files** and directories.

**File** class constructors are used for the creation of the object of the **File** class. These constructors are overloaded. The various forms of these constructors are given below:

```
File f1 = new File ("c: / Minerva / ravenX");
```

Here it has been shown how to open the existing **file** in a window platform.

```
File f2 = new File ("c: / Minerva ", "rian");
```

## NOTES

Here, the first parameter is the absolute path and the second parameter is the **file** that has to be opened.

```
File f3 = new File ("Java", "temp");
```

Here, the first parameter is a directory name in which the existing **file** temp has to be opened.

A novice programmer may ask, why are **files** required? The answer is, **files** are required to have a persistent storing of data. Sometimes, one needs to read the data from **files** rather than a standard input device like a keyboard. It may also be stated that in Java, directories are also treated as **files**. If one is dealing with a directory, then one has to use the `list ()` method provided by Java to list out all the **files** residing in the directory.

Java is quite smart when it comes to deal with a path separator. In Windows, one can use both `.`. As in Microsoft, if one wants to use `\`, then the escape sequence `\\` has to be used. However, in UNIX and Linux, one can use `.`.

## Methods

By the help of the predefined methods of a **File** class, a programmer can retrieve the properties of a **file**.

```
String getName ()
```

This method returns the name of the **file** through which this method is invoked

```
String getPath () and String getAbsolutePath ()
```

These two methods are used to get the absolute path of the **file** through which it is invoked.

```
String getParent ()
```

This method returns the name of the parent directory of the **file** through which it is invoked.

```
boolean exists ()
```

This method checks whether the **file** through which it is invoked, exists or not.

```
boolean isFile () and boolean isDirectory ()
```

These two methods are used to determine whether the **File** object through which it is invoked, is a directory or a **file**.

```
import java.io.File;
```

One needs to import this package to have various methods to deal with a **File** object.

## Program 4.1

```
class Demo
{
 public static void main(String args[])
 {
 File myfile=new File ("/dir1/pex");
 System.out.println("Name: "+myfile.getName());
 System.out.println("Path: "+myfile.getPath());
 System.out.println("My absolute path:
 "+myfile.getAbsolutePath());
 }
}
```



```
System.out.println("My parent : "+myfile.getParent());
System.out.println("Name: "+myfile.getName());
if(myfile.exists())
 System.out.println("file does exist!!!");
else
 System.out.println("file does not exist");
if(myfile.canRead())
 System.out.println("file is readable ");
else
 System.out.println("file is not readable");
if(myfile.canWrite())
 System.out.println("file is writeable ");
else
 System.out.println("file is not writeable");
if(myfile.isFile())
 System.out.println("It is a normal file");
else
 System.out.println("It is not a normal file might be system
file like device driver.");
System.out.println("file was last
modified"+myfile.lastModified());
System.out.println("size of the file is in
bytes"+myfile.length());
}
}
```

#### **boolean canRead() and boolean canWrite ()**

These methods are used to check if one can read from the specified **file** or write into the specified **file** respectively. These methods return a **boolean** value, depending on readability/write ability of the **file**.

#### **long lastModified()**

This method returns the last modification time of the **file**.

#### **Use of lastModified(), canWrite() and canRead() Methods**

This can be clarified with the following examples:

#### **Program 4.2**

```
import java.io.*;
public class File2
{
 public static void main(String args[])
 {
 File f1 = new File ("c:/Java","abc.txt");
 if(f1.canRead())
 System.out.println ("we can read from this file");
 else
 System.out.println ("we cannot read from this file");
 if(f1.canWrite())
```

## **NOTES**

## NOTES

```
System.out.println("we can write to this file");
else
System.out.println("we cannot write to this file");
System.out.println("The file was last modified
at"+f1.lastModified()+"seconds after January 1 1980");
}
}
```

### Output of the program:

```
C:\java>java File2
we can read from this file
we can write to this file
The file was last modified at 1230222210000 seconds after January 1 1980
```

The **file** used in the above example has both read and write option. Therefore, that is the output. However, the output of `lastModified()` method is interesting. It shows the time period in millisecond after which the **file** is modified with respect to 1<sup>st</sup> January, 1980.

### How to rename a file?

The **renameTo ()** method is used to **rename** an existing **file**.

### Program 4.3

```
import java.io.*;
class Demo
{
 public static void main (String[] args)
 {
 File myFile=new File ("/dir1/pex");
 boolean b1=myFile.renameTo ("Minarva");
 if (b1)
 System.out.println ("Rename operation is successful");
 else
 System.out.println ("File cannot be renamed");
 }
}
```

The `renameTo ()` method returns boolean true value if **rename** is done successfully, otherwise false is returned.

### Deleting an existing file

The **deletion** of a **file** can be performed with the help of `delete ()` method. On successful deletion of a **file**, the method returns a boolean true, otherwise it returns a boolean false.

### Program 4.4

```
import java.io.*;
class Demo
{
```

```
public static void main (String [] args)
{
 File myFile = new File ("dir1/pex");
 boolean b1 = myFile.delete ();
 if (b1)
 System.out.println ("file is deleted");
 else
 System.out.println ("File cannot be deleted ");
}
}
```

Here, `delete ()` method returns `true` to the boolean variable **b1** if the **file** is deleted successfully, else **b1** will get a `false` value. The output will be printed accordingly.

There is another method `deleteOnExit ()`, which deletes the **file** when one completes the operation `exit` from the execution phase of the program.

**File** class also provides some other useful methods as discussed below.

### Use of `length ()` Method

This method is used to know the **file** size in bytes. Another example is given below:

#### Program 4.5

```
import java.io.*;
public class File3
{
 public static void main (String args [])
 {
 File f = null;
 for (int i = 0; i < args.length; i++)
 {
 f = new File ("c:/Java", args [i]);
 }
 File f1 = new File ("c:/Java/renfile");
 if (f.exists ())
 {
 System.out.println (f + " exists");
 System.out.println ("its size is " + f.length () + " bytes");
 f.renameTo (f1);
 System.out.println ("Renamed file name : " + f1);
 System.out.println ("deleting the file " + f);
 System.out.println ("=====");
 f.delete ();
 } else {
 System.out.println (f + " does not exist");
 }
 }
}
```

## NOTES

## NOTES

```
}
}
```

In this program, the name of the **file** is passed through a command line argument which has to be renamed. This **file** is held by **f**. Then another **File** object **f1** is created whose name is **renfile**. Then the **rename ()** method is invoked through **f** and this method takes the **File** object as argument which holds the new name. Upon the successful execution of this method, the desired **file** is renamed.

### To check the Space Available in a Specified File

J2SE 6 provides three methods to get the various attributes associated with a particular partition where the **file** resides. The given example shows this

#### Program 4.6

```
import java.io.*;
class Demo
{
 public static void main (String [] args)
 {
 File myFile=new File ("/dir1/pex");
 if(myFile.exists()==false)
 {
 System.out.println("The specified file does not
exist");
 return ;
 }
 long x=myFile.getFreeSpace ();
 x=x/1000;
 System.out.println("Amount of space available in MB: "+x);
 x=myFile.getTotalSpace ();
 x=x/1000;
 System.out.println("total space in MB: "+x);
 }
}
```

One can easily guess the output. Methods like **getFreeSpace ()** and **getTotalSpace ()**, when invoked on the specific **file**, check the free space available and the total space available respectively in terms of bytes. The returned value of these two methods has been divided by 1000 to get the values in MBs.

### How to Make a File Read Only?

This can be shown by the following example:

#### Program 4.7

```
import java.io.*;
class Demo
{
```

```
public static void main (String [] args)
{
 File myFile = new File ("/dir1/peX");
 if (myFile.exists () == false)
 {
 System.out.println ("file does not exist");
 return;
 }
 boolean b1 = myFile.setReadOnly ();
 if (b1)
 System.out.println ("Operation is successful");
 else
 System.out.println ("Operation failed");
 }
}
```

The `setReadOnly ()` method is used to make a read only **file**.

### 4.2.1 Directory and Creating Directory

A **directory** is a collection of **files** and **directories**. In Java, **directories** are also treated as **files**. If one wishes to deal with the **directories**, then the `list ()` method can be used. When the `list ()` method is invoked by the **directory** object (created through **File** class constructor), then the list of other **files** and **directories** are extracted from it. This method is overloaded. One of them is `String [] list ()`. The program given below clarifies this.

#### Program 4.8

```
import java.io.File;
class DirectoryList
{
 public static void main (String args [])
 {
 String directory_name = "/Minerva";
 File myFile = new File (directory_name);
 if (myFile.isDirectory () == true)
 {
 System.out.println ("Directory of" + directory_name);
 String s1 [] = myFile.list ();
 for (int i = 0; i < s1.length; i++)
 {
 File f1 = new File (directory_name + "/" + s1 [i]);
 if (f1.isDirectory ())
 {
 System.out.println (s1 [i] + " is a directory");
 }
 }
 }
 else
 {
```

## NOTES

## NOTES

```
 System.out.println(sl[i] + " is simply a file!!");
 }
}
else
{
 System.out.println(directory_name + " is not a directory");
}
}
```

Output is straightforward. Method names clearly indicate what their tasks are.

### Use of `mkdir()` Method

This method is used to create a directory and it returns a boolean `true/false` indicating the success/failure of the creation. The following example denotes the usage of this method.

#### Program 4.9

```
import java.io.*;
public class File5
{
 public static void main(String args[]) throws IOException
 {
 File myFile=new File("c:/Alice/wonderLand");
 if(myFile.mkdir()==true)
 System.out.println("created a directory");
 else
 System.out.println("Unable to create a directory");
 }
}
```

Run the program by `C:\java>java File5`

#### Output of the program:

```
created a directory
```

If the `mkdir()` method is successfully executed the boolean `true` value is returned, else the method is going to return `false`.

### Use of Overloaded Form of `list()` Method

In this form of `list()` method, it is used to filter out the required **files** from a directory. Often one feels that a particular set of **files** has to be opened or listed out, instead of all the **files** present in the directory.

#### Program 4.10

```
import java.io.*;
public class File4 implements FilenameFilter
{
 String w;
 public File4(String w)
```

```
{
 this.w="."+w;
}

public boolean accept (File dir, String name)
{
 return name.endsWith(w) ;
}

public static void main (String args[]) throws IOException
{
 for (int p=0;p<args.length;p++)
 {
 File f1 = new File ("e:/cobra/applet");
 FilenameFilter only=new File4 (args[p]);
 String s[]=f1.list (only) ;
 System.out.println ("printing files with
"+args[p]+"extension in the "+f1.getPath()+"
directory");

 for (int i=0;i<s.length;i++)
 System.out.println(s[i]);
 }
}
```

Run the program by C:\java>java File4 java

### **Output of the program:**

```
printing files with java extension in the C:\java directory
File1.java
File2.java
File3.java
File4.java
```

In this program, the overloaded `list()` method has been invoked. This also implicitly invokes the `accept()` method. Inside the `accept()` method, the `endsWith()` method is invoked. The entire operation performs only one thing, i.e., it filters out the required **files**.

---

## **4.3 STREAM CLASSES**

---

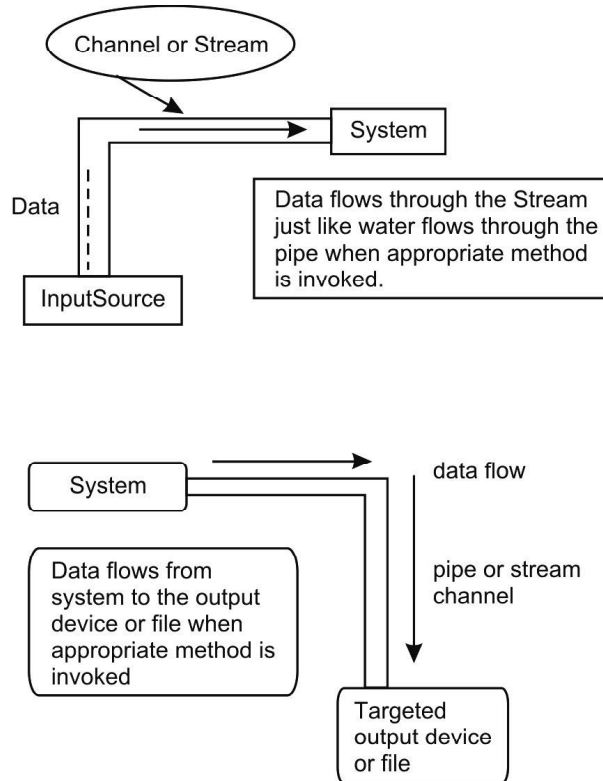
In Java I/O (Input/Output) streams are flow of data you can either read from, or write to. Streams are typically linked to a data source, or data destination, like a file, network connection, etc.

A stream has no concept of an index of the read or written data, as an array does. Nor can you typically move forth and back in a stream, unlike you do in an array, or in a file using `RandomAccessFile`. A stream is just a continuous flow of data.

A **stream** means a channel or a pipe. Like flow of water in a pipe, data flows from the source to the destination through the channels in Java.

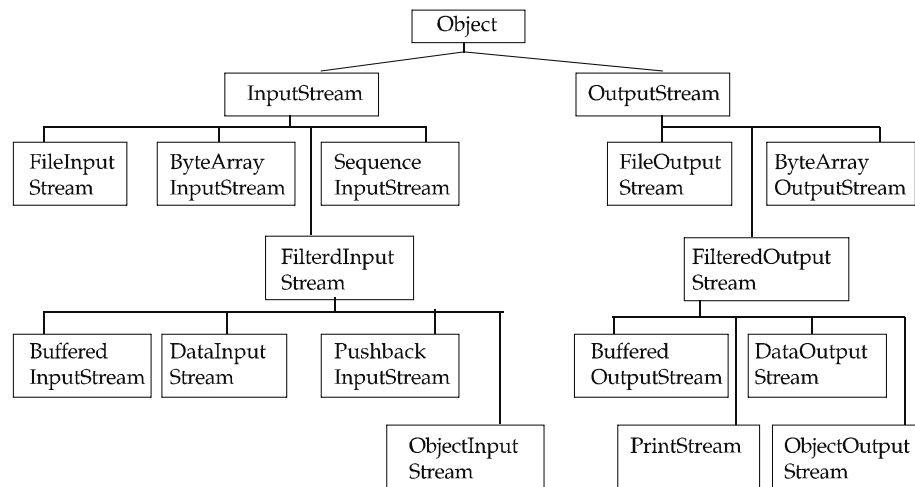
## **NOTES**

**NOTES**



**Fig. 4.1** Data flow

These channels are the objects of various **Stream** classes provided by Java. Java provides two types of **streams**: input **streams** and output **streams**. By the use of input **streams**, one receives the data from the source and by the use of output **streams** one writes the data at the desired destination. Java provides two ways to perform read and write operations. `Reader` class objects read and `Writer` class objects write the data in the form of characters. **Stream** class objects read and write in the form of bytes. Classes that end with the term **Reader**, deal with `char` datatype and the classes that end with the term **Stream** deal with bytes. A hierarchy structure of a **Stream** class is given below:



**Fig. 4.2** Hierarchy of Stream Class



With the help of the following table, one can see the various retain input **Stream** classes and their functions.

With by the help of these retain input **Stream** classes, the system directly reads the data from **file** and buffer in byte format.

**Table 4.1** Class, Function and Supported Methods

| Class                | Function                                                                                     | Supported methods                                                      |
|----------------------|----------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| ByteArrayInputStream | Allows a buffer in memory to be used as an InputStream                                       | available(), mark(), markSupported(), read(), insert(), skip()         |
| FileInputStream      | For reading information from a file                                                          | available(), close(), finalize(), read(), skip()                       |
| FilterInputStream    | Abstract class providing interface for useful functionality to the other InputStream classes | available(), close(), mark(), markSupported(), read(), reset(), skip() |

With the help of the next table, one can see various Output **Stream** classes and their methods through which data is written in targeted output sources like **file** and buffer in byte format.

**Table 4.2** Class, Function and Supported Methods

| Class                 | Function                                                                                         | Supported methods                                              |
|-----------------------|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| ByteArrayOutputStream | Creates a buffer in memory<br>All the data one sends to the streams is placed in this buffer.    | reset(), size(), toByteArray(), toString(), write(), writeTo() |
| FileOutputStream      | Abstract class providing an interface for useful functionality to the other OutputStream classes | close(), flush(), write()                                      |

### Low-Level Stream

**Low-Level Input Streams** have methods that read input and return the input as bytes. On the other hand, **Low-level output streams** have methods that are supplied with bytes and they write the bytes as output.

#### FileInputStream

It is a class that helps to read the data from a **file**. When the programmer wants to read the data from the **file** by using a `FileInputStream` and if that **file** is not present, then program is terminated at the runtime by throwing `FileNotFoundException`. The `read()` method of a `FileInputStream` returns an `int` which contains the byte value of the byte read. If the `read()` method returns `-1`, there is no more data to read in the stream, and it can be closed. That is, `-1` as `int` value, not `-1` as byte value.

### NOTES

## NOTES

There is a difference here. There are two types of constructors available with this class. The first constructor takes the name of the **file** as a **String** argument

```
FileInputStream f = new FileInputStream("c:/Java/
temp.exe");
```

The second constructor takes the **File** class object as an argument

```
File f = new File ("c:/Java/temp.exe");
FileInputStream f1=new FileInputStream(f);
```

### **FileOutputStream**

This class helps to create a new **file** and write the data into byte format. Two types of constructors are applicable to this class. The `write()` method of a `FileOutputStream` takes an `int` which contains the byte value of the byte to write. The first constructor takes the filename as a string argument

```
FileOutputStream f=new FileOutputStream("c:/Java/
temp.exe");
```

The second constructor takes the **File** class object as an argument

```
File f = new File("c:/Java/temp.exe");
FileOutputStream f1=new FileOutputStream(f);
```

In the case of **FileOutputStream**, if the programmer writes the data into a read-only **file**, then the program generates `IOException`.

### **Program 4.11**

```
import java.io.*;
public class ReadWriteFile
{
 public static byte get () [] throws Exception
 {
 byte in[]=new byte[50];
 System.out.println("enter the text.");
 System.out.println("only 50 bytes of data is stored in the
 array");
 System.out.println("press enter after each line to get
 input into the program");
 for (int i=0;i<50;i++)
 {
 in[i]=(byte) System.in.read();
 }
 return in;
 }
 public static void main (String args []) throws Exception
 {
 byte input []=get ();
 FileOutputStream f=new FileOutputStream("c:/Java/
 write.txt");
```

```
 for (int i=0;i<50;i++)
 {
 f.write(input[i]);
 }
 f.close ();
 int size;
 FileInputStream fl=new FileInputStream("c:/Java/
write.txt");
 size=fl.available ();
 System.out.println("reading contents of file write.
Text");

 for (int i=0;i<size;i++)
 {
 System.out.print ((char) fl.read ());
 }
 f.close ();
 }
}
```

Run the program by C:\>java ReadwriteFile

### Output of the program:

Enter the text.

Only 50 bytes of data is stored in the array

Press enter after each line to get input into the program

Z c x c z x z x v z x v v z v v c z v v c c

Xcxxxzvzx v zvv vcv c vv vzvvvvz

Reading contents of write.Text

Z c x c z x z x v z x v v z v v c z v v c c

Xcxxxzvzx v

The execution begins from the `main()` method. Here input is an array of bytes. The `get()` method is static in nature, hence it can be invoked from the `main()` method directly. The `get()` method returns a reference to an array of bytes. Inside the `get()` method body, an array of bytes of length 50 is declared. The statement `System.in.read()` is invoked to take the input from keyboard and the input is stored in the array of bytes. Finally, the reference to array of bytes, i.e., `in` is returned from the `get()` method and held by `input`. Next the **FileOutputStream** object `f` is connected to the file "write.txt". When the `write()` method is invoked through `f` inside for loop, the data present in the byte array is written to "write.txt". The `write()` method takes the reference to array of bytes, i.e., `input` as its argument. After the writing is complete, `f` closes the 'write.txt' file by calling the `close()` method.

## NOTES

## NOTES

### ByteArrayInputStream

This class is a subclass of `InputStream` in which input data comes from a specified array of `byte` values. This is useful for reading data in memory as if it were coming from a file, pipe, or socket. Note that the specified array of bytes is not copied when a `ByteArrayInputStream` is created.

```
ByteArrayInputStream b = new ByteArrayInputStream
byte buf[])
```

This constructor takes a **byte array** as its parameter. Through this constructor, a programmer takes the input from the specified array of bytes.

```
ByteArrayInputStream b=new ByteArrayInputStream
(byte buf [], int off, int len)
```

In this constructor, **off** is the offset of the first byte to be read and **len** is the number of bytes to be read into the array.

### ByteArrayOutputStream

This class implements a buffer, which can be used as an **OutputStream**. The size of the buffer increases as data is written into the **Stream**. The data is retrieved, using the methods `toArray()` and `toString()`. It has two types of constructors.

```
ByteArrayOutputStream o = new ByteArrayOutputStream
Stream()
```

This creates a buffer of 32 bytes to store the data.

```
ByteArrayOutputStream o=new ByteArrayOutputStream
(int size)
```

The above constructor creates a buffer of size `int`. The methods of this class return `void` and throw an `IOException` or error conditions.

### Program 4.12

```
import java.io.*;
public class ByteArray
{
 public static void main (String args[]) throws Exception
 {
 ByteArrayOutputStream f=new ByteArrayOutputStream(12);
 System.out.println("enter 10 characters and press the enter
 key");
 System.out.println("These will be converted to uppercase and
 displayed");
 while (f.size() !=10)
 {
 f.write(System.in.read());
 }
 System.out.println("Accepted characters in the array");
 byte b[]=f.toByteArray();
 System.out.println("displaying characters in the array");
 for (int i=0; i<b.length; i++)
```

```
 {
 System.out.println((char)b[i]);
 }
 ByteArrayInputStream inp=new ByteArrayInputStream (b) ;
 int c;
 System.out.println("Converted to upper case characters");
 for (int i=0;i<1;i++)
 {
 while ((c=inp.read()) != -1)
 {
 System.out.print(Character.toUpperCase ((char)c));
 }
 System.out.println();
 inp.reset();
 }
 }
 }
```

Run the program by C:\java ByteArray

### **Output of the program:**

Enter 10 characters and press the enter key

These will be converted to uppercase and displayed

Learn java From Book

Accepted characters in the array

Displaying characters in the array

L

e

a

r

n

j

a

v

a

Converted to upper case characters

LEARN JAVA

Program execution begins from `main()`. `f` is an object of **ByteArrayOutputStream**.

The constructor of **ByteArrayOutputStream** takes integer value as its argument. The argument defines the size of the byte array or buffer that is attached to `f`.

```
f.write(System.in.read());
```

## **NOTES**

## NOTES

The statement which appeared in the above program, is a bit complex in nature. Here the `write()` method is invoked through `f`. Inside the `write` method, the `read()` method is invoked through `System.in`. The statement `System.in.read()` reads the character entered from the keyboard and returns it. Then that returned character is written into the buffer held by `f`. Each time when the statement is executed, one character is read from the keyboard and written into the buffer. This process continues until the loop expires.

```
byte b []=f.toByteArray();
```

The `toByteArray()` method is invoked through `f` which converts the buffer into an array of bytes and `b` holds that array. By this technique, one can fetch the data from the array of bytes through the index.

```
ByteArrayInputStream inp=new ByteArrayInputStream
(b);
```

The `inp` is an object of `ByteArrayInputStream`. The constructor takes `b` as its argument. The `inp` holds the buffer according to the size of `b`. The `read()` method is invoked through `inp` and the process continues until the loop expires.

### SequenceInputStream

This is the child class of `InputStream` class. In this class two or more other inputstreams are combined into one. Firstly, all bytes from the first input stream is iterated and returned, then the bytes from the second input stream. This class reads the data sequentially from two or more input sources.

### Constructor

```
SequenceInputStream (InputStream is1, InputStream
is2);
```

### Program 4.13

```
import java.io.*;
public class Sequence
{
 public static void main (String args[])
 {
 try{
 FileInputStream fis1=new FileInputStream("c:/a.txt");
 FileInputStream fis2=new FileInputStream("c:/b.txt");
 SequenceInputStream s =new SequenceInputStream (fis1, fis2);
 int ch;
 while ((ch=s.read()) !=-1)
 {
 System.out.print ((char) ch);
 }
 fis1.close();
 fis2.close();
 s.close();
 } catch (FileNotFoundException fe)
 {
```

```

 fe.printStackTrace();
 }
 catch (IOException ie)
 {
 ie.printStackTrace();
 }
}

```

Run the program by C:\>java Sequence

### Output of the program:

Java Is A Robust Language

Java Is Leader Of Internet

**fis1** holds the file **a.txt**. **fis2** holds the file **b.txt**. Here, the **s** is an object of `SequenceInputStream` class. This particular class is used when one needs to take the input from different sources sequentially. Through **s**, different **streams** merge together to take the input. Given below is the constructor of `SequenceInputStream`.

```

SequenceInputStream s =new SequenceInputStream
(fis1, fis2);

```

It takes two **files** as its argument. The process of extraction of data continues until the end of **file** of **fis2** is reached. The `read()` method is invoked through the object **s** to read the data from the **files**.

`FilterInputStream` class has the child classes named `BufferedInputStream`, `DataInputStream`, `PushbackInputStream`.

`FilterOutputStream` class has the child classes named `BufferedOutputStream`, `DataOutputStream`, `PrintStream`.

**Table 4.3** Class, Function and Supported Methods

| Class                            | Function                                                                                                                       | Supported Methods                                                                                                                                                             |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DataInputStream</code>     | Used in context with <code>DataOutputStream</code> . Hence one can read data.                                                  | <code>read()</code> , <code>readBoolean()</code> , <code>readByte()</code> , <code>readFloat()</code>                                                                         |
| <code>ObjectInputStream</code>   | Primitives ( <code>int</code> , <code>char</code> , <code>long</code> , etc.) from a stream can be read in a portable fashion. | <code>readFully()</code> , <code>readInt()</code> , <code>readLine()</code> , <code>readLong()</code> , <code>readShort()</code> , <code>skipBytes()</code>                   |
| <code>BufferedInputStream</code> | Use of this class facilitates to read data from the buffer.                                                                    | <code>available()</code> , <code>mark()</code> , <code>markSupported()</code> , <code>read()</code> , <code>reset()</code> , <code>skip()</code>                              |
| <code>PushbackInputStream</code> | Has a one-byte pushback buffer, so the last character is pushed back.                                                          | <code>available()</code> , <code>mark()</code> , <code>reset()</code> , <code>read()</code> , <code>unread()</code> , <code>read</code> , can be <code>markSupported()</code> |

## NOTES

## NOTES

| Class                | Function                                                                                                                      | Supported Methods                                                                                                                                      |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| DataOutputStream     | Used in context with DataInputStream. Hence one can write primitives (int, char, long, etc.) to a stream in portable fashion. | flush(), size(), write(), writeBoolean(boolean b), writeDouble(double d), writeFloat(float f), writeInt(int i), writeLong(long l), writeShort(short s) |
| PrintStream          | For producing formatted output. While DataOutputStream handles the storage of data, PrintStream handles display.              | checkError(), close(), flush(), print(), println(), setError(), write(),                                                                               |
| BufferedOutputStream | This is used to prevent a physical write every time one sends a piece of data. flush() can be called to flush the buffer      | flush(), write()                                                                                                                                       |

### High-Level Stream

**High-Level Input Streams** take their input from other input streams, whereas **high-level output streams** direct their output to other output streams. **High-level input streams** are: BufferedInputStream, DataInputStream and ObjectInputStream, etc. **High-level output streams** are: BufferedOutputStream, DataOutputStream, ObjectOutputStream, PrintStream, etc.

#### BufferedInputStream

This class accepts the input by using a buffered array of bytes that act as cache and it utilizes the mark() and reset() method. Chunks of bytes from the buffered array can be chosen and read. The **BufferedInputStream** class maintains an internal array of characters in which it buffers the data that it reads from its source. The default size of the buffer is 2048 bytes. A **BufferedInputStream**, is beneficial in certain situations where reading a large number of consecutive bytes from a data source is not significantly more costly than reading a single byte. This class constructor is overloaded.

```
BufferedInputStream bis = new BufferedInputStream
(InputStream is);
```

It creates a buffered input stream with a 2048 byte buffer.

```
BufferedInputStream bis=new BufferedInputStream
(InputStream
is,int bufsize);
```

It creates a buffered input stream with an internal buffer of bufsize bytes. If the bufsize is less than 0, then it throws IllegalArgumentException.

#### BufferedOutputStream

The output is stored in a buffered array of bytes, which acts as a cache for writing. Data written in the **BufferedOutputStream** will continue until and unless the buffer is full. This class constructor is overloaded.



```
BufferedOutputStream b= new BufferedOutputStream
(OutputStream os) ;
```

It creates a **BufferedOutputStream** with a default 512 byte buffer.

```
BufferedOutputStream b=new BufferedOutputStream
(OutputStream
os, int bufsize) ;
```

It creates a **BufferedOutputStream** with a buffer of `bufsize` bytes. If the `bufsize` is less than 0 then the program is terminated by throwing `IllegalArgumentException`.

### **DataInputStream**

This class reads bytes from another **stream** and translates them into Java primitives, char array and `String` by the help of some pre-defined methods. These methods are:

- **byte readByte () throws IOException**
- **boolean readBoolean () throws IOException**
- **short readShort () throws IOException**
- **char readChar () throws IOException**
- **int readInt ( ) throws IOException**
- **float readFloat () throws IOException**
- **long readLong () throws IOException**
- **double readDouble () throws IOException**
- **String readLine ( ) throws IOException**

The constructor is:

```
DataStream dis=new DataInputStream(InputStream
is) ;
```

### **DataOutputStream**

This class supports the writing of primitive data types of Java to output sources. A set of methods exists in this class to write the data to the output source in any primitive data types format. These methods are:

- **void writeByte (byte b) throws IOException**
- **void writeBoolean (boolean b) throws IOException**
- **void writeShort (short s) throws IOException**
- **void writeChar (char c) throws IOException**
- **void writeInt (int i) throws IOException**
- **void writeFloat (float f) throws IOException**
- **void writeLong (long l) throws IOException**
- **void writeDouble (double d) throws IOException**

## **NOTES**

## NOTES

The constructor is:

```
DataOutputStream dos=new DataOutputStream
(OutputStream os);
```

In order to create a simple text **file**, one can use a **FileOutputStream**.  
An example is given below:

### Program 4.14

```
import java.io.*;
class Demo{
 public static void main (String [] args) throws IOException
 {
 DataInputStream din=new DataInputStream (System.in);
 FileOutputStream fout=new FileOutputStream ("myFile.txt");
 System.out.println ("Press # to save & quit the file");
 char ch;
 while ((ch= (char) din.read ()) != '#')
 fout.write (ch);
 fout.close ();
 }
}
```

`System.in` represents the standard input device, i.e., keyboard. **din** is an object of **DataInputStream** class. This is what one calls a channel or **stream**. Now **din** is connected to the keyboard. Next **fout** is an object of **FileOutputStream** class. It also behaves like a channel. It is connected to the output **file** `myFile.txt`. Inside, while loop `read()` method is invoked through the object **din**, `read()` method reads the data from the keyboard through the channel named **din** and the returned character value is saved in the character variable **ch**. Next, the `write()` method is invoked and the stored value in **ch** is moved to `myfile.txt` through the output channel object `fout`. All this operation continues till **#** is pressed by the user. Then, finally, `close()` method is invoked to close the operation and to save the **file**.

### How to append a file?

If the user has to use the **file** `myFile.txt` again to write something more at the end, then the previous data will be lost. So, some changes are required to be made in the above program to append a **file**.

### Program 4.15

```
import java.io.*;
class Demo
{
 public static void main (String [] args) throws IOException
 {
 DataInputStream din=new DataInputStream (System.in);
 FileOutputStream fout=new FileOutputStream
("myFile.txt",true);
```

```
System.out.println("Press # to save & quit the file");
char ch;
while((ch=(char)din.read())!=#)
 fout.write(ch);
fout.close();
}
```

The extra parameter passed in the constructor of **FileOutputStream** may be seen. This boolean true value opens the **file** in the append mode.

### **Use of BufferedOutputStream**

In order to improve the performance, one has to use the **BufferedOutputStream**. How is the performance improved? In the previous program, each time a character is read from the keyboard, the same is written to the **file** and each time, in the background an appropriate system call is made to do the operation. This involves a lot of overhead. What else can one do? A buffer, i.e., a temporary storage area, can be created and the data stored there, till the input operation is completed. Then the entire content of the buffer can be written into the file at once. An example is given below:

### **Program 4.16**

```
import java.io.*;
class Demo
{
 public static void main(String[] args) throws IOException
 {
 DataInputStream din=new DataInputStream(System.in);
 FileOutputStream fout=new FileOutputStream("myFile.txt");
 BufferedOutputStream br=new BufferedOutputStream(fout);
 System.out.println("Press # to save & quit the file");
 char ch;
 while((ch=(char)din.read())!=#)
 br.write(ch);
 br.close();
 }
}
```

**br** is an object of **BufferedOutputStream**. It is a buffer or a chunk of memory attached with **fout**. Now, the read method is invoked through **din**. When the `write()` method is invoked through **br**, the characters entered from keyboard are stored in the temporary buffer **br** until the buffer is filled or **#** is pressed. If the buffer is full and the user is still entering the data, then **IOException** will be generated. If the programmer does not mention the size of the buffer explicitly, then the default size is 512 bytes.

## **NOTES**

## NOTES

### Steps to be Remembered

The various steps are:

- Create an input stream object and connect it to keyboard by  
`DataInputStream din=new  
DataInputStream(System.in);`
- Create an output stream object and connect it to the **file** where one will write something by `FileOutputStream fout=new  
FileOutputStream("myFile.txt");`
- Create a buffer and attach it to an output stream object by:  
`BufferedOutputStream br= new  
BufferedOutputStream(fout)`
- Read from the keyboard by the `din.read()` method retain;
- Write to the **file** by `bout.write()` method.

### Reading a File

Data stored in a **file** can be read by the use of **FileInputStream** object.

#### Program 4.17

```
import java.io.*;
class Demo
{
 public static void main(String[] args) throws IOException
 {
 FileInputStream fin=new FileInputStream("myFile.txt");
 int ch;
 while((ch=(char)fin.read())!=-1)
 System.out.print((char)ch);
 fin.close();
 }
}
```

Java detects the end of **file** when it encounters `-1`. To indicate the end of **file**, OS stores `-1`. At the end of every **file**, **fin** is an object of `FileInputStream`. It is simply an input channel connected to `myFile.txt`. When `read method()` is invoked by **fin**, data is read from the **file** through the channel **fin**, stored in **ch** and then displayed on the monitor. This process continues till `-1` is encountered.

### Copying the Content of One File to Another

This can be clarified with the following example:

#### Program 4.18

```
import java.io.*;
class Demo
{
 public static void main(String[] args) throws IOException
 {
```

```
FileInputStream fin=new FileInputStream("myFile1.txt");
FileOutputStream fout=new FileOutputStream("myFile2.txt");
int ch;
while((ch=fin.read())!=-1)
 fout.write((char)ch);
fout.close();
}
```

**fin** is a `FileInputStream` object which is connected to `myFile1.txt`. The object **fin** is used to extract the data from the source that is connected to it. **fout** is an object of `FileOutputStream` class which is connected to `myFile2.txt`. This stream is used to write the data in the destination. Inside, loop **fin** reads the character from `myFile1.txt` through `read()` method and stores it in the character **ch**. Next, this character **ch** is written in the file `myFile2.txt` through **fout** by the invocation of `write()` method.

### Use of `BufferedReader` to Improve Performance

This can be clarified with the following example:

#### Program 4.19

```
import java.io.*;
class Demo
{
 public static void main(String[] args) throws IOException
 {
 try
 {
 FileInputStream fin=new FileInputStream("myFile.txt");
 }
 catch (FileNotFoundException e)
 {
 System.out.println("file does not exists.")
 return;
 }
 BufferedReader br=new BufferedReader (fin);
 int ch;
 while((ch=br.read())!=-1)
 System.out.print((char)ch);
 fin.close();
 }
}
```

In this program, `BufferedReader` has been used to improve the performance. The `BufferedReader` object **br** is connected to **fin**, the object of `InputStreamReader`. Here, the `read()` method is invoked through **br** to read the data from `myfile.txt`.

## NOTES

## NOTES

### Use of DataOutputStream

This can be clarified with the following example:

#### Program 4.20

```
import java.io.*;
public class DataStream
{
 public static void main (String args[]) throws IOException
 {
 BufferedReader d=new BufferedReader (new
 InputStreamReader(new FileInputStream ("c:/
 temp.txt"));

 DataOutputStream o=new DataOutputStream (new
 FileOutputStream ("C:/templ.txt"));
 String line;
 while ((line=d.readLine()) !=null)
 {
 String a=(line.toUpperCase ());
 System.out.println(a);
 o.writeBytes (a+"\r\n");
 }
 d.close();
 o.close();
 }
}
```

Suppose the temp.txt file contains the line 'Learn Java as it is an object oriented language' as its content. The output appears as shown below:

Run the program by C:\java DataStream

#### Output of the program:

```
LEARN JAVA AS IT IS AN OBJECT ORIENTED LANGUAGE
```

### PushbackInputStream

This class is used to read a character from the **InputStream** and return the same. This is done without disturbing the **InputStream**. This class allows the most recently read byte to be put back into the stream, as if it had not yet been read.

#### Constructor

The various constructors are:

**PushbackInputStream (InputStream is)**

**PushbackInputStream (InputStream is, int size)**

### PrintStream

This class is used to write the text or primitive data types. Primitives are converted to character representation. The methods of this class are widely used in Java

applications. The two methods that are very familiar. These are: `System.out.println()` and `System.out.print()`.

### Constructor

The various constructors are:

```
PrintStream (String s)
PrintStream (OutputStream os)
PrintStream (OutputStream os ,boolean b)
```

### Serialization

It is the process of writing the state of an object to a byte stream. It is a technique that is required when the programmer wants to save the state of the object in a persistent storage area. Later on, the programmer restores the objects by using the process **Deserialization**. In other words, **serialization** is a technique of storing object contents into a **file**. **Serializable** interface is an empty or marker interface without any members in it. Marking interface is useful to mark the object of a class for special purposes. `Static` and `transient` variables are not serialized. **Deserialization** is the process of reading back the object from the **file**. A programmer reads the object from a stream by the help of the `ObjectInputStream`.

### Constructor

The constructor is,

```
ObjectInputStream (InputStream in) throws
IOException.
```

The serialized objects should be read through the object `in`.

*Table 4.4 Methods and their description*

| <i>Methods</i>                   | <i>Description</i>                                                     |
|----------------------------------|------------------------------------------------------------------------|
| <code>int available()</code>     | Returns the number of bytes that are available in the input sources.   |
| <code>void close()</code>        | Closes the invoking <code>String</code>                                |
| <code>int read()</code>          | Returns the integer representation of the next available byte of input |
| <code>Object readObject()</code> | Returns the <code>Object</code> from the invoking <code>Stream</code>  |
| <code>long skip(long n)</code>   | Skips <code>n</code> number of bytes from the input sources            |

`ObjectOutputStream` class is used to write the objects to a stream.

### Constructor

```
ObjectOutputStream (OutputStream out) throws
IOException
```

Through the `outputstream` object, the **serializable** objects are written in the output sources.

### NOTES

**Table 4.5: Methods and their Description**

| Methods                                   | Description                                     |
|-------------------------------------------|-------------------------------------------------|
| <code>void close()</code>                 | Closes the invoking stream                      |
| <code>void flush()</code>                 | Finalizes the output sources                    |
| <code>void write(byte b[])</code>         | Writes an array of bytes to the invoking stream |
| <code>void writeObject(Object obj)</code> | Writes Object obj to the invoking stream        |

## NOTES

### Program 4.21

```
import java.io.*;
class Ex1 implements Serializable
{
 int i, j;
 transient int k;
 void show(int i, int j, int k)
 {
 this.i=i;
 this.j=j;
 this.k=k;
 }
}
public class Serial
{
 public static void main(String args[])
 {
 try{
 Ex1 e1=new Ex1 ();
 e1.show(20,30,40);
 FileOutputStream fos=
 new FileOutputStream("c:/s1.txt");
 ObjectOutputStream oos=
 new ObjectOutputStream(fos);
 oos.writeObject(e1);
 FileInputStream fis=new FileInputStream("c:/s1.txt");
 ObjectInputStream ois=new ObjectInputStream(fis);
 Ex1 e2=(Ex1)ois.readObject();
 System.out.println("Data Is "+e2.i+"\t"+e2.j+"\t"+e2.k);
 }catch (Exception e)
 {
 e.printStackTrace();
 }
 }
}
```

Run the program by C:\>java Serial



## Output of the program:

Data Is 20 30 0

## Externalizable Interface

```
public interface Externalizable extends Serializable
```

The identity of the class of an `Externalizable` instance is mentioned in the serialization stream and the responsibility of the saving and restoring the contents of its instance lies with that class. The `writeExternal()` and `readExternal()` methods of the `Externalizable` interface are implemented by a class thus giving it complete control over the format and contents of the stream for an object and its supertypes. These methods must explicitly coordinate with the supertype to maintain its state. These methods surpass customized implementations of `writeObject` and `readObject` methods.

Object Serialization uses the `Serializable` and `Externalizable` interfaces. Object persistence mechanisms can use them as well. Each object to be stored is tested for the `Externalizable` interface. If the object supports `Externalizable`, the `writeExternal()` method is called. If the object does not support `Externalizable` but implements `Serializable`, the object is saved using `ObjectOutputStream`.

Reconstruction of `Externalizable` object creates an instance by using the public no-arg constructor, then the `readExternal()` method is called. `Serializable` objects are restored by reading them from an `ObjectInputStream`.

An `Externalizable` instance can designate a substitution object via the `writeReplace()` and `readResolve()` methods documented in the `Serializable` interface.

## Methods

```
public void writeExternal(ObjectOutput out) throws
IOException
```

The object implements the `writeExternal` method to save its contents. It is done by either calling the methods of `DataOutput` for its primitive values or calling the `writeObject` method of `ObjectOutput` for objects, strings, and arrays.

```
public void readExternal(ObjectInput in) throws
IOException
```

The object implements the `readExternal` method to restore its contents by calling the methods of `DataInput` for primitive types and `readObject` for objects, strings and arrays.

## StreamTokenizer

Java provides in-built method for pattern matching from data extracted from the input stream. The pattern matching is done by breaking the `InputStream` into tokens which are later delimited by a set of characters.

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

## NOTES

## NOTES

### Program 4.22

```
import java.io.*;
public class wordcounter
{
 public static void main (String args []) throws IOException
 {
 FileReader fr=new FileReader ("c:/temp.txt");
 StreamTokenizer input=new StreamTokenizer (fr);
 int tok;
 int count=0;
 while ((tok=input.nextToken ()) !=input.TT_EOF)
 {
 if (tok==input.TT_WORD)
 System.out.println ("word found :"+input.sval);
 count++;
 }
 System.out.println ("found "+count + " words in temp.txt");
 }
}
```

Run the program by C:\>java wordcounter

#### Output of the program:

```
word found : Learn
word found : java
word found : as
word found : it
word found : is
word found : an
word found : object
word found : oriented
word found : language
found 9 words in temp.txt
```

StreamTokenizer defines four integer fields named **TT\_EOF**, **TT\_EOL**, **TT\_NUMBER** and **TT\_WORD**. There exists another variable **ttype**, the token recognizing variable. **ttype** is equal to **TT\_WORD**, if `nextToken()` method recognizes the element as word. If the element is a number, then **ttype** is equal to **TT\_NUMBER**. If the token is a simple character, then **ttype** contains the value of that character. If the element is end of line, then **ttype** is equal to **TT\_EOL**. Similarly, when end of file is reached, **ttype** is equal to **TT\_EOF**.

#### Reader and Writer Classes

The difference between readers and input stream is that while the readers are able to **read** characters, the input **streams read** bytes. This increases the power of the Java classes by being able to **read** any character and thus enabling internalization. To say it in simple terms, it is possible to **write** Java programs in languages like German, French, Japanese, etc.

The functionality of the writers is similar to the output **streams** and it is possible to write one block of bytes or characters.

The following section deals with a few sub-classes of **Reader** and **Writer** classes.

## NOTES

### Reader Class

Some of the sub-classes of the Reader class are:

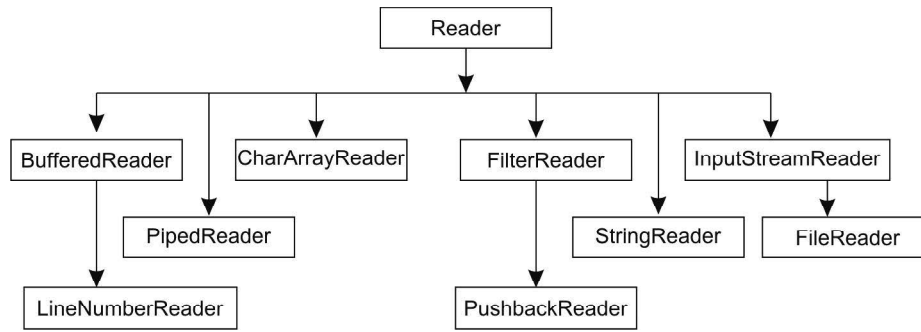


Fig. 4.3: Sub-Classes of Reader Class

### FileReader

The **FileReader** class enables reading character **files**. It uses default character encoding. Its usage is similar to `FileInputStream` class and its constructors are identical to those of `FileInputStream` class. The constructor is given below:

```
public FileReader (File f)
```

This constructor can throw a `FileNotFoundException`.

### CharArrayReader

The **CharArrayReader** allows the usage of a character array as an `InputStream`. The usage of **CharArrayReader** class is similar to `ByteArrayInputStream`. The constructors are given below:

- `CharArrayReader (char c[])`
- `CharArrayReader (char c[], int start, int num)`

### InputStreamReader

This class **reads** bytes from an input stream and converts them to characters according to a mapping algorithm. The default mapping identifies bytes as common ASCII characters and converts them to unicode characters of Java. The constructor is given below:

```
public InputStreamReader (InputStream istream)
```

### FilterReader

This class allows the reading of filtered character **streams**. There is one instance variable **in** which is a protected reference to the **Reader** that is being filtered.

```
Protected FilterReader (Reader in)
```

### BufferedReader

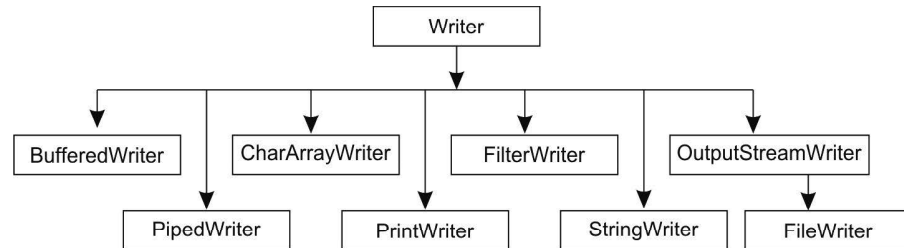
This class accepts a **Reader** object as its parameter and adds a buffer of characters to it. This class is mainly useful because of its `readLine ( )` method.

```
public BufferedReader (Reader r)
```

## NOTES

### Writer Class

A few of the sub-classes of the **Writer** class are:



**Fig. 4.4:** Sub-Classes of *Writer* Class

Reader works exclusively with 16-bit chars, designed for Unicode.

### FileWriter

The **FileWriter** allows **writing** character **files**. It uses the default character encoding and buffer size. The usage of **FileWriter** class is similar to that of `FileOutputStream` class.

The constructor is given below and it can throw an `IOException`.

```
public FileWriter (File f)
```

### Program 4.23

```
import java.io.*;
public class FileRead
{
 public static void main (String args [])
 {
 try
 {
 BufferedReader br=new BufferedReader (
 new InputStreamReader (System.in));
 System.out.println ("Enter The Text");
 String s=br.readLine ();
 char c[]=s.toCharArray ();
 FileWriter fw=new FileWriter ("File1.txt");
 fw.write(c);
 fw.close ();
 System.out.println ("Read The Data From The File");
 FileReader fr=new FileReader ("File1.txt");
 int ch;
 while ((ch=fr.read ())!=-1)
 {
```

```
 System.out.print((char)ch);
 }
 fr.close();
}
catch(Exceptione)
{
 e.printStackTrace();
}
}
}
```

Run the program by F:\rashmi\io>java FileRead

### **Output of the program:**

Enter The Text

Learn Java

Read The Data From The File

Learn Java

#### **CharArrayWriter**

This class uses character array as the `OutputStream`. The constructor of the class is overloaded.

```
CharArrayWriter ()
CharArrayWriter (int num)
PrintWriter
```

This class contains methods that make the generation of formatted output simple. It can be used instead of `PrintStream`. The constructor is:

```
public PrintWriter (OutputStream ostream)
```

The stream is not flushed each time the `println()` method is called.

#### **FilterWriter**

This class is used to **write** filtered character **streams**. It has one instance variable `out`, which is a protected reference to the **Writer** that is being filtered.

```
protected FilterWriter (Writer out)
BufferedWriter
```

This class buffers data to the character output stream. Functionality of this class is the same as that of `BufferedOutputStream` class. The constructor is:

```
public BufferedWriter (Writer w)
```

### **Program 4.24**

```
import java.io.*;
public class ReadWrite
{
 public static void main (String args [])
 {
 try
 {
 BufferedReader in=new BufferedReader (new
 FileReader (args[0]));
```

## **NOTES**

## NOTES

```
String s1="";
String s2="Learn Java";
 while((s1=in.readLine())!=null)
System.out.println(s1);
StringReader in2=new StringReader(s2);
int c;
System.out.println("Printing individual characters of the
File"+args[0]);
while((c=in2.read())!=-1)
System.out.print((char)c);
BufferedReader ind=new BufferedReader(new
StringReader(s2));
PrintWriter p=new PrintWriter(new BufferedWriter(new
FileWriter("demo.txt")));
while((s1=ind.readLine())!=null)
p.println("output "+s1);
in.close();
in2.close();
ind.close();
p.close();
}
catch(Exception e)
{
 e.printStackTrace();
}
}
```

Run the program by C:\>java ReadWrite a.txt

### **Output of the program:**

```
Java Is A Robust Language
Printing individual characters of the File a.txt
L
e
a
r
n

J
a
v
a
```

### **21.19.3 Use of FileReader and FileWriter Classes**

FileReader and FileWriter classes perform the read and write operation by characters.

A program to write something into a **file** is given below:

### Program 4.25

```
import java.io.*;
class Demo
{
 public static void main (String [] args) throws IOException
 {
 String data="On this planet\nLife is like ships in the
 harbour!!! ";
 FileWriter fw1=new FileWriter ("myFile.txt");
 for (int i=0;i<data.length();i++)
 fw1.write (data.charAt (i));
 fw1.close ();
 }
}
```

Here the output is not visible to the user because, we are not printing anything on the console. Here **fw1** is an object of `FileWriter` class which points to `myFile.txt`. **data** is an object of string, which is to be written into the **file** `myFile.txt`. `data.length()` is used to calculate the length of string. The `write()` method is invoked through **fw1** to write the data in `myFile.txt`.

### Reading from a File by `FileReader`

This can be clarified with the following example:

### Program 4.26

```
import java.io.*;
class Demo
{
 public static void main (String [] args) throws IOException
 {
 try
 {
 FileReader fr1=new FileReader ("myFile.txt");
 }
 catch (FileNotFoundException e)
 {
 System.out.println ("file does not exist");
 return;
 }
 int ch;
 while ((ch=fr1.read())!=-1)
 System.out.println ((char)ch);
 fr1.close ();
 }
}
```

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

## NOTES

## NOTES

}

Here **fr1** is an object of `FileReader` class which points to `myFile.txt`. The `read()` method is invoked through **fr1** to read the data from `myFile.txt`.

### File Encryption

The following program shows how to encrypt a file.

#### Program 4.27

```
import java.io.*;
public class Encrypt
{
 public static void main(String args[])
 {
 try{
 FileInputStream fis=new FileInputStream("dd.txt");
 int i=fis.available();
 int mak[]=new int[i];
 int k=0, ch;
 while((ch=fis.read())!=-1)
 {
 mak[k]=ch;
 mak[k]=mak[k]+2;
 k++;
 }
 char c[]=new char[i];
 for(int j=0;j<c.length;j++)
 {
 c[j]=(char)mak[j];
 }
 String s1=new String(c);
 FileOutputStream fos=new FileOutputStream("rr.txt");
 byte bb[]=s1.getBytes();
 fos.write(bb);
 fos.close();
 }catch(FileNotFoundException fe)
 {
 fe.printStackTrace();
 }
 catch(IOException ie)
 {
 }
 }
}
```



In the above program we pass the file name in the `FileInputStream` constructor which needs to be encrypted. The `available()` method of `FileInputStream` class returns the number of bytes that can be read from the `FileInputStream` and store it in the `int` variable because the `available` method return type is `int` type. Then we store it in an array of `int` type. We write `while((ch=fis.read())!=-1)` that means the `read()` method reads the data from the file till the end and store it in an `int` variable `ch`. And then we store the `ch` in an array of `int` type. After that we write `mak[k]=mak[k]+2;` that means we increase their ascii value. In this line we encrypt the data. After that we store it in an array of character by typecasting. Then we store the array of character in the `String` constructor. The encrypted data should be written in a file by the `FileOutputStream` class. So we pass the file name in the `FileOutputStream` constructor. Through the `getBytes()` method we convert it into the `byte` and through `write` method we write it on the file. Because `FileOutputStream` writes in `byte` format.

## NOTES

## 4.4 BASIC OF NETWORKING

The network systems comprises a server, client and communication media (Refer Figure 4.5). A machine running a process that sends request for the services is known as client. On the other hand, a machine running a process that responds to the client's request by offering requested services is known as server. A server can handle many clients at the same time. To make the clients and the server communicate, a connecting medium is required. The communication medium may be wired or wireless network.

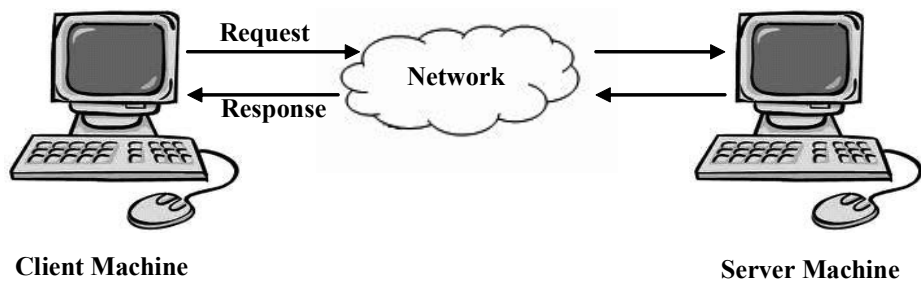


Fig. 4.5 Client-Server Communication

A network can be small (having two computers) or as vast as the Internet. Communication over network requires some reliable delivery services that can carry information between two machines. The delivery services must work regardless of the hardware and software used on the network. The Internet provides delivery services using a suite of protocols known as TCP/IP named after its two primary protocols, namely, Transport Control Protocol and Internet Protocol. The TCP/IP consists of four layers, namely Application, Transport, Network and Link layer. All the layers along with their corresponding protocols are shown in Figure 4.2.

## NOTES

|                          |                          |
|--------------------------|--------------------------|
| (HTTP, TELNET, FTP, ...) | <b>Application Layer</b> |
| (TCP, UDP, ...)          | <b>Transport Layer</b>   |
| (IP, ...)                | <b>Network Layer</b>     |
| (Device Driver, ...)     | <b>Link Layer</b>        |

*Fig. 4.6 TCP/IP Model*

### TCP/IP Protocols

The three most commonly used protocols within the TCP/IP suite are IP, TCP and UDP. To develop network application, one must have a clear understanding of these protocols.

#### TCP (Transmission Control Protocol)

TCP is a trustworthy and connection oriented protocol that permits the data that originates from source machine to be delivered without error to the destination. TCP sets up a connection between the source machine and the destination machine by transmitting control information before initiating the communication. This mechanism is known as handshake. Once the connection is established, data transfer between the two machines begins. TCP fragments the data into discrete messages (known as TCP segments) and passes them to the Internet layer. At the destination, the receiving TCP sends an acknowledgement that guarantees that the data has been received. It then reassembles the segments to form the original message. In case the segments are lost or corrupted, TCP is responsible for retransmitting the necessary segments. When all the data has been exchanged between these machines, it closes the connection.

#### IP (Internet Protocol)

IP (Internet Protocol) is an unreliable and connectionless protocol that manages the address part of each packet (the basic unit of IP transmissions) so that the packet reaches the right destination. Being a connectionless protocol, IP does not transmit control information before initiating communication between the source and the destination. It sends data from the source to the destination expecting that the data will be delivered at the receiving end properly. It is unreliable as it does not retransmit lost or corrupted packets, that is, it does not guarantee the safe delivery of packets.

#### UDP (User Datagram Protocol)

Unlike TCP based applications, some network applications do not require a host-to-host and reliable channel of communication. Instead they require a mode of communication which transmits independent, self contained messages whose time of arrival, order of arrival, content, etc. are not guaranteed. This mode of network communication is governed by the UDP. The UDP (User Datagram Protocol) is considered as an untrustworthy and connectionless protocol which enables application to send independent, self contained messages known as datagrams

over the network. It has no mechanism for detecting errors or retransmitting lost or corrupted information. It is used in the applications that require prompt delivery instead of accurate delivery, such as weather forecasting, clock server, video, games and audio.

## IP Address

Every machine on the Internet is identified by a numerical address known as IP address. An IP address is made up of a 32-bit number, organized as four 8-bit values. It is represented in a format known as dotted decimal notation. In this representation, each group of bit is represented by its decimal equivalent which is between 0 and 255. For example, 1.160.10.240 is an IP address. This address type was specified by IPv4 (Internet Protocol, version 4).

Since this representation is not user friendly, IP address is mapped to domain name like `www.google.com` which is easier to remember. Special servers on the Internet perform this mapping called domain name servers. It allows users to work with domain names; however, the Internet operates on the IP addresses.

*Note:* New addressing scheme (known as IPv6) uses 128-bit value to represent an address.

## URLs

Each Webpage has a typical and unique address termed as a URL or Uniform Resource Locator which is used to identify its location on the Internet. The URL consists of four parts: protocol, name of the Web server, the directory on that server and the file within that directory.

The syntax of URL is as follows:

```
protocol://domain name: port number/<directory path>/<object name>#spot
```

For example, `http://www.yahoo.com/education/FIIT/home.htm#top` is a URL.

The various parts of a URL are described in Table 4.6.

*Table 4.6 Parts of a URL*

| Part           | Example                      | Description                                                                                                    |
|----------------|------------------------------|----------------------------------------------------------------------------------------------------------------|
| Protocol       | <code>http://</code>         | It represents the name of the protocol like ftp, http, etc.                                                    |
| Domain Name    | <code>www.yahoo.com</code>   | It represents the name of the Web server where the desired Web page or other resource resides.                 |
| Directory Path | <code>/education/FIIT</code> | It represents the location of the Web page in the Web server's file system.                                    |
| Object Name    | <code>/home.htm</code>       | It specifies the name of the file for the desired Web page or the name of the other resource that is required. |
| #Spot          | <code>#top</code>            | It specifies the particular location of the text or the graphic on the Web page.                               |

## NOTES

## NOTES

### Types of URL

There are two types of URL, namely absolute URL and relative URL.

- (i) **Absolute URL:** An absolute URL defines the exact location of the Web page or any other resource on the Internet. For example, *http://www.yahoo.com/education/FIIT/home.htm*. Here, the Web page *home.htm* is stored under the folders */education/FIIT* on the Web server *www.yahoo.com*.
- (ii) **Relative URL:** In a relative URL, each part of a URL does not need to be specified. You can abbreviate a URL by making it 'relative' to the current location. For example, suppose there exists a file *mypicture.gif* residing in the image folder under the *FIIT* folder. The relative URL for the file will be */images/mypicture.gif*.

### Ports and Port Numbers

A machine provides a variety of services including e-mail, Telnet, FTP, etc. Using IP address, a client can connect to the machine but cannot connect to the desired service. To resolve this, with each service, a port number is associated. A port is a logical number assigned to a particular service through which the service is requested.

Clearly, to avail the a service of a server, a client machine does not just connect to the server, it connects to a port on that server. Each packet that is sent over a network contains the IP address of the host machine and the port number to identify the particular application running on that host machine. An IP address can be considered as the house address where a letter is sent via post and port number is the name of the person to whom letter is to be delivered.

*Note:* The port numbers below 1024 are known as well known ports and are reserved for standard services. For example, the port number used for Telnet is 23.

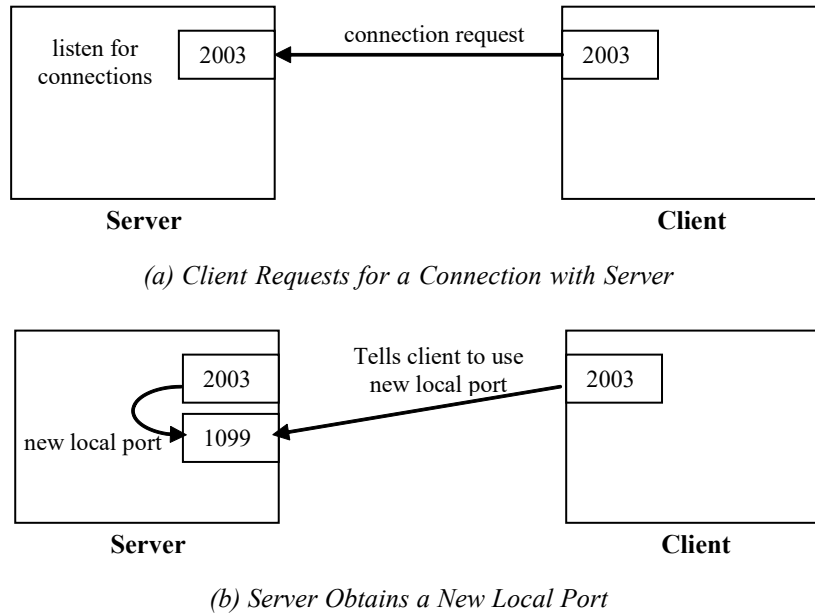
### Sockets

The grouping of IP address and port number is acknowledged as socket. A socket identifies an endpoint of a two way communication link between two programs running on the network. When a client requests for a connection on a particular port, the server identifies and keeps track of the socket that it will use to communicate with that client. A server can communicate on the same port with many clients using sockets to determine the destination and source of the communication.

### Socket Based Communication

In socket based communication, the server (program) running on a computer binds a socket to a specific port. The server listens to this socket for any client's connection request as shown in Figure 4.7(a). When a request is made, the server accepts the request. Once the request is accepted, the server binds a new socket to a different port (see Figure 4.7(b)). The new socket is required so that the server can continue to listen to the original socket for new connection request and at the same time keep serving the connected clients.

## NOTES



**Fig. 4.7** Establishment of Path for Two-Way Communication

### 4.4.1 Proxy Server

This is the first layer of the RMI architecture and acts as an interface between the application layer and other RMI layers. The method calls initiated at the client end to process the service are intercepted by the proxy layer. The proxy layer performs the mapping of interface variables initialised by the client application program with the variables at the reference layer.

#### Remote Reference Layer

This layer acts as an interface between the proxy layer and the transport layer. The proxy layer receives request for service from the client proxy layer and manages the semantics of the call to be send to the server using the transport layer. The proxy layer manages the semantics of the request based on remote reference protocols. Similarly, the proxy layer manages the semantics of the service processed by the server, prior to a delivery of the response to the client side.

### 4.4.2 Domain Naming Services

It is very difficult to remember the IP address to connect to the Internet. The **Domain Name System** (DNS) is used to overcome this problem. DNS maps one particular IP address to a string of characters, which is popularly known as **domain** name. For example, www.yahoo.com implies com is the **domain** name reserved for US commercial sites, yahoo is the name of the company and www specifies that the site is available in the World Wide Web. This name is visible to the user. However, in the background, the name maps to a particular IP address.

## NOTES

### 4.4.3 Networking Classes and Interfaces

Java facilitates creation of network applications through the classes and interfaces defined in the `java.net` package. Some of the classes defined in this package are listed in Table 4.7.

*Table 4.7 Some of the Classes of the java.net Package*

| Class              | Description                                                                                                                       |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Authenticator      | It represents an object that obtains authentication for a network connection.                                                     |
| ContentHandler     | It is an abstract class—superclass of all the classes that contains methods to read an object from a <code>URLConnection</code> . |
| DatagramPacket     | It contains methods to implement a connectionless packet delivery service.                                                        |
| DatagramSocket     | It represents a socket for sending and receiving datagram packets.                                                                |
| DatagramSocketImpl | It is an abstract datagram and multicast socket implementation superclass.                                                        |
| HttpURLConnection  | It represents a <code>URLConnection</code> having support for HTTP-specific features.                                             |
| InetAddress        | It represents an IP address.                                                                                                      |
| ServerSocket       | It implements server sockets.                                                                                                     |
| Socket             | It implements client sockets.                                                                                                     |
| SocketImpl         | It is an abstract class—superclass of all classes that implements sockets.                                                        |
| URL                | It represents a Uniform Resource Locator, a pointer to a resource on the WWW.                                                     |
| URLConnection      | It is an abstract class—superclass of all classes that represent a communication link between the application and a URL.          |
| URLConnectionImpl  | It is an abstract class—superclass of all stream protocol handlers.                                                               |

Some of the interfaces defined in `java.net` package are listed in Table 4.8.

*Table 4.8 Interfaces of java.net Package*

| Interface                 | Description                                                                                     |
|---------------------------|-------------------------------------------------------------------------------------------------|
| ContentHandlerFactory     | It defines factory for content handlers.                                                        |
| CookiePolicy              | It provides a mechanism to decide which cookie should be accepted and which should be rejected. |
| CookieStore               | It represents a storage for cookie.                                                             |
| DatagramSocketImplFactory | It defines factory for the implementation of datagram socket.                                   |
| FileNameMap               | It provides a mechanism to map between a file name and a MIME type string.                      |
| SocketImplFactory         | It defines factory for the implementation of socket.                                            |
| SocketOptions             | It defines methods to get or set socket options.                                                |
| URLConnectionImplFactory  | It defines factory for URL stream protocol handlers.                                            |

**Note:** Factory methods are simply a convention by which the static methods in a class return an instance of that class.

### 4.4.4 InetAddress Class

As stated earlier, users use the domain names, whereas the Internet operates on IP addresses. In Java, the domain names can be resolved to their IP addresses and vice versa by using the `InetAddress` class.

An `InetAddress` object can be created using one of the available factory methods. Commonly used factory methods of `InetAddress` class are listed in Table 4.8.

**Table 4.9** Methods of the *InetAddress* Class

| Method                                                                          | Description                                                                                                                                                                         |
|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static <code>InetAddress</code><br><code>getLocalHost()</code>                  | It returns the <code>InetAddress</code> object that represents the local host.                                                                                                      |
| static <code>InetAddress</code><br><code>getByName(String hostname)</code>      | It returns an <code>InetAddress</code> for a host name passed to it.                                                                                                                |
| static <code>InetAddress[]</code><br><code>getAllByName(String hostname)</code> | It returns an array of <code>InetAddress</code> that represents all of the addresses resolved for a host name passed to it; used in case a single name represents several machines. |

## NOTES

All these methods throw an `UnknownHostException`. The methods `getLocalHost()` and `getByName(String hostname)` throw this exception if they are unable to resolve the host name. The method `getAllByName()` throws this exception if it is unable to resolve the name to even a single address.

### 4.4.5 Datagram Packet Network

Java supports UDP by providing two classes, namely `DatagramPacket` and `DatagramSocket`.

#### **DatagramSocket**

The `DatagramSocket` class is used to send or receive the datagram packets. It defines four constructors that are listed in Table 4.10.

**Table 4.10** Constructors Defined in *DatagramSocket* Class

| Constructor                                                        | Description                                                                                                                                                              |
|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DatagramSocket()</code>                                      | It creates a datagram socket.                                                                                                                                            |
| <code>DatagramSocket(int portnumber)</code>                        | It creates a datagram socket and binds it to the port number specified by <code>portnumber</code> .                                                                      |
| <code>DatagramSocket(int portnumber, InetAddress ipAddress)</code> | It creates a datagram socket and binds it to the port and the <code>InetAddress</code> specified by <code>portnumber</code> and the <code>ipAddress</code> respectively. |
| <code>DatagramSocket(SocketAddress address)</code>                 | It creates a datagram socket and binds it to the <code>SocketAddress</code> specified by the address.                                                                    |

**Note:** While creating the datagram socket, if any error occurs, all these constructors throw a `SocketException`.

The class `DatagramSocket` defines many methods. Two important methods are listed in Table 4.11.

**Table 4.11** Methods of the *DatagramSocket* Class

| Method                                           | Description                                                        |
|--------------------------------------------------|--------------------------------------------------------------------|
| <code>void send(DatagramPacket packet)</code>    | It sends packet to the port.                                       |
| <code>void receive(DatagramPacket packet)</code> | It receives packet specified by <code>packet</code> from the port. |

**Note:** Both the methods `send()` and `receive()` throw a `IOException` when an error occurs.

## DatagramPacket

The DatagramPacket class is used to contain the data. It defines many constructors; some of them are listed in Table 4.12.

### NOTES

**Table 4.12** Constructors Defined in DatagramPacket Class

| Constructor                                                                                 | Description                                                                                                          |
|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| DatagramPacket(byte data[], int length)                                                     | It creates a DatagramPacket for receiving the packets of length length in the buffer.                                |
| DatagramPacket( byte data[],int offset, int length )                                        | It creates a DatagramPacket for receiving the packets of length length, specifying an offset into the buffer.        |
| DatagramPacket( byte data[], int offset, int length, InetAddress ipAddress, int portnumber) | It creates a DatagramPacket for sending packets of length length to the specified port number on the specified host. |

**Table 4.13** Methods of the DatagramPacket Class

| Method                                 | Description                                                                                                     |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| InetAddress getAddress()               | It returns the address of the machine to which the datagram is sent or from which the datagram is received.     |
| byte[] getData()                       | It returns the data contained in the datagram.                                                                  |
| int getLength()                        | It returns the length of the valid data contained in the byte array that is returned from the getData() method. |
| int getPort()                          | It returns the port number on the remote host.                                                                  |
| void setAddress(InetAddress ipAddress) | It sets the address specified by ipAddress to which a packet is to be sent.                                     |
| void setData(byte[] data)              | It sets the data to data.                                                                                       |
| void setLength(int length)             | It sets the size of the packet to length.                                                                       |
| void setPort(int portnumber)           | It sets the port number specified by portnumber on the remote host.                                             |

**Example 4.1:** A program to demonstrate a UDP server program is as follows:

```
import java.net.*;
import java.io.*;
import java.util.*;
class UDPServerExample
{
 public static void main (String args []) throws Exception
 {
 int serverportno=1139;
 int clientportno=1140;
 try
 {
 //creates a socket and binds to port no 1139
 DatagramSocket ds=new
 DatagramSocket (serverportno);
 //obtains the IP address for the localhost
 InetAddress ipadd=InetAddress.getLocalHost();
 String pstr="Hello this is your message";
 byte buf[]=pstr.getBytes();
 // creates a DatagramPacket for sending the packets
 DatagramPacket send_packet=new
 DatagramPacket (buf,buf.length, ipadd, clientportno);
 ds.send(send_packet); //send packets
 }
 }
}
```



```
 ds.close();
 }
 catch (Exception e)
 {
 System.out.println(e.getMessage());
 }
}
}
```

**Example 4.2:** A program to demonstrate a UDP client program is as follows:

```
import java.net.*;
import java.io.*;
class UDPClientExample
{
 public static void main (String args []) throws Exception
 {
 int serverportno=1139;
 int clientportno=1140;
 try
 {
 //creates a socket and binds to port no 1140
 DatagramSocket ds=new DatagramSocket (clientportno);
 byte b []=new byte [1024];
 //creates a DatagramPacket for receiving the packets
 DatagramPacket dp=new DatagramPacket (b,b.length);
 ds.receive (dp); //receives packets
 System.out.println (new String (dp.getData ()));
 }
 catch (Exception e)
 {
 System.out.println (e.getMessage ());
 }
 }
}
```

Note that both the server and the client programs are running between two ports on the local machine. The server and the client are running on port numbers, 1139 and 1140 respectively. So, run these programs on different command prompt windows.

### **Check Your Progress**

1. Write the use of file class.
2. Define the term directory.
3. What do you mean by stream?
4. Write the types of streams.
5. Which suit of protocols does the Internet provide for delivery services?
6. What is the combination of IP address and port number known as?
7. Define the term domain name system.
8. For what purpose is the `InetAddress` class used?
9. If an error occurs while creating the datagram socket, which type of exception is thrown?

### **NOTES**

## NOTES

---

## 4.5 APPLET BASIC

---

**Applets** are executed in the web browser and have made Java a web-enabled language. **Applet codes** also embed the HTML tags together. This is the package that has made Java a language for distributed application.

Generally, Java programs can be classified into two groups, namely applications and applets. Unlike applets, Java applications do not require a browser to run. They can be created like any normal programming language program. An applet is executed in the browser. An applet is a class file that displays graphics application in the web browser and one can embed applet codes in the web pages by HTML tags. Briefly, it can be said that an applet is a Java byte code embedded in an HTML page. The program structure of applets differs from the other Java applications.

### Definition

An **applet** is a dynamic and interactive program that can run inside a web page displayed by a Java-capable browser, such as a HotJava Browser or an Internet Explorer browser, which are world wide web browsers used to view web pages. An applet is a class present in a java.applet package.

A special HTML tag is embedded in an applet to make it run on the web browser. The **appletviewer** application, present in the **jdk**, is used to run and check the applets. An applet has added advantages, such as frame, event-handling facility, graphics context and surrounding user interfaces.

Java applets have some restrictions to ensure full security and to make them virus free. Some of these are as follows:

- (a) Applets have no permission to read or write the file system.
- (b) Applets can communicate with the server in which they were stored originally, but not with the others.
- (c) Applets cannot execute any programs on the system.

### 4.5.1 Applet Life Cycle

Each applet class inherits the properties and methods of the class, Applet. An applet is loaded on the web browser or the appletviewer tool of Java. An Applet may change its current state when a specific method is called by AWT. Java provides four methods to change the state of an applet. These methods are as follows:

- `init()`
- `start()`
- `stop()`
- `destroy()`

The `init()` method is called when an applet loads the initialization process. The initialization process creates the objects that an applet needs for loading images, fonts and colours or for setting up the initial parameters such as variables and constants. This method is called only once when an applet is loaded the first time. The syntax to initialize the applet is:

```
public void init()
{
 // init() method definition.
}
```

The `start()` method is called when an applet is initialized and starts the execution of the applet. The syntax to start the applet is:

```
public void start()
{
 // start() method definition.
}
```

The `paint()` method is called when you want to display an applet. The syntax to display the applet is:

```
public void paint(Graphics g)
{
 g.drawString(str, 10, 10);
}
```

The `stop()` method is called when either the end user stops an applet or an applet loses the focus. The syntax to stop the applet is:

```
public void stop()
{
 // stop() method definition.
}
```

The `destroy()` method is called when an applet is destroyed. When you want to exit from the web browser or appletviewer tool of Java, an applet calls this method to free the resources. This method also occurs only once in the life cycle of an applet. The syntax to destroy the applet is:

```
public void destroy()
{
 // destroy() method definition.
}
```

Here, only the `paint()` method is a member of the graphics class, while the other methods are members of the applet class. Figure 4.8 shows the life cycle of an applet.

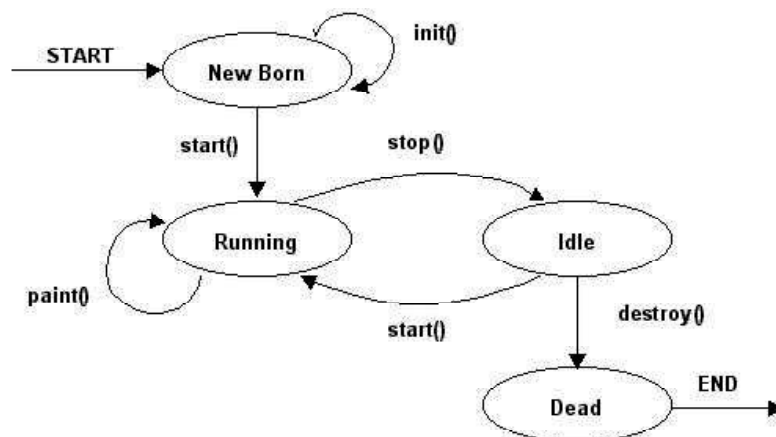


Fig. 4.8 Life Cycle of an Applet

There are four states in the life cycle of an applet: new born, running, idle and dead. The newborn state consists of a newly loaded applet that is initializing its

## NOTES

## NOTES

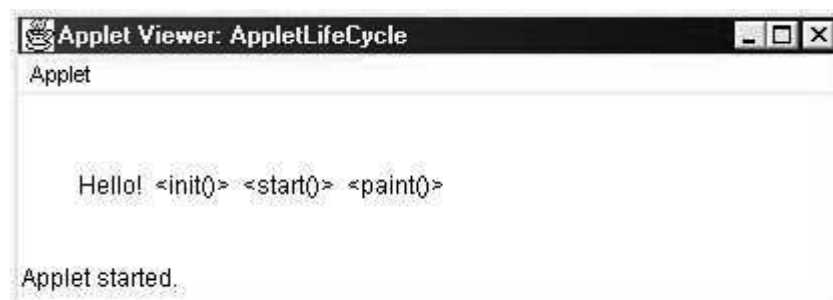
resources. In the running state, an applet is executed and it displays the text or image that the applet contains. An applet that is initialized but is not currently in the running state is said to be in the idle state. When the `destroy()` method is invoked, the applet acquires a dead state and releases all the resources. The following code creates an applet that shows the various states of an applet life cycle:

### Creating an Applet showing the various States of an Applet Life cycle

```
import java.applet.Applet;
import java.awt.Graphics;
public class AppletLifeCycle extends Applet
{
 String str="Hello! ";
 public void init()
 {
 str=str+"<init()> ";
 }
 public void start()
 {
 str=str+"<start()> ";
 }
 public void stop()
 {
 str=str+"<stop()> ";
 }
 public void destroy()
 {
 str=str+"<destroy()> ";
 }
 public void paint(Graphics g)
 {
 str=str+"<paint()> ";
 g.drawString(str, 30, 50);
 }
 /*
 <APPLET CODE="AppletLifeCycle" HEIGHT=80 WIDTH=410>
 </APPLET>
 */
}
```

The above code shows the various states of the applet life cycle using applet methods. The `init()` method initiates all the variables that are used in the applet program. The `paint()` method displays text on the applet.

The following screenshot shows the output.



You can stop the execution of the applet by calling the `stop()` method.

The following screenshot shows the suspended applet.



AWT calls the `start()` and `paint()` methods when you re-start the applet.

The following screenshot shows re-starting the applet that was suspended by the `stop()` method.



There is one additional method, `paint`, that is not a part of the applet life cycle but is used for displaying the contents of the applets on the appletviewer or web browser. The `paint()` method is called to display text or graphics on an applet. This method takes an argument, which is the instance of the `Graphics` class. The syntax to display the text on an applet is:

```
public void paint(Graphics g)
{
 //paint() method definitions.
}
```

The above syntax shows the `paint()` method of the `graphics` class of Java.

### 4.5.2 Simple Banner Applet

The body section of the HTML file contains a pair of `<APPLET...>` and `</APPLET>` tags, which allows one to provide the name of the applet and helps the browser recognize the space required for the applet. The following code shows the minimum code required to place the `FirstApplet` applet on the web page:

```
<APPLET
 CODE=FirstApplet.class
 WIDTH=300
 HEIGHT=150 >
</APPLET>
```

The above written HTML code helps a web browser load the compiled Java applet `FirstApplet.class`, which is present in the same directory as the HTML file. The `<APPLET>` tag discussed above includes the following:

## NOTES

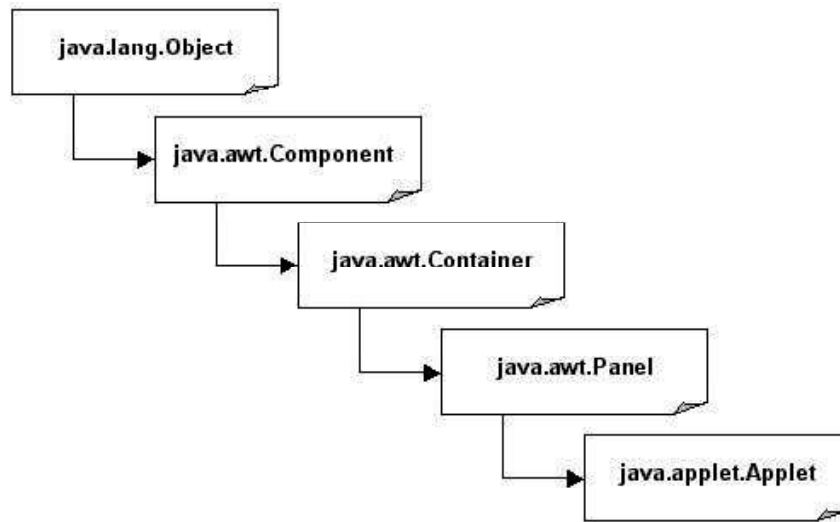
- Name of the Applet
- Width of the Applet (in pixels)
- Height of the Applet (in pixels)

## NOTES

### The Applet Class

The applet class is a member of the Java Application Programming Interface (API) package, `java.applet`. The applet class is used for creating a Java program that displays an applet.

Figure 4.9 shows the hierarchical representation of the Java classes.



*Fig. 4.9 Java Class Hierarchies*

The applet class is extended from the class, Panel, which is further extended from the classes, Container, Component and Object. The object class is a member of the `java.lang` package and is called in the program automatically. The classes, Component, Container and Panel are members of the `java.awt` package and provide the visual components such as label, button or text fields. The applet is the only member of the `java.applet` package.

The applet class has several methods that are used to display the text and the image, play the audio file and respond when you interact with the applet. Table 5.1 lists the various methods of an applet class.

*Table 4.14 Applet Class Methods*

| Methods                             | Description                                                                                                                    |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| <code>void destroy()</code>         | Terminates an applet when the web browser or Java tool calls this method.                                                      |
| <code>getAccessibleContext()</code> | Returns the accessibility context of an object. Accessibility context represents the information about the accessible objects. |
| <code>getAppletContext()</code>     | Returns the context that is associated with the applet.                                                                        |

|                                                      |                                                                                                                                                |
|------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>GetAppletInfo()</code>                         | Returns a string type that describes the information about the applet.                                                                         |
| <code>GetAudioClip(URL url, String clip-name)</code> | Returns an object of the audio clip that encapsulates the location and name of the audio clip.                                                 |
| <code>getCodeBase()</code>                           | Returns the URL that is associated with the invoking applet.                                                                                   |
| <code>getDocumentBase()</code>                       | Returns the URL of the HTML document that is invoking the applet.                                                                              |
| <code>getImage(URL url, String image-name)</code>    | Returns the image object that encapsulates the location and name of the image.                                                                 |
| <code>getLocale()</code>                             | Returns the locale object that contains a set of end user preferences such as language, country, region or time.                               |
| <code>getParameter(String param-name)</code>         | Returns a string that contains the parameter associated with the paramname.                                                                    |
| <code>getParameterInfo()</code>                      | Returns a table that describes the information about the parameters that are recognized by the applet.                                         |
| <code>void init()</code>                             | Begins the execution of the applet when it is called by the web browser or Java tool.                                                          |
| <code>IsActive()</code>                              | Returns the Boolean type true if the applet is started, otherwise returns false.                                                               |
| <code>void play(URL url, String clip-name)</code>    | Plays the audio clip if it is found at the specified URL.                                                                                      |
| <code>void resize(int width, int height)</code>      | Changes the size of an applet according to the specified height and width.                                                                     |
| <code>void setStub(AppletStub stub-object)</code>    | Returns the stub object of an applet. A stub is a piece of program that provides the linkage between the applet and the web browser.           |
| <code>ShowStatus(String str)</code>                  | Displays a string in the status window of the appletviewer or web browser.                                                                     |
| <code>void start()</code>                            | Starts the execution of the applet when it is called by the web browser or Java tool.                                                          |
| <code>void stop()</code>                             | Suspends the execution of the applet when it is called by the web browser or Java tool. This suspended stage is resumed by the start() method. |

---

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

## NOTES

## NOTES

Applets are programs based on the Internet that can be used to display text, image or animation. For example, an applet can be created that will display a text. To display text on the applet, the following four steps are to be followed:

1. Create a Java program for an applet
2. Compile the Java program
3. Create a web page that contains an applet
4. Run the applet

### Adding an Applet to an HTML File

To run the applet, the HTML file that embeds an applet on the web page has to be created using the <APPLET> tag. This file has to be saved as FirstApplet.html. The following program code creates an HTML web page that embeds an applet:

```
<HTML>
 <HEAD>
 <TITLE>First Applet Program</TITLE>
 </HEAD>
 <BODY>
 <APPLET CODE="FirstApplet.class" HEIGHT=150 WIDTH=300>
 </APPLET>
 </BODY>
</HTML>
```

The above code creates an HTML file in which the <APPLET> tag is used to embed the applet. The attribute of the <APPLET> tag, HEIGHT and WIDTH, sets the dimension of the applet window.

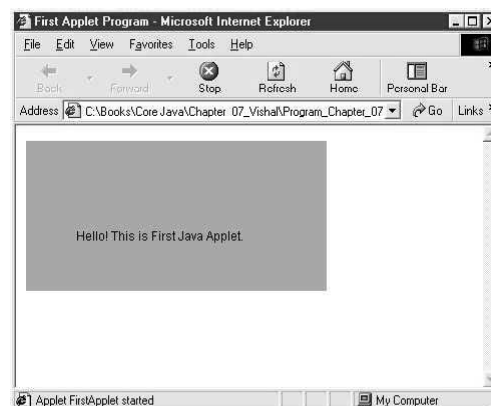
**Note:** You may specify the <APPLET> tag as a comment inside the Java applet source file. The code, which is specified inside the Java applet source file, is:

```
/*
 <APPLET CODE="FirstApplet" HEIGHT=150 WIDTH=300>
</APPLET>
*/
```

### Running the Applet

You can run the applet either on the web browser or the appletviewer of Java. When you run the applet on a web browser, you invoke the HTML file from the web browser.

The following screenshot displays the applet in Internet Explorer.





You can display the applet in appletviewer either by running the Java file, which consists of the <APPLET> tag as comment or by running the separate HTML file. The code to display the applet in the appletviewer of Java is:

```
C:\>java>Unit07>appletviewerFirstApplet.html
Or
C:\>java>Unit07>appletviewerFirstApplet.java
```

The above code shows an applet in the appletviewer of Java.

The following screenshot shows the output.



### More about the Applet Tag

The <APPLET> tag is used to display an applet in the web browser or appletviewer tool of Java. You can also use the <OBJECT> tag of HTML to display the applet. The appletviewer executes the <APPLET> tag and displays an applet in an applet window. A web browser, such as Internet Explorer or Netscape Navigator, displays the applet on a web page interpreting the <APPLET> tag. The <APPLET> tag has various attributes that enable you to integrate your applet into the web page. The syntax of <APPLET> with all its attributes is:

```
<APPLET [CODEBASE] [CODE] [ALT] [NAME] [WIDTH] [HEIGHT] [ALIGN] [VSPACE]
[HSPACE]>
</APPLET>
```

The above syntax shows the <APPLET> tag and its attributes. The <APPLET> tag provides nine attributes, which are as follows:

- **CODEBASE:** Specifies the base URL of the applet class file. The base URL is the complete path of the applet where it is stored. This is an optional attribute that searches the executable file of an applet, class file, from the specified URL.
- **CODE:** Specifies the name of the file that contains a class file. This is a compulsory attribute of the <APPLET> tag that is relative to the URL of the HTML file.
- **ALT:** Specifies the tool tip text of an applet. This is also an optional attribute that is displayed to provide information about the applet whether the applet is running or not on the web browser.
- **NAME:** Specifies the name of an applet. This is an optional attribute of the <APPLET> tag. The name of an applet is obtained by the getApplet() method.
- **WIDTH:** Specifies the width of the applet display area in pixels.

### NOTES

## NOTES

- **HEIGHT:** Specifies the height of the applet display area in pixels.
- **ALIGN:** Specifies the alignment of the applet. The values of the ALIGN attribute are LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE and ABSBOTTOM.
- **VSPACE:** Sets the vertical space of the applet, in pixels, on each side of the applet.
- **HSPACE:** Sets the horizontal space of the applet, in pixels, on each side of the applet.

### Passing Parameters to Applets

The <APPLET> tag enables you to pass the applet parameter in the applet using the <PARAM> tag. The <PARAM> tag has two attributes, NAME and VALUE, which are used to set the name and values of the parameter. The syntax of the <PARAM> tag that is enclosed within the <APPLET> tag is:

```
<APPLET>
<PARAM [NAME] [VALUE]>
</APPLET>
```

The above syntax shows the <APPLET> tag with parameter values. The NAME represents the name of the parameter and VALUE sets a value for that parameter. The <PARAM> tag is an empty tag. You can put several <PARAM> tags inside a single <APPLET> tag. You can retrieve the value of a parameter using the getParameter() method that returns the specified parameter as a String object. The following program code shows an applet displaying the information of an end user that is passed as the parameter of the <PARAM> tag:

### Displaying User Information by passing Parameters

```
import java.awt.*;
import java.applet.Applet;
public class AppletHTML extends Applet
{
 String firstName, lastName, sex, add, city, phone, email;
 Font font1, font2;
 public void init ()
 {
 font1=newFont ("Arial", Font.PLAIN, 16);
 font2=newFont ("Arial", Font.BOLD, 20);
 }
 public void start ()
 {
 //Stringparam;
 firstName=getParameter ("firstName");
 if (firstName==null)
 firstName="Not Found";
 lastName=getParameter ("lastName");
 if (lastName==null)
 lastName="Not Found";
 sex=getParameter ("sex");
 if (sex==null)
 sex="Not Found";
 add=getParameter ("add");
 if (add==null)
 add="Not Found";
 }
}
```

## NOTES

```
 city=getParameter("city");
 if(city==null)
 city="Not Found";
 phone=getParameter("phone");
 if(phone==null)
 phone="Not Found";
 email=getParameter("email");
 if(email==null)
 email="Not Found";
 }
 publicvoidpaint(Graphicsg)
 {
 g.setFont(font2);
 g.drawString("-: User Information :-", 75,
0);
 g.setFont(font1);
 g.drawString("First Name : "+firstName, 5,
0);
 g.drawString("Last Name : "+lastName, 5, 80);
 g.drawString("Sex : "+sex, 5, 100);
 g.drawString("Address : "+add, 5, 120);
 g.drawString("City : "+city, 5, 140);
 g.drawString("Phone No : "+phone, 5, 160);
 g.drawString("Email : "+email, 5, 180);
 }
 /*
 <APPLET CODE="AppletHTML" WIDTH=375 HEIGHT=200>
 <PARAMNAME=firstName VALUE="Vishal">
 <PARAMNAME=lastName VALUE="Jayaswal">
 <PARAMNAME=sex VALUE="Male">
 <PARAMNAME=add VALUE="779-A,Civil-Lines">
 <PARAMNAME=city VALUE="Jhansi">
 <PARAMNAME=phone VALUE="9891066098">
 <PARAMNAME=email VALUE="vishal1431@rediffmail.com">
 </APPLET>
 */
 }
```

The above code shows the user information that is passed as a parameter of the `<PARAM>` tag. The `getParameter()` method retrieves the value of each parameter and displays this value on the applet.

The following screenshot shows the output of the user information.



## NOTES

You can create an interactive or dynamic applet by passing the values of the parameter that are specified within the <PARAM> tag of the <HTML> document. When you are working with the <PARAM> tag, you can change the contents that are displayed on the applet by changing the parameter values in the HTML file.

### Different Methods used to Set the Graphical Environment

An applet uses the classes and methods of AWT to perform its input and output operations. To display the output in the applet, one uses the drawstring() method present in the graphics class. Its general form is:

```
void drawString (String msg, int x_co, int y_co)
```

Here, msg holds the string to be written on the applet screen starting from the coordinates specified by int x\_co, int y\_co. In a Java window, the upper-left corner is location 0,0.

The setBackground () method is used to set the background color of the applet.

The setForeground () method is used to set the foreground color of the applet, i.e., the colour of the text to be written.

The above methods are defined in component class, and their general forms are:

(a) void setBackground (Color Color)

(b) void setForeground (Color Color)

**Color** specifies the new colour. The color class defines the following constants that can be used to specify colours:

```
Color.black
Color.magenta
Color.blue
Color.orange
Color.cyan
Color.pink
Color.darkGray
Color.red
Color.gray
Color.white
Color.green
Color.yellow
Color.lightGray
```

The following example sets the background color to pink and the text color to magenta.

#### Example 4.3

```
setBackground (Color.pink);
setForeground (Color.magenta);
```

An example for the creation of an applet is given below:

#### Example 4.4

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=750 height=500>
</applet>
*/
public class abc extends Applet {
 String msg = "WELCOME TO APPLET";
 public void init () {
 setBackground (Color.cyan);
```

```
setBackground(Color.red);
}
publicvoidstart() {
msg=msg+"MISS SAMITA";
}
publicvoidstop() {
}
publicvoidpaint(Graphicsg1) {
g1.drawString(msg, 50, 30);
}
}
```

### Output of the program:



From the above example, you can be seen that the steps to create an applet are:

First, one has to import both the packages, **java.applet** and **java.awt**. After that, the HTML tag has to be defined, which is required to make the applet run in the web browser. Then the `init()` method has to be defined. Then the `start()` method and the `paint` method have to be defined. Here, the `stop()` or `destroy()` methods are not required and therefore have not been defined.

Now from the above example, it can be seen that first the `init()` method is called and the background color and foreground color are set using the `setBackground(Color.cyan)` and `setForeground(Color.red)` methods. After that, the `start` method is called, in which the **msg** variables content is modified. Then the `paint` method is called, in which the code `g.drawString(msg, 50, 30)` has been written to print the output on the applet.

Another example would make the concept clearer and help in understanding the usage of passing parameters to the applet:



## NOTES

## NOTES

### Example 4.5

```
import java.applet.Applet;
import java.awt.*;
public class Second extends Applet
{
 Font f=new Font ("TimesRoman", Font.BOLD, 40);
 String name;
 public void init ()
 {
 name=getParameter ("name");
 if (name==null)
 {
 name="Java";
 name="Have a nice day "+name;
 }
 }
 public void paint (Graphics g1)
 {
 g1.setFont (f);
 g1.setColor (Color.blue);
 g1.drawString (name, 50, 50);
 }
}
/*<applet code="Second.class" width=200 height=200 align=TOP>
<param name="name" value="Sai">
</applet>*/
```

An instance of the font class **f** is declared. This object has been initialized to contain TimesRoman as font name, font size as 40 and font style as BOLD. The `init ()` method, declared in line 7, contains the `getParameter ()` method, which accepts the name (a string) as its parameter. The `paint ()` method of the component class is overridden to execute the `paint ()` method in the class. This method contains the `drawString ()` method, apart from the two methods, namely, `setFont ()` and `setColor ()`. These two methods are used to set the desired font and colour respectively.

### Passing Parameters to Applets

A programmer can set the parameter, as already explained. To retrieve the value, the `getParameter ()` method has to be used, which takes the name of the parameter and returns the value stored in the parameter.

### Example 4.6

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=750 height=500>
<param name="param1" value="SAI">
</applet>
*/
public class abc extends Applet {
 String msg;
 public void init () {
 setBackground (Color.white);
 setForeground (Color.red);
 }
}
```

```
}
publicvoidstart () {
mesg=getParameter ("param1");
}
publicvoidpaint (Graphicsg1) {
g1.drawString (mesg, 50, 30);
}
}
```

### Output of the program:



Applet started.

---

### Playing an Audio Clip

Consider the following example.

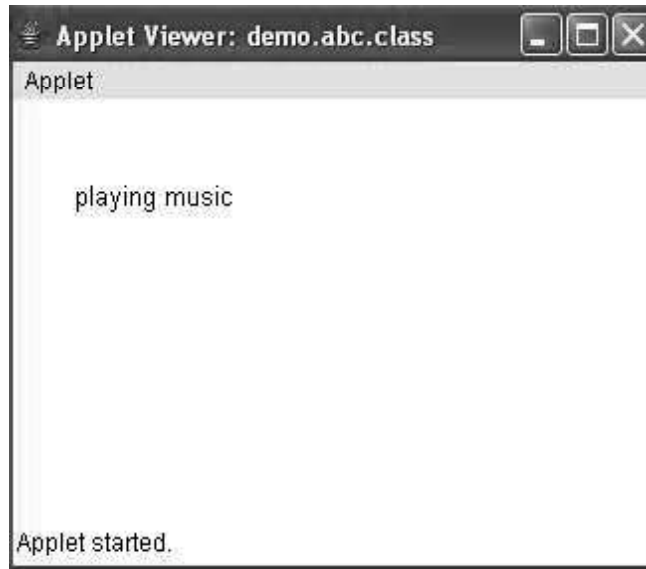
#### Example 4.7

```
import java.applet.*;
import java.awt.*;
publicclassplayerextendsApplet {
 AudioClipa1;
 publicvoidinit ()
 {
 a1=getAudioClip (getCodeBase (), "sai.au");
 }
 publicvoidstart ()
 {
 a1.play ();
 }
 publicvoidpaint (Graphicsg1)
 {
 g1.drawString ("playingmusic", 50, 30);
 }
}
/*<applet code="Play" width=200 height=200>
</applet>*/
```

## NOTES

## NOTES

### Output of the program:



This program will open the applet and the music file named sai.au will be played.

The `getAudioClip()` method returns the URL of the music file specified as its parameter value. Then the `play` method is called to play the music file.

One can only play the \*.au format music files and the music file should reside in the same directory in which the Java file is present.

The following is a designer applet to play and stop music files:

#### Example 4.8

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class Play extends applet implements ActionListener
{
 AudioClip a1;
 Button b1, b2;
 public void init ()
 {
 b1=new Button ("Start");
 b2=new Button ("Stop");
 a1=getAudioClip (getCodeBase (), "sai.au");
 b1.addActionListener (this);
 b2.addActionListener (this);
 add (b1);
 add (b2);
 }
 public void actionPerformed (ActionEvent ae)
 {
 if (ae.getSource () == b1)
 {
 a1.play ();
 }
 else {
 a1.stop ();
 }
 }
}
```



```
 }
 public void paint (Graphics g1)
 {
 g1.setColor (Color.red);
 g1.setFont (new Font ("Arial", Font.BOLD, 40));
 g1.drawString ("This Is Music World", 50, 70);
 }
}
/*<applet code="Play" width=200 height=200>
</applet>*/
```



## NOTES

### AppletContext

It is an interface that is implemented by an object that represents the environment of the applet. To use it, one has to first create an object of AppletContext, using the getAppletContext() method. This is shown below:

#### Example 4.9

```
AppletContext context=getAppletContext ();
```

In AppletContext, some interesting methods are defined such as the getApplet() method. It returns the name of the applet and the getApplets() method, which returns all the applets in the document base.

```
getApplet ()
getApplets ()
```

In AppletContext, another method, showDocument() is also defined, which either takes a URL or a URL and the file name in the form of a string. It creates an HTML page in the web browser, which shows the applet. It is useful with HTML frames where the applet should reside in a frame and the new HTML pages should be shown in another frame.

The showStatus() method shows the string passed to it as a parameter on the status bar at the bottom of the web browser. The following example shows the use of the showDocument() and showStatus() methods.

#### Example 4.10

```
AppletContext Code Sample
import java.applet.*;
import java.awt.*;
import java.net.*;
//<applet code="applet1.class" width=400 height=100></applet>
public class applet1 extends Applet {
 public void init () {
 try {
```

## NOTES

```
URL url1=new URL("http://wallpaper.net/pkomisar/
Flowere.html");
getAppletContext().showDocument(url1);
}catch (MalformedURLExceptionme){}
}
publicvoidpaint(Graphicsg1){
g1.setFont(newFont("Monospaced",Font.BOLD,22));
g1.drawString("HelloWorld",55,65);
getAppletContext().setStatus("Putsthemessageinstatusbar");
}
}
```

Now, in the above program, the user gets the current appletcontext using the `getAppletContext()` method. Then, using the `showDocument()` method, the user just transfers the control to another HTML file, which is passed to it as a parameter. The `setStatus()` method displays the string passed to it as a parameter at the bottom of the applet window.

**Note:** One can use console output in an applet such as `System.out.println()`. The string passed to it will not get displayed on the screen; rather it will get displayed in the console. It is generally used for the purpose of debugging; otherwise, the use of these methods is discouraged.

### Advantages of a Java Applet

A Java **applet** has the following advantages:

- (a) The applet can work properly on all versions of installed Java, excluding the latest plug-in version.
- (b) Almost all web browsers support the applet.
- (c) A user can permit it to have full access to the machine on which it is running.
- (d) It can improve with use, which means that after a first applet is run, the JVM is already running and starts quickly, benefitting the regular Java users. However, the JVM will need to restart each time the browser starts a fresh.
- (e) In terms of its speed of execution, it is slower than C++ codes, but faster than JavaScript.

### Disadvantages of a Java Applet

A Java applet has the following disadvantages:

- (a) Sometimes, the Java plug-in is required, but it is not available on every web browser by default.
- (b) The process of loading an applet is very slow.

It can now be concluded that without an applet, Java cannot get the honour of a web-enabled language. The AWT package present in Java is an important package. It is needed to develop a sophisticated applet.

### 4.5.3 Handling Events

Java provides various classes and interfaces to handle the events generated. Some of these have been discussed ahead.

## Event Classes

Java 1.1 event classes encapsulate all types of events occurring in the system. A successful handling of the events requires in-depth understanding of these classes. All these classes have been arranged in a hierarchy having `EventObject` class as their root element. This class acts as the superclass of all the event classes. `EventObject` class belongs to `java.util` package. Some of the event classes are discussed here.

## NOTES

### 1. The `ActionEvent` Class

The `ActionEvent` class represents the event that is generated when a user selects a menu item, presses a button or double-clicks a list item.

The `ActionEvent` class defines the following constructors:

```
ActionEvent (Object source, int event_type, String command_name)
ActionEvent (Object source, int event_type, String command_name, int
modifier_key)
ActionEvent (Object source, int event_type, String command_name, long
event_time, int modifier_key)
```

where,

- (i) `source` is a reference to the object that originated the event. The reference to the object can be obtained by using the `getSource ()` method.
- (ii) `event_type` specifies the type of the event. The `getId ()` method returns the type of the event.
- (iii) `command_name` is the name of the command which invoked the `ActionEvent` object which can be obtained by invoking the `getActionCommand ()` method.
- (iv) `event_time` represents the system time at which the event has occurred. The `getWhen ()` method returns the time of occurrence of the event.
- (v) `modifier_key` is the value of the modifier key (`ALT`, `CTRL`, `META`, `SHIFT`) which was pressed when the event was generated. The `getModifiers ()` method returns the value of the modifier key.

### 2. The `ComponentEvent` Class

The `ComponentEvent` class represents the event that is generated when the position, size or visibility of a component alters.

The `ComponentEvent` class defines the following constructor:

```
Component (Component source, int event_type)
```

where,

- (i) `source` is a reference to the component that originated the event. The reference to the component can be obtained by using `getComponent ()` method.
- (ii) `event_type` specifies the type of the event. The different types of component events are represented by integer constants—`COMPONENT_MOVED`, `COMPONENT_RESIZED`, `COMPONENT_HIDDEN`, `COMPONENT_SHOWN`.

## NOTES

### 3. The ContainerEvent Class

The `ContainerEvent` class inherits `ComponentEvent` class. The container event is generated when a component is added to or removed from a container.

The `ContainerEvent` class defines the following constructor:

```
ContainerEvent (Component source, int type, Component component)
```

where,

- (i) `source` is a reference to the component that originated the event. The reference to the component can be obtained by using `getContainer()` method.
- (ii) `event_type` specifies the type of the event. The different types of events are represented by integer constants—`COMPONENT_ADDED`, `COMPONENT_REMOVED`.
- (iii) `component` represents the component that has been added to or removed from the container and can be obtained by using the `getChild()` method

### 4. The FocusEvent Class

The `FocusEvent` class represents the event that is generated when a component gains or loses the focus of the input. The `FocusEvent` class is a subclass of the `ComponentEvent` class.

The `FocusEvent` class defines the following constructors:

```
FocusEvent (Component source, int event_type)
FocusEvent (Component source, int event_type, Boolean tempflag)
FocusEvent (Component source, int event_type, Boolean tempflag,
Component opp_component)
```

where,

- (i) `source` is a reference to the component that originated the event.
- (ii) `event_type` specifies the type of the event. The different types of events are represented by integer constants—`FOCUS_GAINED`, `FOCUS_LOST`.
- (iii) `tempflag` specifies whether the focus event is temporary. To determine if the focus event is temporary, the `isTemporary()` method is used.
- (iv) `opp_component` represents the another component that takes part in the focus change. It means if the `FOCUS_GAINED` event takes place then the `opp_component` is that component which lost focus. On the other hand, if `FOCUS_LOST` event occurs then the `opp_component` is that component which gains focus.

### 5. The ItemEvent Class

The `ItemEvent` class represents the event which is generated when a menu item is selected or deselected, checkbox item is clicked or a list item is selected.

The `ItemEvent` class defines the following constructor:

```
ItemEvent (ItemSelectable source, int event_type, Object item, int
item_state)
```

where,

- (i) `source` is a reference to the component that originated the event. The `getItemSelectable()` method returns the reference to the component that originated this event.
- (ii) `event_type` specifies the type of the event. The different types of events are represented by integer constants—`ITEM_SELECTED`, `ITEM_DESELECTED`.
- (iii) `item` is a reference to the item that originated the event. It can be obtained by using the `getItem()` method.
- (iv) `item_state` returns the current state of the item (selected or deselected) which generated the item event. The `getStateChange()` method returns the current state of the item.

## NOTES

### 6. The `KeyEvent` Class

The `KeyEvent` class represents the event that is generated when the user interacts with the application through keys. Java supports key events through the `KeyEvent` class.

The `KeyEvent` class defines the following constructor:

```
KeyEvent(Component source, int event_type, long event_time, int
modifier_key, int code, char character)
```

where,

- (i) `source` is a reference to the component that originated the event.
- (ii) `event_type` specifies the type of the event. The different types of events are represented by integer constants—`KEY_PRESSED`, `KEY_RELEASED`, `KEY_TYPED`.
- (iii) `event_time` represents the system time at which the event has occurred.
- (iv) `modifier_key` is the value of the modifier key (`ALT`, `CTRL`, `META`, `SHIFT`) which was pressed when the event was generated.
- (v) `code` returns the value of virtual key codes—`VK_0` to `VK_9`, `VK_A` to `VK_Z`, `VK_UP`, `VK_DOWN`, `VK_LEFT`, `VK_RIGHT` etc. The `getKeyCode()` method returns the value of the virtual key.
- (vi) `character` specifies the character that has been entered and can be obtained by using `getKeyChar()` method.

### 7. The `MouseEvent` Class

The `MouseEvent` class represents the event that is generated when the user interacts with the application through the mouse. The `MouseEvent` class encapsulates the mouse events.

The `MouseEvent` class defines the following constructor:

```
MouseEvent(Component source, int item_type, long event_time, int
modifier_key, int a, int b, int clicks_count, boolean popup_triggers)
```

where,

- (i) `source` is a reference to the component that originated the event.

## NOTES

- (ii) `event_type` specifies the type of the event. The different types of events are represented by integer constants—`MOUSE_CLICKED`, `MOUSE_EXITED`, `MOUSE_ENTERED`, `MOUSE_PRESSED`, `MOUSE_RELEASED`, `MOUSE_DRAGGED`, `MOUSE_MOVED`, `MOUSE_WHEEL`.
- (iii) `event_time` represents the system time at which the event occurred.
- (iv) `modifier_key` is the value of the modifier key (`ALT`, `CTRL`, `META`, `SHIFT`) which was pressed when the event was generated.
- (v) `int a`, `int b` represents the *X*, *Y* coordinates of the position of the mouse within the component, respectively. The `getX()` and `getY()` methods return the *X*, *Y* coordinates respectively. Alternatively, the `getPoint()` method can be used to obtain both the coordinates.
- (vi) `clicks_count` represents the total number of mouse clicks took place for this event. The `getClickCount()` method returns the total number of mouse clicks.
- (vii) `popuptriggers` specifies whether the pop-up window will appear. To determine if the pop-up window appears `isPopupTrigger()` method is used.

### 8. The `TextEvent` Class

The `TextEvent` class represents the event that is generated when the user enters or changes text in text fields or text areas. The `TextEvent` encapsulates text events.

The `TextEvent` class defines the following constructor:

```
TextEvent(Object source, int event_type)
```

where,

- (i) `source` represents the object that originated the event.
- (ii) `event_type` specifies the type of the event. The event is represented by integer constant—`TEXT_VALUE_CHANGED`.

### 9. The `WindowEvent` Class

The `WindowEvent` class represents the event that is generated when the state of the window changes. The `WindowEvent` class is a subclass of `ComponentEvent` class.

The `WindowEvent` class defines the following constructors:

- (i) `WindowEvent(WindowSource, int event_type)`
- (ii) `WindowEvent(WindowSource, int event_type, Window opposite)`
- (iii) `WindowEvent(WindowSource, int event_type, int from_State, int to_State)`
- (iv) `WindowEvent(WindowSource, int event_type, Window another, int from_State, int to_State)`

where,

`source` is a reference to the component that originated the event. The reference to the component is obtained by using `getWindow()` method.

## NOTES

`event_type` specifies the type of the event. The event is represented by integer constant—`WINDOW_ACTIVATED`, `WINDOW_DEACTIVATED`, `WINDOW_GAINED_FOCUS`, `WINDOW_LOST_FOCUS`, `WINDOW_OPENED`, `WINDOW_CLOSED`, `WINDOW_CLOSING`, `WINDOW_ICONIFIED`, `WINDOW_DEICONIFIED`, `WINDOW_STATE_CHANGED`.

`opposite` represents the another window when the window focus or window activation events took place and can be obtained by using `getOppositeWindow()` method.

`from_state` specifies the old state of the window, that is, the state before the state of the window changed. The `getOldState()` method returns the old state of the window.

`to_state` specifies the new state of the window, that is, the state after the state of the window changed. The `getNewState()` method returns the new state of the window.

### Event Sources

As stated earlier, event sources are the objects which generate events. Some of the examples of event sources are listed in the Table 4.15.

*Table 4.15 Some of the Event Sources*

Event Source	Description
Button	Generates action events when the button is pressed
Choice	Generates item events when choice changes
Window	Generates window events when the state of the window changes
Text components	Generates text events when a text is entered in text field or text area
Menu item	Generates action events when a menu item is selected, generates item events when a checkable menu is selected or deselected
Check box	Generates item events when a checkbox is selected or deselected
List	Generates action events when an item is double-clicked, generate action events when an item is selected or deselected
Scrollbar	Generates events when the scroll bar is manipulated

### Event Listener Interfaces

Events generated by the event source are sent to event listeners which handle them in an appropriate manner. Event listeners implement various interfaces. Some of the interfaces are discussed in this section.

#### 1. The ActionListener Interface

When an action event occurs, the `actionPerformed()` method defined by the `ActionListener` interface is invoked.

## NOTES

The general form of the `actionPerformed()` method is

```
void actionPerformed(ActionEvent e)
```

where,

`e` is the reference to an object of `ActionEvent` class

### 2. The `ComponentListener` Interface

The methods provided by the `ComponentListener` interface are invoked when the state of the component changes, that is, the component is resized, moved, shown or hidden.

The general form of the methods defined in `ComponentListener` interface are

- `void componentShown(ComponentEvent com)`
- `void componentHidden(ComponentEvent com)`
- `void componentMoved(ComponentEvent com)`
- `void componentResized(ComponentEvent com)`

where,

`com` is the reference to an object of the `ComponentEvent` class

### 3. The `ContainerListener` Interface

The methods defined by the `ContainerListener` interface are invoked when the component is added to or removed from a container. The object of `ContainerEvent` class is passed as method parameter.

The general form of the methods defined in `ContainerListener` interface are

```
void componentAdded(ContainerEvent con)
void componentRemoved(ContainerEvent con)
```

### 4. The `FocusListener` Interface

The methods defined by the `FocusListener` interface are invoked when the focus of the keyboard is either lost or gained by the component. The object of the `FocusEvent` class is passed as a method parameter.

The general form of the methods defined in `FocusListener` interface are

- `void focusGained(FocusEvent foc)`
- `void focusLost(FocusEvent foc)`

### 5. The `ItemListener` Interface

The method `itemStateChanged()` defined by the `ItemListener` interface is invoked when the state of an item is changed. The object of the `ItemEvent` class is passed as a method parameter.

The general form of the method defined in `ItemListener` interface is

```
void itemStateChanged(ItemEvent item)
```

### 6. The `KeyListener` Interface

The methods defined by the `KeyListener` interface are invoked when any key is pressed, released or a character is entered. The object of `KeyEvent` class is passed as a method parameter.



The general form of the methods defined in `KeyListener` interface are

- `void keyPressed (KeyEvent key)`
- `void keyReleased (KeyEvent key)`
- `void keyTyped (KeyEvent key)`

If a key B is pressed and released, then three events—key pressed, key typed and key released are generated.

## 7. The `MouseListener` Interface and `MouseMotionListener` Interface

The methods defined by the `MouseListener` interface are invoked when the mouse is clicked, pressed, released, when the mouse enters or exits the component. Similarly, the methods defined by the `MouseMotionListener` interface are invoked when the mouse is dragged or moved from one position to another. The object of `MouseEvent` class is passed as a method parameter.

The general form of the methods defined in `MouseListener` interface are

- `void mousePressed (MouseEvent me)`
- `void mouseReleased (MouseEvent me)`
- `void mouseClicked (MouseEvent me)`
- `void mouseEntered (MouseEvent me)`
- `void mouseExited (MouseEvent me)`

The general form of the methods defined in `MouseMotionListener` interface are

- `void mouseMoved (MouseEvent me)`
- `void mouseDragged (MouseEvent me)`

## 8. The `TextListener` Interface

The method `textChanged ()` defined by the `TextListener` interface is invoked when the text inside the text area or text field alters. The object of `TextEvent` class is passed as method parameter.

The general form of the method defined in `TextListener` interface is

```
void textChanged (TextEvent te)
```

## 9. The `WindowListener` Interface

The methods defined by the `WindowListener` interface are invoked when the state of the window changes. The object of `WindowEvent` class is passed as a method parameter.

The general form of the methods defined in `WindowListener` interface are

- `void windowOpened (WindowEvent we)`
- `void windowClosed (WindowEvent we)`
- `void windowClosing (WindowEvent we)`
- `void windowActivated (WindowEvent we)`
- `void windowDeactivated (WindowEvent we)`
- `void windowIconified (WindowEvent we)`
- `void windowDeiconified (WindowEvent we)`

To understand the concept of event handling, consider the following example. It demonstrates the generation of key events when the user presses or releases any keyboard key. The methods of the `KeyListener` interface are invoked when

## NOTES

## NOTES

the key is pressed or released and the output is displayed to the user in the status bar of the applet window.

### 4.5.4 AudioClip

AudioClip is an interface present in `java.applet` package. This interface is used to play an audio clip in the background of an applet. This interface supports only `.au` extension file.

#### How to Create an Object of AudioClip Interface?

As it is a interface, it cannot be instantiated. Through the `getAudioClip` method of `Applet` class programmer instantiate `getAudioClip` interface. This method is overridden.

```
public AudioClip getAudioClip (String name)
public AudioClip getaudioClip (URL u, String name)
```

#### Methods

```
public abstract void play ()
```

This method is used to play an audio file in the background of the applet.

```
public abstract void loop ()
```

This method is used to repeatedly play an audio file in the background of an applet.

```
public abstract void stop ()
```

This method is used to close an audio file.

#### How to Play an AudioClip?

An example may be seen below for understanding the concept before a detailed explanation is given.

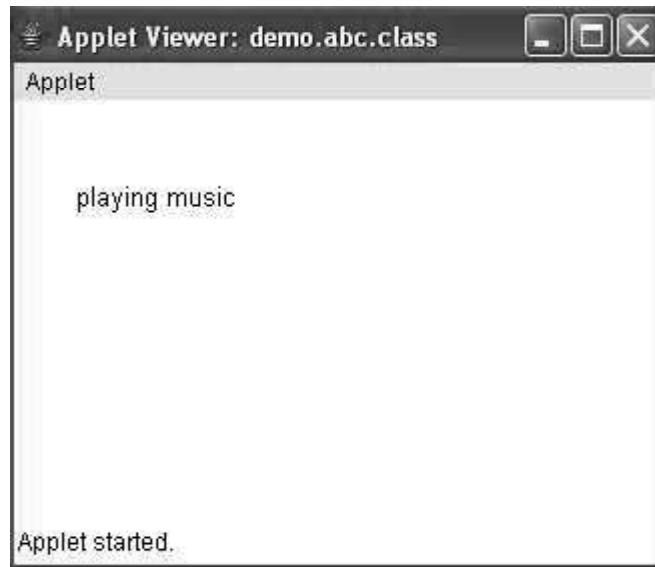
AudioClip is an interface present in `java.applet` package. This interface is used to play an audio file in the background of applet. As AudioClip is an interface it cannot be instantiated. Through the `getAudioClip ()` method of `Applet` class programmer instantiated AudioClip interface. This interface supports only `.au` extension file.

#### Program 4.28

```
import java.applet.*;
import java.awt.*;
public class player extends Applet {
 AudioClip a1;
 public void init ()
 {
 a1=getAudioClip (getCodeBase (), "sai.au");
 }
 public void start ()
 {
 a1.play ();
 }
}
```

```
public void paint (Graphics g1)
{
 g1.drawString ("playing music", 50, 30);
}
}
/*<applet code="Player" width=200 height=200>
</applet>*/
```

### Output of the program:



This program will open the **applet** and the music file named **sai . au** will be played.

The `getAudioClip ()` method returns the URL of the music file specified as its parameter value. Then `play` method is called to play the music file.

One can only play the **\* . au** format music files and the music file should reside in the same directory in which the Java file is present.

Given below is a designed **applet** to play and stop the music files:

### Program 4.29

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class Play extends Applet implements ActionListener
{
 AudioClip a1;
 Button b1, b2;
 public void init ()
 {
 b1=new Button ("Start");
 b2=new Button ("Stop");
 a1=getAudioClip (getCodeBase (), "sai . au");
 b1.addActionListener (this);
```

## NOTES

## NOTES

```
b2.addActionListener(this);
add(b1);
add(b2);
}
public void actionPerformed(ActionEvent ae)
{
 if(ae.getSource()==b1)
 {
 a1.play();
 }else{
 a1.stop();
 }
}
public void paint(Graphics g1)
{
 g1.setColor(Color.red);
 g1.setFont(new Font("Arial", Font.BOLD, 40));
 g1.drawString("This Is Music World", 50, 70);
}
}
/*<applet code="Play" width=200 height=200>
</applet>*/
```

### Output of the program:



---

## 4.6 AWT CLASSES

---

Abstract Window Toolkit (AWT) provides several graphics, windowing and user interface tools which are used to develop GUI of applets as well as stand-alone applications running in GUI environment. In order to use the classes and interfaces defined in AWT, `java.awt.*` package needs to be imported.

### AWT Classes

The AWT classes can be categorized into many groups which are discussed as follows:

### GUI Components

GUI components include visual elements that facilitate user's interaction. The `Component` class and `MenuComponent` class are the superclasses which

represent GUI components. The `Component` class is an abstract class, therefore its subclasses are used to create components. In AWT terminology, the user interface elements, such as button, text field and checkbox are termed as components.

**Table 4.16** Subclasses of Component Class

Class	Description
<code>Button</code>	It creates a push button control.
<code>Checkbox</code>	It creates a checkbox control.
<code>Choice</code>	It creates a dropdown list of textual entries.
<code>Label</code>	It creates a label control to display a string.
<code>List</code>	It creates a scrollable list of textual entries.
<code>ScrollBar</code>	It creates a scrollbar control for items.
<code>TextComponent</code>	It is a superclass of <code>TextField</code> and <code>TextArea</code> classes used to create single-line or multi-line textfields respectively.

## NOTES

The `Container` class is a subclass of `Component` class. It is used to contain various `Component` objects as well as other `Container` objects within it. The `Container` class' object groups, manages, and positions components and treats them as a unit. Two immediate subclasses of `Container` class are as follows:

- **Panel:** It is used for grouping components. An `Applet` is a subclass of `Panel`.
- **Window:** It is used for creating and handling windows. Two subclasses of `Window` are `Dialog` and `Frame`.

The `MenuComponent` class is an abstract class. Its immediate subclasses are `MenuBar` and `MenuItem` which are used to create menu bars and menu items, respectively.

## Layouts

Different layout classes are used for arranging, positioning, and determining the shape and size of the various components held by the container. All these classes implement `LayoutManager` interface of AWT package. Some of the commonly used classes are listed in Table 4.17.

**Table 4.17** Layout Classes

Class	Description
<code>BorderLayout</code>	It positions the components into five regions: east, west, north, south and center.
<code>CardLayout</code>	It arranges the components as a deck of cards such that only one component is visible at a time.
<code>FlowLayout</code>	It arranges the components horizontally.
<code>GridLayout</code>	It arranges the components into grid.

## Graphics Tools

The `Graphics` class encapsulates various methods for drawing various shapes and displaying output on the screen. Image loading and adding is supported by `Image` class. The classes, such as `Color` and `Font` are used to set display of graphical components. `Point`, `Polygon`, `Rectangle` contains methods to draw points, polygons and rectangles, respectively.

## Event Handlers

The `Event` class plays a significant role in handling GUI events. The `AWTEvent` class encapsulates various AWT events.

## NOTES

### 4.6.1 Window Fundamentals

The AWT defines various classes and methods that enable you to create and manage windows. The two important windows are those derived from class `Panel` and class `Frame`. The window derived from class `Panel` is used by applets and the class `Frame` creates a standard window. In this section, we will discuss the class hierarchy related to these two classes.

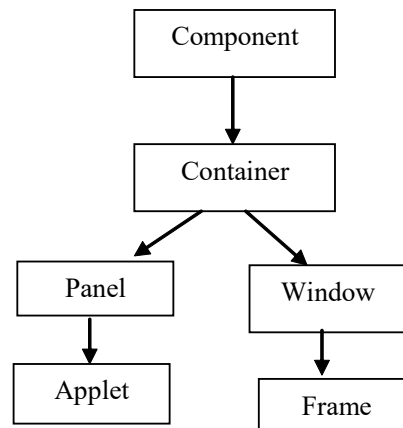


Fig. 4.10 Class Hierarchy for `Panel` and `Frame`

#### Component

The `Component` class is at the top of the AWT hierarchy. It is the abstract class; its subclasses are used to create the visual components such as button, text field, checkbox, etc. that facilitate user's interaction. It defines the various methods to manage events like mouse and keyboard input, resize and move the window, etc.

#### Container

The `Container` class is a subclass of `Component` class. It is used to contain various `Component` objects as well as other `Container` objects within it. The `Container` class' object groups, manages, and positions components and treats them as a unit.

#### Panel

The `Panel` class is a subclass of `Container`; it simply implements the `Container` class and does not add any new methods. It is the superclass of `Applet`. The output that is directed to the applet is actually displayed on the `Panel` object. A panel can be thought of as a window which does not contain any title bar, menu bar, or border. That is why, when we run an applet inside a browser, these items cannot be seen.

A panel only serves to organize the components contained in it. The components can be added directly into an applet or frame, but additional levels of grouping can be provided by adding components to panels and adding panels to a top-level applet or frame.

### **Window**

The `Window` class is also a subclass of `Container`. A top-level window can be created using the `Window` class. The top-level window is placed directly on the desktop and is not contained within any other object. Generally, the objects of the `Window` class are not directly created, rather, the `Frame` class, a subclass of `Window`, is used.

### **Frame**

`Frame` is a subclass of `Window`. It enables us to create an independent window for our application. Unlike `panel`, a `Frame` window has its own title bar, menu bar and resizing corners. It may also have pull down menu. We can think of a `Frame` window as a fully functioning window.

### **Canvas**

The `Canvas` class is not a part of the hierarchy for applet or frame windows. The canvas component is specially built to support graphics operation. It provides a blank window upon which you can draw.

## **4.6.2 Working With Frame Windows**

The `java.awt` class provides a class named **`Frame`**, which extends to the `Window` class and therefore, acts as a container itself.

The signature of this class is:

```
public class java.awt.Frame extends java.awt.Window
implements java.awt.MenuContainer
```

`Frame` is considered as a heavy weight container and is therefore, visible to the user. `Frame` cannot be executed on the browser. It is only meant for a stand-alone application. So, it cannot support the Web-enabled applications.

### **Constructors**

The constructors are:

```
Frame ()
```

It is used to generate a frame without any title.

```
Frame (java.lang.String)
```

This constructor is used to generate a frame, having a title as specified by the `String` type parameter.

To properly execute the `Frame` application, it is mandatory for the programmer to use two methods of the `Component` class. These two methods are:

- **`public void setSize(int, int)`**

## **NOTES**

## NOTES

This method is used for the purpose of setting the size of the frame with the specified coordinates.

- **public void setVisible (boolean)**

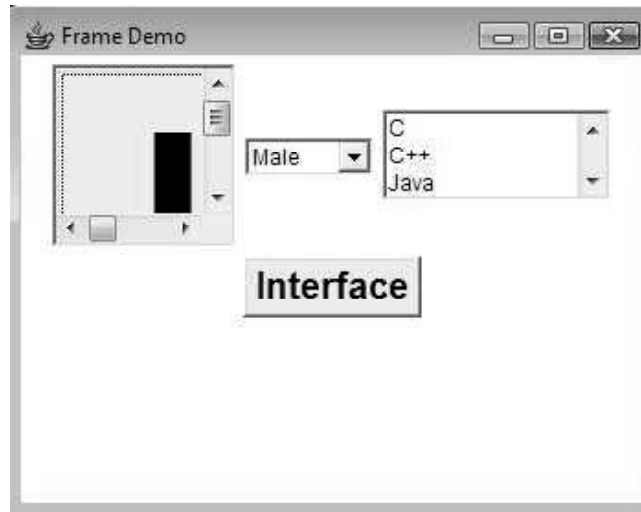
This method is used for setting the visibility of the frame. It takes a `boolean` value as its argument. If one passes `true`, then the frame is visible to the user. However, if `false` value is passed, then a user cannot see the frame on the screen.

### Program 4.30

```
import java.awt.*;
public class Frame1 extends Frame
{
 Button b1;
 Choice c1;
 List l1;
 ScrollPane sp1;
 public Frame1 ()
 {
 Frame f1=new Frame ("Frame Demo");
 b1=new Button ("Interface");
 b1.setFont (new Font ("Arial", Font.BOLD, 20));
 c1=new Choice ();
 c1.addItem ("Male");
 c1.addItem ("Female");
 l1=new List (3, true);
 l1.add ("C");
 l1.add ("C++");
 l1.add ("Java");
 l1.add (".Net");
 sp1=new ScrollPane ();
 Button b2=new Button ("Java Is A Language");
 b2.setFont (new Font ("Arial", Font.BOLD, 14));
 sp1.add (b2);
 f1.setLayout (new FlowLayout ());
 f1.add (sp1);
 f1.add (c1);
 f1.add (l1);
 f1.add (b1);
 f1.setSize (300, 280);
 f1.setVisible (true);
 }
 public static void main (String args [])
 {
 new Frame1 ();
 }
}
```



## Output of the program:



This example is a demonstration of the use of `Frame` class and how the container frame contains different components on it.

Here the user-defined class `Frame1` extends the `Frame` class to get the status of a frame. In the body of the class, four components, i.e. one `Button`, one `Choice`, one `List` and one `ScrollPane` are declared. Inside the constructor, the frame is instantiated and all the declared components are instantiated. `Frame` supports the `BorderLayout` by default. So, here, the layout is explicitly changed/set to `FlowLayout`. Then the components are attached to the frame by the invocation of the `add` method of the `Frame` class and passing the intended objects in its parameter. Then the size and the visibility of the frame are set by the invocation of pre-defined methods `setSize()` and `setVisible()`.

The frame then gets life by the instantiation of the user-defined frame class, i.e., `Frame1`, inside the `main()` method.

### 4.6.3 Frame Window and Event Handling in a Frame Window

The delegation model explains how event handling takes place in Java. You can map the event delegation model to the actual Java statements that are used for handling events. For example, when an end user clicks an AWT button, a message is displayed in the text field. The source in this case is the AWT button that the end user interacts with. The event listener that receives and processes the button click event in the example is the `ActionListener` interface. The class you will create to handle this event should implement the `ActionListener` interface. The `ActionListener` interface defines a single method, `actionPerformed()` that you will need to define within the class. The syntax of the user-defined class to handle event is:

```
public class EventExample extends Applet implements
ActionListener
{
public void init()
{
```

## NOTES

## NOTES

```
}
public void actionPerformed (ActionEvent ae)
{
}
}
```

The above syntax shows a class that implements the `ActionListener` interface and defines the `actionPerformed ()` method.

You will need to register the `ActionListener` interface to the button after creating the AWT button. The `addActionListener (reference)` method is used for registering the `ActionListener` interface. The syntax of the user-defined class to register event is:

```
public class EventExample extends Applet implements
ActionListener
{
 Button b1;
 public void init ()
 {
 b1 = new Button ("Message");
 b1.addActionListener (this);
 }
 public void actionPerformed (ActionEvent ae)
 {
 }
}
```

The above syntax shows how to use the `addActionListener ()` method to register the AWT button, which is the source to the `ActionListener` interface, which is the event listener.

The following program code shows the complete program to display a message in the text field when a button is clicked:

### Using Event Handling

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code="EventExamp" width=200 height=200>
</applet>*/
public class EventExamp extends Applet implements ActionListener
{
 TextField t1;
 Button b1;
 public void init ()
 {
 t1 = new TextField (20);
 b1 = new Button ("Message");
 b1.addActionListener (this);
 add (t1);
 add (b1);
 }
 public void actionPerformed (ActionEvent ae)
 {
 if (ae.getSource () == b1)
 {
 t1.setText ("Event Handling in Java");
 }
 }
}
```

```
}
}
}
```

The above code shows a process defined in the `actionPerformed()` method that displays a message in the Text Field. The object, `ae`, is the object of the `ActionEvent` class that is received by the `actionPerformed()` method when the button click event occurs.

Figure 4.11 shows the output.



Fig. 4.11 Displaying Message in Textbox

### ActionListener

The `ActionListener` interface declares a method that is called when an action event is generated. The method defined in `ActionListener` interface is a single method, `void actionPerformed(ActionEvent ae)`. You use the `ActionListener` interface when you handle events, such as clicking AWT buttons. The `addActionListener()` method enables tracing a button object for the occurrence of an action event. The following program code shows how to implement the `ActionListener` interface:

#### Implementing ActionListener Interface

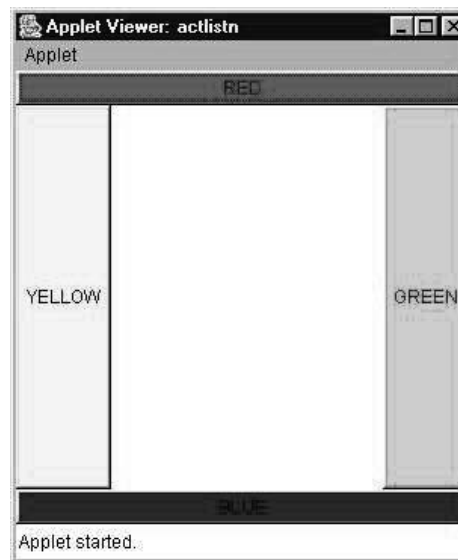
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code=actlistnheight=300width=300>
</applet>*/
public class actlistn extends Applet implements ActionListener
{
 Button b1, b2, b3, b4;
 public void init()
 {
 b1 = new Button("RED");
 b1.setBackground(Color.red);
 b2 = new Button("BLUE");
 b2.setBackground(Color.blue);
 b3 = new Button("GREEN");
 b3.setBackground(Color.green);
```

### NOTES

## NOTES

```
b4=newButton ("YELLOW");
b4.setBackground(Color.yellow);
setLayout (newBorderLayout ());
add (b1, "North");
add (b2, "South");
add (b3, "East");
add (b4, "West");
b1.addActionListener (this);
b2.addActionListener (this);
b3.addActionListener (this);
b4.addActionListener (this);
}
publicvoidactionPerformed (ActionEvent ae)
{
Objectob=ae.getSource ();
if (ob==b1)
setBackground (Color.red);
if (ob==b2)
setBackground (Color.blue);
if (ob==b3)
setBackground (Color.green);
if (ob==b4)
setBackground (Color.yellow);
}
}
```

The above program code shows implementing the `ActionListener` with the instances of `AWT Button` class. `Border layout` is set to the applet window and four button objects, `RED`, `BLUE`, `GREEN` and `YELLOW` are created. When you click a button the color of background of the window changes to the color of the button label. Figure 4.12 shows the output of the above program code.



**Fig. 4.12** Using `ActionListener` Interface

### **AdjustmentListener**

`AdjustmentListener` is responsible for handling all the events that are generated when an end user moves the scrollbar. The various integer constants

used by the adjustment listener method are:

- BLOCK\_DECREMENT
- BLOCK\_INCREMENT
- TRACK
- UNIT\_DECREMENT
- UNIT\_INCREMENT
- ADJUSTMENT\_VALUE\_CHANGED

### **MouseListener**

The `MouseListener` interface defines five methods for handling different operations that are associated with a mouse. These methods are:

- `void mouseClicked(MouseEvent me)`: Is invoked when you click a mouse button.
- `void mouseEntered(MouseEvent me)`: Is invoked when the mouse enters an AWT object to which the mouse event is associated.
- `void mouseExited(MouseEvent me)`: Is invoked when the mouse leaves an AWT object to which the mouse event is associated.
- `void mousePressed(MouseEvent me)`: Is invoked when the mouse key is pressed.
- `void mouseReleased(MouseEvent me)`: Is invoked when the mouse key is released.

The `addMouseListener()` method is used for monitoring an AWT object for the occurrence of a mouse event. The following program code shows how to implement the `MouseListener` interface:

### **Implementing the MouseListener Interface**

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code=mouslstnwidth=400height=300></applet>*/
public class mouslstn extends Applet implements MouseListener
{
 String msg = "";
 int mx = 0, my = 0;
 public void init()
 {
 addMouseListener(this);
 }

 public void mouseClicked(MouseEvent me)
 {
 mx = me.getX();
 my = me.getY();
 msg = "MouseClicked at Position "+mx+" "+my;
 repaint();
 }

 public void mouseEntered(MouseEvent me)
 {
 mx = 10;
 }
}
```

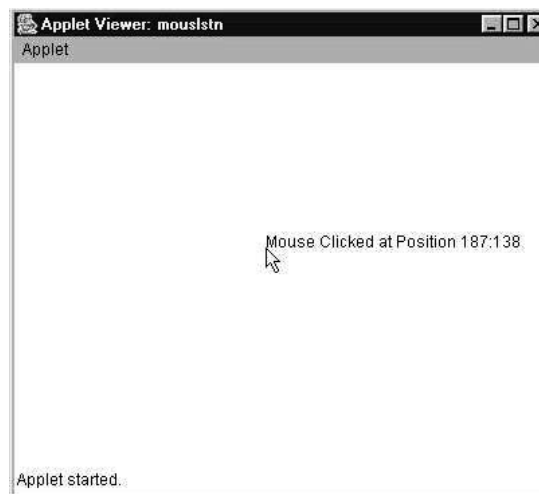
### **NOTES**

## NOTES

```
my=10;
msg="Mouse Entered";
repaint ();
}
public void mouseExited (MouseEvent me)
{
mx=10;
my=10;
msg="Mouse Left";
repaint ();
}
public void mousePressed (MouseEvent me)
{
}
public void mouseReleased (MouseEvent me)
{
}
public void paint (Graphics g)
{
g.drawString (msg, mx, my);
}
}
```

The above code shows how to implement `MouseListener` with the Applet window object. The `mouseClicked()` method displays the position where the mouse is clicked in the Applet window. The `me.getX()` method retrieves the position of the X coordinate where the mouse is clicked. The `me.getY()` method retrieves the position of the Y coordinate where the mouse is clicked. The `mouseEntered()` method displays a message, `Mouse Entered`, when the mouse enters the applet window. The `mouseExited()` method displays a message, `Mouse Exited`, when the mouse enters the applet window.

Figure 4.13 shows the output.



**Fig. 4.13** Using the `MouseListener` Interface

**Note:** You need to give empty implementation of the methods that you do not create in a program.

## **MouseMotionListener**

MouseMotionListener is another built-in interface for handling mouse events. This interface handles events that are associated with the movement of the mouse. MouseMotionListener defines two methods that are associated with the movement of a mouse. These methods are:

- void mouseMoved (MouseEvent me) : Is invoked when you move a mouse on an AWT object that is being traced for mouse movement events.
- void mouseDragged (MouseEvent me) : Is invoked when a mouse button is pressed and the mouse is moved on an AWT object that is being traced for mouse movement events.

The addMouseListener () method is used for tracing an AWT object for the occurrence of a mouse movement event. The following program code shows how to implement the MouseMotionListener interface:

### **Implementing MouseMotionListener**

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet code=freedrawheight=300width=300>
</applet>*/
public class freedraw extends Applet implements
MouseMotionListener
{
 int x, y;

 public void init ()
 {
 setLayout (new GridLayout (5, 2));
 x=0;
 y=0;
 addMouseListener (this);
 }

 public void mouseDragged (MouseEvent me)
 {
 x=me.getX ();
 y=me.getY ();
 repaint ();
 }

 public void mouseMoved (MouseEvent me)
 {}

 public void update (Graphics g)
 {
 paint (g);
 }

 public void paint (Graphics g)
 {
 g.fillOval (x, y, 5, 5);
 }
}
```

## **NOTES**

## NOTES

The above code shows how to implement the `MouseMotionListener` interface with the applet window. A freehand drawing mechanism is created where dragging the cursor plots pixels at those positions creating an image. The `mouseDragged()` method is used for retrieving and plotting the pixel positions.

Figure 4.14 shows the output.



*Fig. 4.14 Freehand Drawing Using MouseMotionListener*

## WindowListener

The `WindowListener` interface defines seven methods for handling window events. An application of the `WindowListener` interface is to close a `Frame` window. The seven methods of the `WindowListener` interface are:

- `void windowActivated(WindowEvent we)`: Is invoked when a window is activated.
- `void windowDeactivated(WindowEvent we)`: Is invoked when a window is deactivated.
- `void windowOpened(WindowEvent we)`: Is invoked when a window is opened.
- `void windowClosed(WindowEvent we)`: Is invoked when a window is closed.
- `void windowClosing(WindowEvent we)`: Is invoked when an end user clicks the close button in a window.
- `void windowIconified(WindowEvent we)`: Is invoked when a window is minimized.
- `void windowDeiconified(WindowEvent we)`: Is invoked when a window is restored.



The `addWindowListener()` method enables monitoring a window for the occurrence of a window event. The following program code shows how to implement the `WindowListener` interface:

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

### Implementing the `WindowListener` Interface

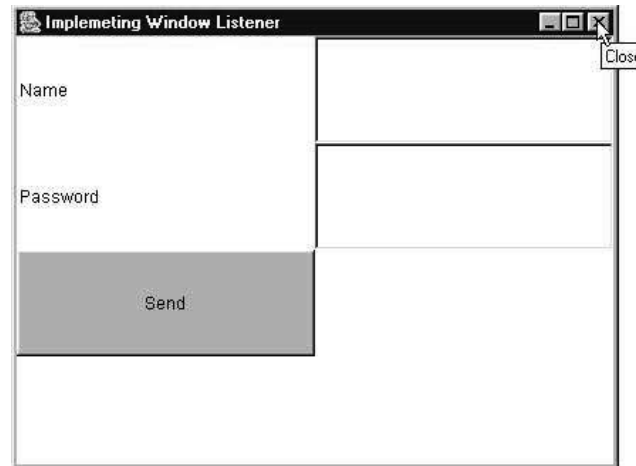
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class windowlistn extends Frame implements WindowListener
{
 Label l1, l2;
 TextField t1, t2;
 Button b1;
 public windowlistn ()
 {
 super ("Implementing WindowListener");
 setLayout (new GridLayout (4, 2));
 l1=new Label ("Name");
 l2=new Label ("Password");
 t1=new TextField (10);
 t2=new TextField (10);
 t3.setEchoChar ('*');
 b1=new Button ("Send");
 add (l1);
 add (t1);
 add (l2);
 add (t2);
 add (b1);
 addWindowListener (this);
 }
 public static void main (String ar[])
 {
 windowlistnd=new windowlistn ();
 d.setSize (400, 400);
 d.setVisible (true);
 }
 public void windowClosing (WindowEvent we)
 {
 this.setVisible (false);
 System.exit (0);
 }
 public void windowActivated (WindowEvent we)
 {}
 public void windowDeactivated (WindowEvent we)
 {}
 public void windowOpened (WindowEvent we)
 {}
 public void windowClosed (WindowEvent we)
 {}
 public void windowIconified (WindowEvent we)
 {}
 public void windowDeiconified (WindowEvent we)
 {}
}
```

### NOTES

## NOTES

The above code shows closing a Frame window using the `WindowListener` interface. The `windowClosing()` method uses the `setVisible` method to close the frame window. The `System.exit(0)` method stops the program from running.

Figure 4.15 shows the output.



*Fig. 4.15 Closing the Frame Window*

## KeyListener

The `KeyListener` interface declares methods that handle input from the keyboard. The methods defined in the `KeyListener` interface are:

- `void keyPressed(KeyEvent ke)`: Is called when an end user presses a key on the keyboard.
- `void keyReleased(KeyEvent ke)`: Is called when an end user releases a key on the keyboard.
- `void keyTyped(KeyEvent ke)`: Is called when an end user types a key on the keyboard, which is the operation of pressing and releasing the key in one fast action.

The `requestFocus()` method of the `Component` class has to be included in your program for receiving keyboard events. The `addKeyListener()` method is used for monitoring keyboard events on objects. The following program code shows how to use the `KeyListener` interface for handling events:

### Implementing the KeyListener Interface

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="test" width="300" height="200">
</applet>*/

public class test extends Applet implements KeyListener
{
 String msg[] = {"", "", "", "", "", "", "", "", "", ""};
 int x = 15;
```

```
int c=x;
int y=15;
int i=0;
public void init()
{
 addKeyListener(this);
 requestFocus();
}
public void keyPressed(KeyEvent ke)
{
 System.out.println(msg);
 if(c<200)
 {
 c=c+5;
 msg[i]+=ke.getKeyChar();
 System.out.println(msg);
 }
 else
 {
 x=15;
 y=y+20;
 c=x;
 i=i+1;
 msg[i]+=ke.getKeyChar();
 }
 if(i>9)
 {
 for(int k=0;k<10;k++)
 {
 msg[0]="";
 }
 x=15;
 y=15;
 c=x;
 i=0;
 }
 repaint();
}
public void keyReleased(KeyEvent ke)
{}
public void keyTyped(KeyEvent ke)
{}
public void paint(Graphics g)
{
 g.drawString(msg[i],x,y);
 System.out.println(msg);
}
public void update(Graphics g)
{
 paint(g);
}
}
```

The above code shows implementing the `KeyListener` Interface for accepting input from the keyboard to the applet window. The `getKeyChar()`

## NOTES

## NOTES

method enables taking input from the keyboard and displaying the information in the applet window. The `keyPressed` method is used for defining the statements that are run when an end user enters input from the keyboard.

Figure 4.16 shows the output.

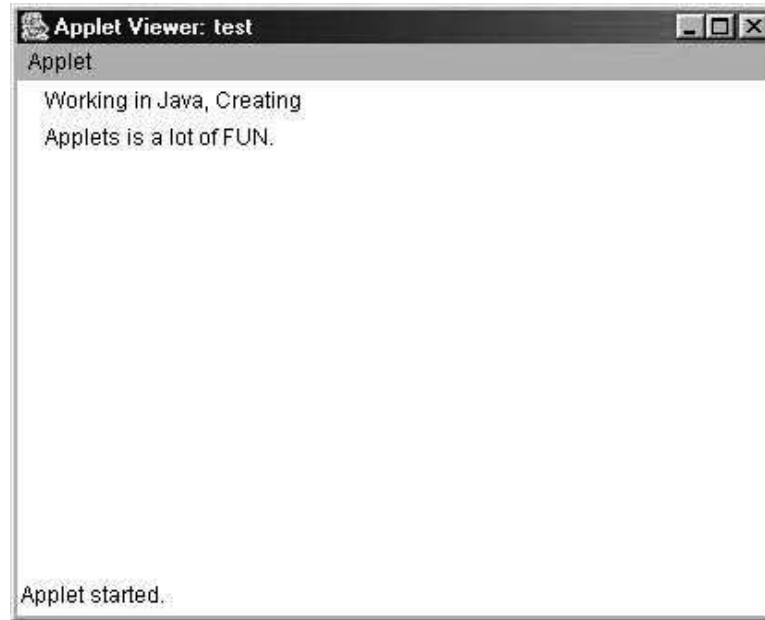


Fig. 4.16 Using `KeyListener`

### 4.6.4 Display Information While Working with Graphics and Color

The `Graphics` class provides different methods to draw and fill various shapes.

#### Drawing Lines

A line can be drawn using the `drawLine()` method of the `Graphics` class. This method takes four parameters, which represent the coordinates of the end points of the line.

The general form of the `drawLine()` method is

```
void drawLine(int a1, int b1, int a2, int b2)
```

where,

`a1, b1` is the coordinate of the starting points of the line.

`a2, b2` is the coordinate of the ending points of the line.

For example, the statement `gra.drawLine(20, 100, 90, 100)` will draw a straight line from the coordinate point `(20, 100)` to `(90, 100)` as shown in Figure 4.17.

\_\_\_\_\_

**(20, 100)** **(90, 100)**

Fig. 4.17 A Straight Line Having Coordinates `(20,100)` and `(90,100)`

## Drawing and Filling Rectangles

A rectangle can be drawn by using the `drawRect ()` method and it also takes the four parameters.

The general form of the `drawRect ()` method is

```
void drawRect (int a1, int b1, int w, int h)
```

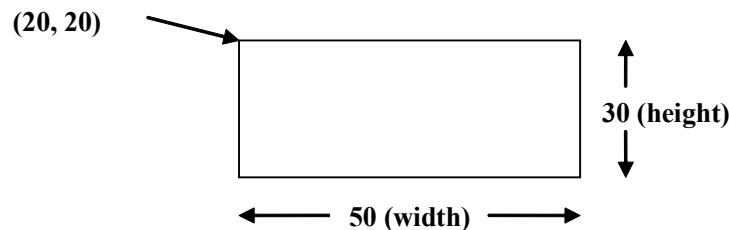
where,

`a1, b1` is the coordinate of the top left corner of the rectangle.

`w` is the width of the rectangle.

`h` is the height of the rectangle.

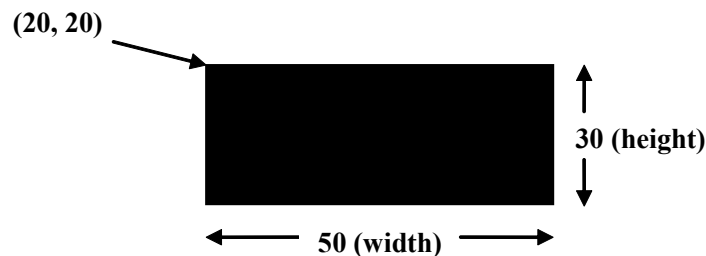
For example, the statement `gra.drawRect (20, 20, 50, 30)` will draw a rectangle starting at `(20, 20)` with width of 50 pixels and height of 30 pixels as shown in Figure 4.18.



*Fig. 4.18 A Rectangle with Width 50 pixels and Height 30 pixels*

Note that the `drawRect ()` method draws only the boundary of the rectangle. To draw a solid (filled) rectangle, `fillRect ()` method is used. This method also takes four parameters similar to the `drawRect ()` method.

To draw a solid rectangle having aforementioned parameters, we use the statement `gra.fillRect (20, 20, 50, 30)`, which draws the rectangle as shown in Figure 4.19.



*Fig.4.19 A Filled Rectangle Having Width 50 pixels and Height 30 pixels*

A rounded outlined rectangle can be drawn by using the `drawRoundRect ()` method. This method takes the six parameters (Refer Figure 4.20).

The general form of the `drawRoundRect ()` method is

```
void drawRoundRect (int a1, int b1, int w, int h, int xdia, int ydia)
```

## NOTES

## NOTES

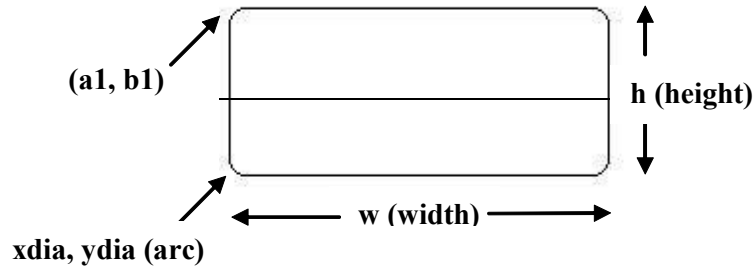


Fig. 4.20 A Rounded Rectangle

where,

$x\text{dia}$  is the diameter of the rounding arc (along X-axis).

$y\text{dia}$  is the diameter of the rounding arc (along Y-axis).

Similarly, a rounded filled rectangle can be drawn using `drawFillRoundRect()` method. This method also takes six parameters similar to the `drawRoundRect()` method.

**Note:** All the shapes are drawn relative to the Java's coordinate system. The origin  $(0, 0)$  of the coordinate system is located at its upper-left corner such that the positive  $x$  values are to its right and the positive  $y$  values are to its bottom.

### Drawing and Filling Ellipses and Circles

An ellipse can be drawn using the `drawOval()` method. The ellipse is drawn within an imaginary bounding rectangle. This method takes four arguments in which the first two represent the top left corner of the bounding rectangle and the next two represent the width and height of the oval or the bounding rectangle (Refer Figure 4.21).

The general form of the `drawOval()` method is

```
void drawOval(int a1, int b1, int w, int h)
```

where,

$a1, b1$  is the coordinate of the top left corner of the bounding rectangle.

$w$  is the width of the bounding rectangle.

$h$  is the height of the bounding rectangle.

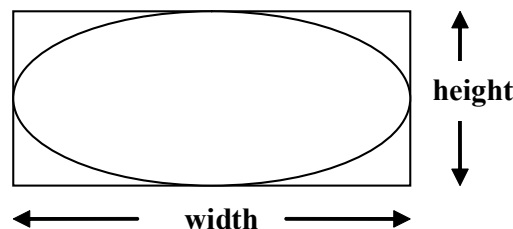


Fig. 4.21 An Ellipse

Similarly, a circle can be drawn using this method but the dimension of width and height should be same. That is, the bounding rectangle must be a square.

Similar to the rectangle methods, the `drawOval()` method draws the boundary of an oval and the `fillOval()` method draws a solid oval.

### Drawing Arcs

An arc can be drawn using the `drawArc()` method. This method takes six arguments in which the first four are same as the arguments of the `drawoval()` method and the next two represents the starting angle of the arc and the sweep angle around the arc, respectively.

The general form of the `drawArc()` method is

```
void drawArc(int a1, int b1, int w, int h, int strt_angle, int sweep_angle)
```

where,

`a1, b1` is the coordinate of the top left corner of the bounding rectangle.

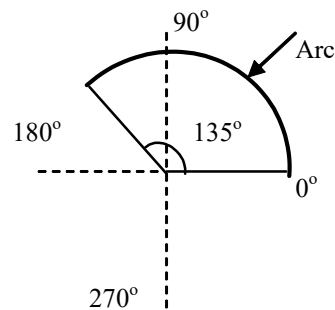
`w` is the width of the bounding rectangle.

`h` is the height of the bounding rectangle.

`strt_angle` is the starting angle of the arc (in degrees).

`sweep_angle` is the number of degrees (angular distance) around the arc (in degrees).

The arc shown in Figure 4.22 has the starting angle as  $0^{\circ}$  degrees and sweep angle as  $135^{\circ}$ .



**Fig. 4.22** An Arc of  $135^{\circ}$  Sweep Angle

You can also draw filled arcs using the `fillArc()` method.

### Drawing Polygons

A polygon is a closed geometrical figure, which can have any number of sides. A polygon can be drawn by using the `drawPolygon()` method. This method takes the three parameters (Refer Figure 4.23).

The general form of the `drawPolygon()` method is

```
void drawPolygon(int a[], int b[], int n)
```

where,

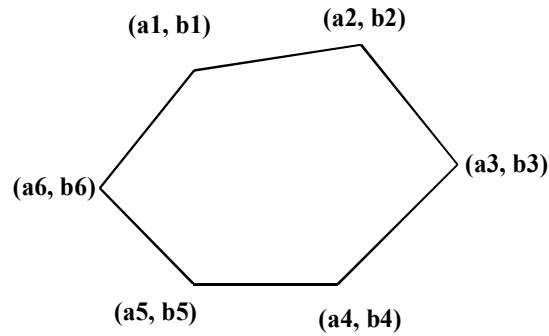
`a[]` is the array of integers having x-coordinates.

`b[]` is the array of integers having y-coordinates.

`n` is the total number of coordinate points required to draw a polygon.

### NOTES

## NOTES



*Fig. 4.23 A Polygon having Six Sides*

### **Example 4.11:** An applet code to demonstrate the use of various methods of Graphics class

```
import java.awt.*;
import java.applet.*;
public class GraphicsExample extends Applet
{
 public void paint (Graphics gra)
 {
 gra.drawRect (10, 40, 80, 40);
 gra.fillRect (130, 40, 80, 40);
 gra.drawRoundRect (250, 40, 80, 40, 8, 8);
 gra.fillRoundRect (370, 40, 80, 40, 8, 8);

 gra.drawOval (0, 125, 80, 40);
 gra.fillOval (120, 125, 80, 40);

 gra.drawArc (240, 125, 80, 40, 0, 180);
 gra.fillArc (370, 125, 80, 40, 0, 180);

 int x[]={100, 150, 200, 170, 130, 100};
 int y[]={250, 200, 250, 300, 300, 250};
 int n=x.length;
 gra.drawPolygon (x, y, n);

 gra.drawOval (270, 200, 80, 80);
 gra.fillOval (380, 200, 80, 80);

 gra.drawLine (480, 50, 480, 350);
 }
}
```

The HTML code for GraphicsExample is

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<CENTER>
<APPLET
 CODE="GraphicsExample.class"
 WIDTH = "600"
 HEIGHT="350">
</APPLET>
```



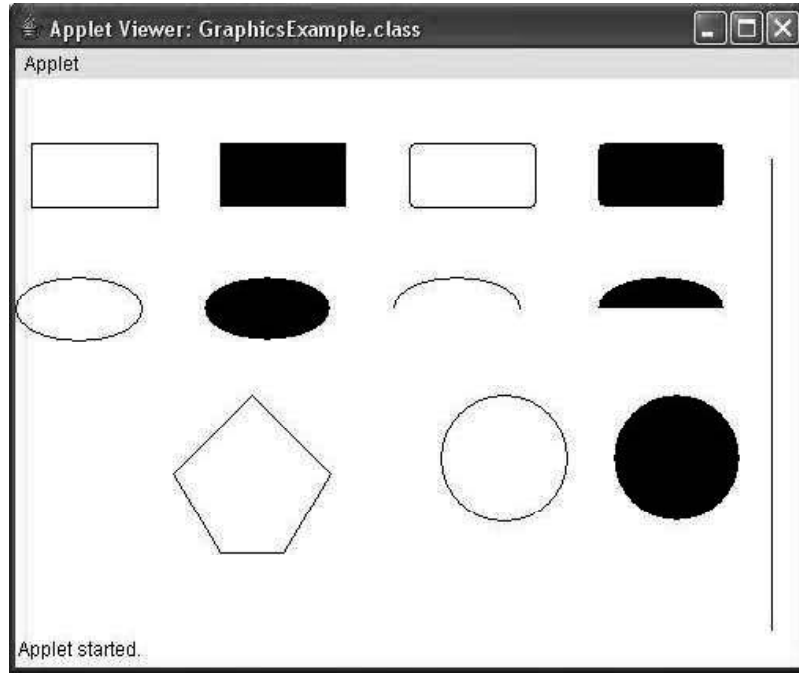
```

 </CENTER>
 </BODY>
</HTML>

```

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

The output of the HTML code is:



## NOTES

### Using Color Class

The `Color` class provides various methods to use any color you want in the display. It defines the various color constants which can be directly used only by specifying the color of your choice. In addition, the `Color` class allows creation of millions of colors. The `Color` class contains three primitive colors namely, *red*, *blue* and *green* and all other colors are a combination of these three colors.

One of the constructors that is used to create color of your choice is

```
Color (int red, intgreen, intblue)
```

where,

red, green, blue can take any value between 0 and 255.

### Setting Background and Foreground Color

To set the color of the background of an applet window, `setBackground()` method is used.

The general form of the `setBackground()` method is

```
voidsetBackground (mycolor)
```

Similarly, to set the foreground color to a specific color, that is, the color of text, `setForeground()` method is used.

The general form of the `setForeground()` method is

```
voidsetForeground (mycolor)
```

## NOTES

where,

`mycolor` is one of the color constants or the new color created by the user.

The list of color constants is as follows:

- `Color.red`
- `Color.orange`
- `Color.gray`
- `Color.darkGray`
- `Color.lightGray`
- `Color.cyan`
- `Color.pink`
- `Color.white`
- `Color.blue`
- `Color.green`
- `Color.black`
- `Color.yellow`

**Example 4.12:** An applet code to demonstrate the use of `Color` class

```
import java.applet.*;
import java.awt.*;
public class ColorExample extends Applet
{
 Color c1, c2;
 public void init ()
 {
 //creating new colors
 Color c1=new Color(0,0,255);
 Color c2=new Color(100,220,190);
 }
 public void paint (Graphics gra)
 {
 setBackground(Color.white); //setting background
 color

 //drawing a line of color c1
 gra.setColor(c1);
 gra.drawLine(10,20,150,60);

 //drawing a solid oval of color c2
 gra.setColor(c2);
 gra.fillOval(10,50,100,200);

 //drawing a rectangle of red color
 gra.setColor(Color.red);
 gra.drawRect(150,120,60,120);
 }
}
```

The HTML code for `ColorExample` is

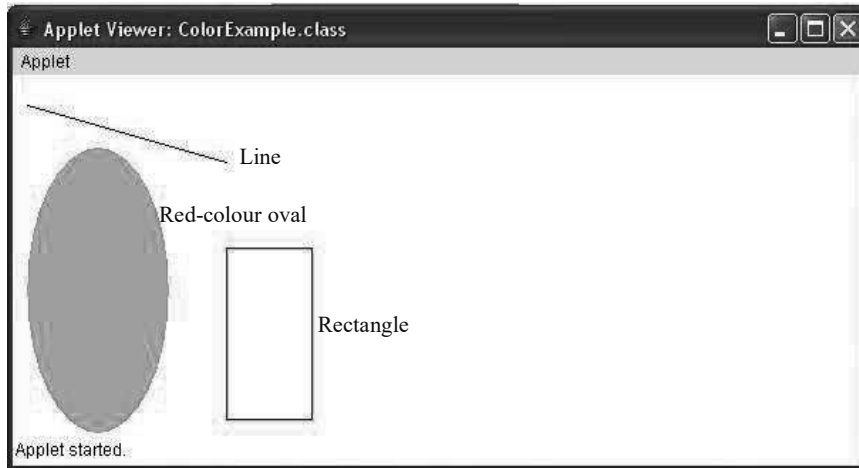
```
<HTML>
<HEAD>
</HEAD>
<BODY>
<CENTER>
<APPLET
CODE="ColorExample.class"
```

```

 WIDTH ="600"
 HEIGHT="250">
 </APPLET>
</CENTER>
</BODY>
</HTML>

```

The output of the HTML code is:



## NOTES

### 4.6.5 Working with Fonts

The `Font` class is used to apply different font styles to the text. To select or apply a new font, a font object should be constructed.

The syntax of the constructor of `Font` class is as follows:

```
Font (String font_name, int font_style, int font_size)
```

where,

`font_name` is the name of the font.

`font_style` is font style.

`font_size` is the size of the font in points.

Some other methods of `Font` class are listed in Table 4.6. Also, refer Example 4.13 to understand the use of `Font` class.

**Table 4.18** Methods of `Font` Class

Method	Description
<code>static Font getFont()</code>	It returns the currently selected font.
<code>int getSize()</code>	It returns the size of the font.
<code>String getName()</code>	It returns the name of the font.
<code>int getStyle()</code>	It returns the style of the font.
<code>String getFamily()</code>	It returns the name of the family of the font.

**Example 4.13:** A program to demonstrate the use of `Font` class is as follows:

```

import javax.swing.*.*;
import java.awt.*.*;
public class FontExample extends JPanel
{
 public void paintComponent (Graphics gra)
 {

```

## NOTES

```
super.paintComponent (gra) ;
Graphics2Dgra2D= (Graphics2D) gra;
FontMyFont=newFont ("CourierNew", Font.BOLD, 16) ;
gra2D.setFont (MyFont) ;
gra2D.drawString ("Hello Java", 20, 40) ;
Font f=gra2D.getFont () ;
String fontName=f.getName () ;
gra2D.drawString ("Font name is :"+fontName, 20, 80) ;
String fontFamily=f.getFamily () ;
gra2D.drawString ("Font family is :"+fontFamily, 20, 120) ;
int fontSize=f.getSize () ;
gra2D.drawString ("Font size is :"+fontSize, 20, 160) ;
int fontStyle=f.getStyle () ;
gra2D.drawString ("Font style is :"+fontStyle, 20, 200) ;

}
publicstaticvoidmain (String[] args)
{
FontExample fe=newFontExample () ;
JFrame fr=newJFrame ("Graphics") ;
fr.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE) ;
fr.add (fe) ;
fr.setSize (400, 300) ;
fr.setLocationRelativeTo (null) ;
fr.setVisible (true) ;
}
}
```

### Check Your Progress

10. Define the term Applet.
11. What does the ComponentEvent class represent?
12. Define the term AudioClip.
13. What is window class?
14. Define ActionListener interface.
15. What is the use of addMouseListener () method?

## 4.7 AWT CONTROLS AND LAYOUT MANAGERS

The components that allow a user to interact with a GUI-based application are called controls. The various controls supported by AWT are labels, push buttons, checkboxes, choice list, list, scrollbars, text area, text field, menu bar, etc. The controls can be added to or removed from a window using the methods defined in the Container class.

The syntax to add a control is as follows:

```
Component add (Component obj)
```

The syntax to remove a control is as follows:

```
void remove (Component obj)
```

where,

`obj` is an instance of the control that is to be added or removed.

To remove all the controls from a window, `removeAll()` method can be used.

Some of the commonly used controls are discussed as follows:

### **Label**

A label is a simple control which is used to display text (non-editable) on the window. Since it does not possess any user interactive feature, it is considered as a passive control.

The `Label` class defines the following constructors:

```
Label() //first
Label(String str1) //second
Label(String str1, int str1_align) //third
```

The first constructor creates a blank label. The second constructor creates a label containing the string specified by `str1`. The third constructor creates a label containing the string `str1` and `str1_align` determines the alignment of the text contained in the label. It can be one of these: `Label.LEFT`, `Label.RIGHT` or `Label.CENTER`. The default alignment of text is `LEFT`.

### **Push Button or Button**

A push button is an active control that has a 3-dimensional appearance. It displays the text and triggers an event when it is clicked or activated.

The `Button` class defines the following constructors:

```
Button() //first
Button(String str1) //second
```

The first constructor creates a button with blank label. The second constructor creates a button having `str1` as label.

### **Checkbox**

A checkbox is a control that consists of a combination of a small box and a label. The label provides the description of the box with which it is associated. It is a two-state control having states `true` (checked) and `false` (unchecked). The state of a checkbox can be changed by clicking on it. `Checkbox` is an object of `Checkbox` class.

The `Checkbox` class defines the following constructors:

```
Checkbox() / /
first
Checkbox(String str1) //second
Checkbox(String str1, boolean state) //third
Checkbox(String str1, boolean state, CheckboxGroup group)

//fourth
```

## **NOTES**

## NOTES

The first constructor creates a checkbox with an empty label. The second constructor creates a checkbox having `str1` as label. The third constructor creates a checkbox having `str1` as label and `state` is used to set the initial state of the checkbox. If it is `true`, the checkbox is checked, else it is unchecked. The default state of the checkbox is `false`. In the fourth constructors the `group` represents the name of the group to which the checkbox belongs.

The checkbox group can be divided into two categories: non-mutually exclusive group, in which more than one item of the group can be checked and mutually exclusive group, in which only one item of the group can be selected.

### Choice

A choice control creates a drop-down list of textual entries. When the user clicks on it a list of choices appears. The user can select only one of the items contained in the list and only the selected item is displayed. Choice is an object of `Choice` class.

The `Choice` class defines only the default constructor, that is, `Choice ()` which creates an empty choice list.

In order to add items to the list, `add ()` method is used which takes the name of the item to be added as an argument.

### List

List, like Choice, displays a list of items. However, list allows the user to make multiple selections from the given list of items. Using list any number of items can be displayed and multiple selections can be made. List is an object of `List` class.

The `List` class defines the following constructors:

```
List ()
//first
List (int Rows) / /
second
List (int Rows, boolean multi_select) //third
```

The first constructor creates an empty list which allows only single selection of items. The second constructor creates a list where `Rows` specifies the number of visible entries in the list. In the third constructor the `multi_select` represents the allowable selections in the list. If it is `true`, then multiple selections are allowed at a time otherwise only one item can be selected at a time

To add items to the list, `add ()` method is used.

### Text Field and Text Area

Text field and text area controls create a single-line and multi-line text area, respectively. These controls are provided by two classes which inherit `TextComponent` class.

The text field allows the user to enter and edit the text in the text field using cut, copy and paste keys, arrow keys and mouse selections. It is an object of `TextField` class.

The `TextField` class defines the following constructors:

```
TextField() / /
first
TextField(int cols) //second
TextField(String str1) //third
TextField(String str1, int cols) //fourth
```

The first constructor creates an empty text field. The second constructor creates a text field in which `cols` represents the width of the text field. That is, the maximum number of characters the text field can contain. The third constructor creates a text field where `str1` specifies the string contained in the text field. The fourth constructor creates a text field containing string `str1` and having its width represented by `cols`.

The text area creates a multi-line text area. It is an object of `TextArea` class.

The `TextArea` class defines the following constructors:

```
TextArea() //first
TextArea(String str1) //second
TextArea(int rows, int cols) //third
TextArea(String str1, int rows, int cols) //fourth
TextArea(String str1, int rows, int cols, int scroll) //fifth
```

The first constructor creates an empty text area. The second constructor creates a text area containing a string specified by `str1`. The third constructor creates a text area such that `rows` represents the height of the text area. That is, the maximum number of lines the text area can contain and `cols` represents the width of the text area. That is, the maximum number of characters each line of the text area can contain. The fourth constructor creates a text area having string `str1`, height equals to `rows` and width equal to `cols`. The fifth constructor creates a text area similar to the one created using fourth constructor except for `scroll` which specifies the scrollbars the text area will have. It can take one of these values—`SCROLLBARS_NONE`, `SCROLLBARS_BOTH`, `SCROLLBARS_HORIZONTAL_ONLY` and `SCROLLBARS_VERTICAL_ONLY`.

### Scrollbar

Scrollbars are horizontally or vertically oriented bars which allow the user to select items between a specified minimum and maximum values. Each end of the scrollbar has an arrow which can be clicked to change the current value of the scrollbar. The slider box indicates the current value of the scrollbar which can be dragged by the user to a new position. Scrollbar is an object of `Scrollbar` class.

The `Scrollbar` class defines the following constructors:

```
Scrollbar() //first
Scrollbar(int orientation) //second
Scrollbar(int orientation, int initial_value, int visible_units,
int min, int max) //third
```

The first constructor creates a vertical scrollbar. The second constructor creates a scrollbar in which `orientation` specifies the orientation of the scrollbar which can take one of the following values `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`. The third constructor creates a scrollbar in which `orientation` specifies the orientation of the scrollbar, `initial_value`

### NOTES

represents the initial value of the scrollbar, `visible_units` represents the number of visible items of a scrollbar at a time and `min` and `max` represents the minimum and maximum values for the scrollbar.

## NOTES

### AWT Layout Managers

As already discussed, AWT container is an instance of `Container` class which holds various components and other containers. Components are first created, and then added to a container. Each container object in Java has a default layout manager which determines the dimension and exact positioning of these components within a container. For example, `window` is a container that may contain components, such as buttons, labels and text fields. When the components are added to the window, the layout manager in effect will determine the size and placement of these components inside the window. The `java.awt` package provides various predefined layout managers each of which implements `LayoutManager` interface. These layout managers differ in the manner they arrange various components within a component.

The general form to set a specific type of layout manager is as follows:

```
void setLayout (LayoutManager lm)
```

where,

`lm` is a reference to the desired layout manager.

If `setLayout ()` method is not invoked, then the default layout manager is invoked automatically.

Java AWT provides different layout managers, each of which implements its own layout policy. Some of the commonly used layout managers are discussed here.

### FlowLayout

`FlowLayout` is the simplest of all the layout managers. It positions the components in the order they are added to the container. It places the components from left to right, that is, in horizontal rows. Once a row gets completely filled with components then the remaining components are placed in the next row. It is the default layout manager for `Applet` and `Panel`. Each component is evenly separated from its neighbor components by leaving a small space not only from above and below it, but also from left and right.

The `FlowLayout` class defines the following constructors:

```
FlowLayout ()
 //first
FlowLayout (int alignment) / /
 second
FlowLayout (int alignment, int hor, int ver) //third
```

The first constructor creates a default layout. It positions the components in the center and leaves a space of five pixels between each component. The second constructor creates a flow layout in which `alignment` specifies the alignment of laid out components. It can take one of these constants—`FlowLayout.LEFT`, `FlowLayout.RIGHT` and `FlowLayout.CENTER`. The third constructor creates a flow layout in which `hor` and `ver` specify the horizontal and vertical space left between each component, respectively



and alignment specifies the alignment of components. Example 4.2 demonstrates the use of `FlowLayout` class.

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

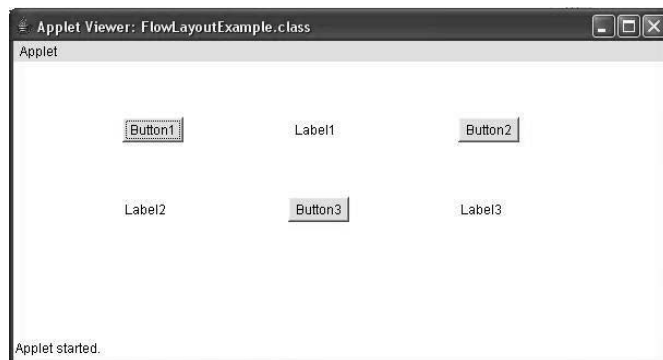
**Example 4.14:** An applet program to demonstrate the use of `FlowLayout` class is as follows:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class FlowLayoutExample extends Applet
{
 Button b1, b2, b3;
 Label l1, l2, l3;
 public void init ()
 {
 setLayout (new FlowLayout (FlowLayout.LEFT, 100, 50));
 b1=new Button ("Button1");
 b2=new Button ("Button2");
 b3=new Button ("Button3");
 l1=new Label ("Label1");
 l2=new Label ("Label2");
 l3=new Label ("Label3");
 add (b1);
 add (l1);
 add (b2);
 add (l2);
 add (b3);
 add (l3);
 }
}
```

The HTML code for `FlowLayoutExample` is as follows:

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
 <CENTER>
 <APPLET CODE="FlowLayoutExample.class"
 WIDTH=600
 HEIGHT=250>
 </APPLET>
 </CENTER>
</BODY>
</HTML>
```

The output of the HTML code is:



## NOTES

## NOTES

### BorderLayout

BorderLayout is the default layout manager for Window, Dialog and Frame classes. It positions the components into five regions—east, west, north, south and center. Each region can contain a single component. When a component is added to a particular region, it is extended to fit that region. That is, only one component can be placed within a region at a time. An attempt to add more than one component to a region hides the previously added component by the recently added component.

The BorderLayout class defines the following constructors:

```
BorderLayout () //first
BorderLayout (int hor, int ver) //second
```

The first constructor creates a default border layout. In the second constructor hor and ver specify the horizontal and the vertical space left between each component, respectively.

To add a component to a specific region, add () method is used.

The general form of add () method is as follows:

```
void add (Component obj, Object region)
```

where,

obj is the component to be added.

region represents the name of region where the component is added. It can take one of these constants, BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.CENTER, BorderLayout.EAST and BorderLayout.WEST

Example 4.15 demonstrates the use of BorderLayout class.

**Example 4.15:** An applet to demonstrate the use of BorderLayout class is as follows:

```
import java.applet.*;
import java.awt.*;
public class BorderLayoutExample extends Applet
{
 Button b1, b2, b3, b4, b5;
 Label l1;
 public void init ()
 {
 //set layout for the applet to be BorderLayout
 setLayout (new BorderLayout ());

 //creating components
 b1=new Button ("Left");
 b2=new Button ("Right");
 b3=new Button ("Top");
 b4=new Button ("Bottom");
 b5=new Button ("Middle");
 l1=new Label ("Center");

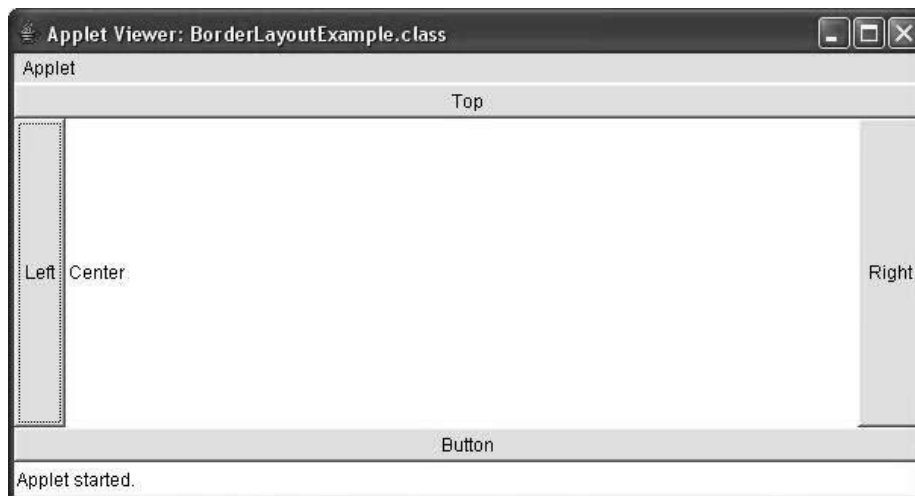
 //adding components
 add (b1, BorderLayout.WEST);
 add (b2, BorderLayout.EAST);
```

```
 add (b3, BorderLayout.NORTH) ;
 add (b4, BorderLayout.SOUTH) ;
 add (b5, BorderLayout.CENTER) ;
 add (l1, BorderLayout.CENTER) ; /*label hides the button*/
 }
}
```

The HTML code for BorderLayoutExample is as follows:

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
 <CENTER>
 <APPLET CODE="BorderLayoutExample.class"
 WIDTH=600
 HEIGHT=250>
 </APPLET>
 </CENTER>
</BODY>
</HTML>
```

The output of the HTML code is:



## GridLayout

The GridLayout manager subdivides the specified region into a grid/matrix of rows and columns. The overall available region is divided equally between rows and columns, such that each row and each column in the layout is of the same size. The component added to the cell gets expanded to the size of that cell. Example 4.16 illustrates the use of GridLayout class.

The GridLayout class defines the following constructors:

```
GridLayout () //first
GridLayout (int rows, int cols) //second
GridLayout (int rows, int cols, int hor, int ver) //third
```

The first constructor creates a single-column layout. The second constructor creates a grid layout in which rows represents the number of rows and cols represents the number of columns in the grid layout. The third constructor allows to specify

## NOTES

## NOTES

the horizontal and vertical space left between each component using `hor` and `ver` respectively.

**Example 4.16:** An applet to demonstrate the use of `GridLayout` class is as follows:

```
import java.applet.*;
import java.awt.*;
public class GridLayoutExample extends Applet
{
 Button b1, b2;
 Checkbox c1;
 Scrollbar s1;
 Label l1;
 TextField t1;
 TextArea ta1;
 Choice choicelist;
 List list;
 String Year[] = {"January", "February", "March", "April",
 "May", "June", "July", "August", "September", "October",
 "November", "December"};
 String Week[] = {"Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday", "Saturday", "Sunday"};
 public void init()
 {
 /*creates a gridlayout having 3 rows and 3 columns and
 12 pixel horizontal and vertical spacing between
 components*/
 setLayout (new GridLayout (3, 3, 12, 12));

 //creating various components
 b1 = new Button ("Button1");
 c1 = new Checkbox ("Checkbox", true);
 s1 = new Scrollbar ();
 b2 = new Button ("Button2");
 l1 = new Label ("Label Component");
 t1 = new TextField ("TextField Component");
 ta1 = new TextArea (5, 20);
 choicelist = new Choice ();
 list = new List (7, true);

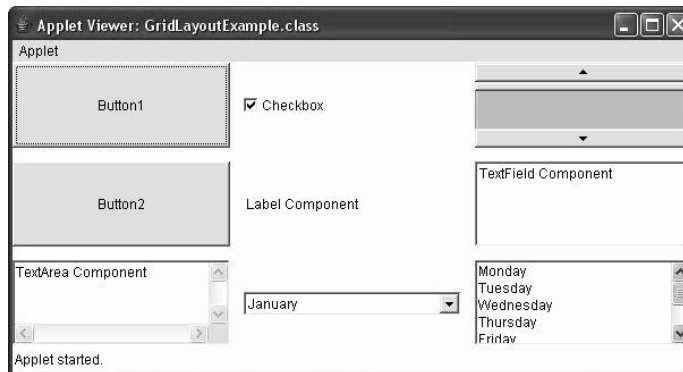
 //adding components to the applet window
 add (b1);
 add (c1);
 add (s1);
 add (b2);
 add (l1);
 add (t1);
 add (ta1);
 for (int i = 0; i < Year.length; ++i)
 {
 choicelist.add (Year[i]);
 }
 add (choicelist);
 ta1.setText ("TextArea Component");
 for (int i = 0; i < Week.length; ++i)
 {
```

```
 list.add(Week[i]);
 }
 add(list);
}
}
```

The HTML code for GridLayoutExample is as follows:

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
 <CENTER>
 <APPLET CODE="GridLayoutExample.class"
 WIDTH=600
 HEIGHT=250>
 </APPLET>
 </CENTER>
 </BODY>
</HTML>
```

The output of the HTML code is:



### CardLayout

CardLayout is used to manage a large number of components. It organizes the components in layers. The components are arranged like a deck of cards such that only one of them is visible at a time. In order to use card layout, it is required to create an object of type Panel that will hold the cards. All these cards form must also be the objects of type Panel. The layout manager of the panel must be set to CardLayout.

The CardLayout class defines the following constructors:

```
CardLayout() //first
CardLayout(int hor, int ver) //second
```

The first constructor creates a default card layout. The second constructor creates a card layout in which hor and ver specify the horizontal and vertical space left between each component respectively. Example 4.17 illustrates the use of CardLayout class.

**Example 4.17:** An applet to demonstrate the use of CardLayout class is as follows:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
```

### NOTES

## NOTES

```
public class CardLayoutExample extends Applet implements ActionListener
{
 Panel mainPanel; // this container will hold the cards
 Panel p1, p2; // these panels will represent the cards
 Panel b; // this panel will hold buttons
 Button button1, button2; // buttons
 CardLayout cl; // card layout object
 public void init ()
 {
 mainPanel = new Panel ();
 cl = new CardLayout ();
 mainPanel.setLayout (cl);
 // set mainPanel's layout
 // to be CardLayout
 // creating two panels (card panels) to show
 p1 = new Panel ();
 p1.add(new Label ("First Card"));
 p1.setBackground(Color.pink);
 p2 = new Panel ();
 p2.add(new Label ("Second Card"));
 p2.setBackground(Color.yellow);
 // creating two buttons and add ActionListener
 button1 = new Button ("First");
 button1.addActionListener (this);
 button2 = new Button ("Second");
 button2.addActionListener (this);
 // creating Panel for adding buttons to it
 b = new Panel ();
 b.add(button1);
 b.add(button2);
 // setting layout for the applet to be BorderLayout
 this.setLayout (new BorderLayout ());
 this.add (b, BorderLayout.SOUTH);
 this.add (mainPanel, BorderLayout.CENTER);
 // adding two card panels to the main panel container
 mainPanel.add (p1, "First");
 mainPanel.add (p2, "Second");
 }
 // Button clicks will respond by showing the so named Panel
 public void actionPerformed (ActionEvent e)
 {
 if (e.getSource () == button1)
 cl.show (mainPanel, "First");
 if (e.getSource () == button2)
 cl.show (mainPanel, "Second");
 }
}
```

The HTML code for CardLayoutExample is as follows:

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
 <CENTER>
 <APPLET CODE="CardLayoutExample.class"
 WIDTH=600
 HEIGHT=250>
```

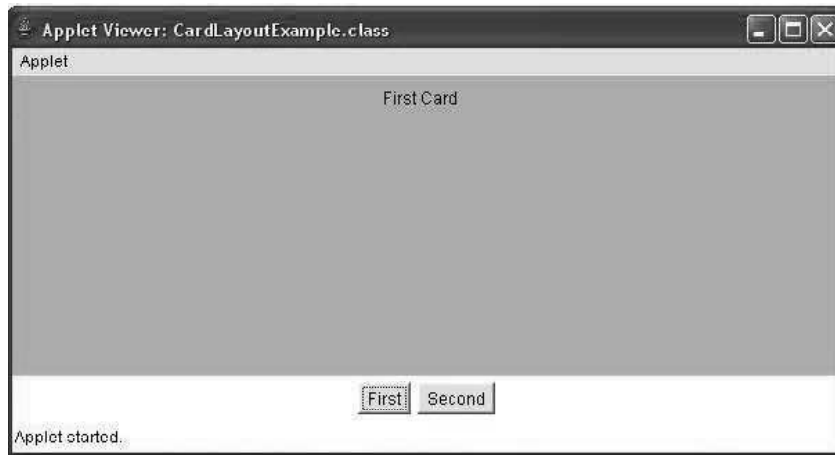
```

 </APPLET>
 </CENTER>
</BODY>
</HTML>

```

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

**The output of the HTML code is:**



**NOTES**

### 4.7.1 AWT Menus

A `MenuBar` represents a list of menus which can be added to the top of a top level window. Each menu is associated with a drop-down list of menu items. The concept of `MenuBar` can be implemented by using three Java classes, namely `MenuBar`, `Menu` and `MenuItem`.

A `MenuBar` is represented by the object of class `MenuBar` in which only the default constructor is defined. A `MenuBar` may consist of one or more menus represented by the objects of the class `Menu`. A menu includes a list of menu items represented by the objects of the class `MenuItem`.

The `Menu` class defines the following constructors:

```

Menu () //first
Menu (String optionName) //second
Menu (String optionName, boolean removable) //third

```

The first constructor creates an empty `Menu`. The second constructor creates a `Menu` with the name of the `Menu` specified by `optionName`. In the third constructor, `removable` may have one of the two values: `True` or `False`. If it is `True`, then the `Menu` can float freely in the application window; otherwise, it remains attached to the `MenuBar`.

The `MenuItem` class defines the following constructors:

```

MenuItem () //first
MenuItem (String itemName) //second
MenuItem (String itemName, MenuShortcut shortcut) //third

```

The first constructor creates a `MenuItem` with no name and no menu shortcut. In the second constructor, `itemName` specifies the name of the `MenuItem`. In the third constructor, `shortcut` specifies the menu shortcut associated with the `MenuItem`.

## NOTES

A checkable MenuItem can also be created by using CheckboxMenuItem, which is a subclass of MenuItem.

The CheckboxMenuItem defines the following constructors:

```
CheckboxMenuItem()
CheckboxMenuItem(String itemName)
CheckboxMenuItem(String itemName, boolean checkable)
```

The first constructor creates an unchecked MenuItem with no name. The second constructor creates an unchecked MenuItem with the name of the MenuItem specified by the itemName. In the third constructor, checkable can either be True or False. If it is True, then the MenuItem is initially checked, otherwise, it is unchecked.

There are two types of menus which are given as follows:

- **Regular Menus:** They are placed at the top of the application window within a MenuBar.
- **Pop-Up Menus:** They appear in the window when the user clicks. For example, a pop-up menu appears on the right-click of the mouse.

**Program 4.30:** An applet to demonstrate the use of MenuBar class, Menu class and MenuItem class

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
class Mframe extends Frame
{
 String message = "";
 Mframe (String title)
 {
 super (title);
 MenuBar mb = new MenuBar ();
 setMenuBar (mb);
 Menu m1 = new Menu ("File");
 MenuItem New = new MenuItem ("New");
 MenuItem open = new MenuItem ("Open");
 MenuItem close = new MenuItem ("Close");
 MenuItem save = new MenuItem ("Save");
 MenuItem exit = new MenuItem ("Exit");
 m1.add (New);
 m1.add (open);
 m1.add (close);
 m1.add (save);
 m1.addSeparator (); //creates a horizontal line for
 //partitioning
 m1.add (exit);
 mb.add (m1);
 Menu m2 = new Menu ("Edit");
 MenuItem cut = new MenuItem ("Cut");
 MenuItem copy = new MenuItem ("Copy");
 MenuItem paste = new MenuItem ("Paste");
 m1.addSeparator (); //creates a horizontal line for
 //partitioning
 Menu search = new Menu ("Search");
 MenuItem find = new MenuItem ("Find");
```



```

MenuItem replace=newMenuItem("Replace");
search.add(find);
search.add(replace);
m2.add(cut);
m2.add(copy);
m2.add(paste);
m2.add(search);
mb.add(m2);
Handler handler=newHandler(this);
New.addActionListener(handler);
open.addActionListener(handler);
close.addActionListener(handler);
save.addActionListener(handler);
exit.addActionListener(handler);
cut.addActionListener(handler);
copy.addActionListener(handler);
paste.addActionListener(handler);
search.addActionListener(handler);
find.addActionListener(handler);
replace.addActionListener(handler);
}
publicvoidpaint(Graphicsg)
{
 g.drawString(message,10,200);
}
}
classHandlerimplementsActionListener
{
 Mframemf;
 publicHandler(Mframemf)
 {
 this.mf=mf;
 }
 publicvoidactionPerformed(ActionEventae)
 {
 Stringmessage="Youclicked:";
 Stringstr=(String)ae.getActionCommand();
 if(str.equals("New"))
 message=message+"New";
 if(str.equals("Open"))
 message=message+"Open";
 if(str.equals("Close"))
 message=message+"Close";
 if(str.equals("Save"))
 message=message+"Save";
 if(str.equals("Exit"))
 message=message+"Exit";
 if(str.equals("Cut"))
 message=message+"Cut";
 if(str.equals("Copy"))
 message=message+"Copy";
 if(str.equals("Paste"))
 message=message+"Paste";
 if(str.equals("Search"))
 message=message+"Search";
 }
}

```

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

## NOTES

## NOTES

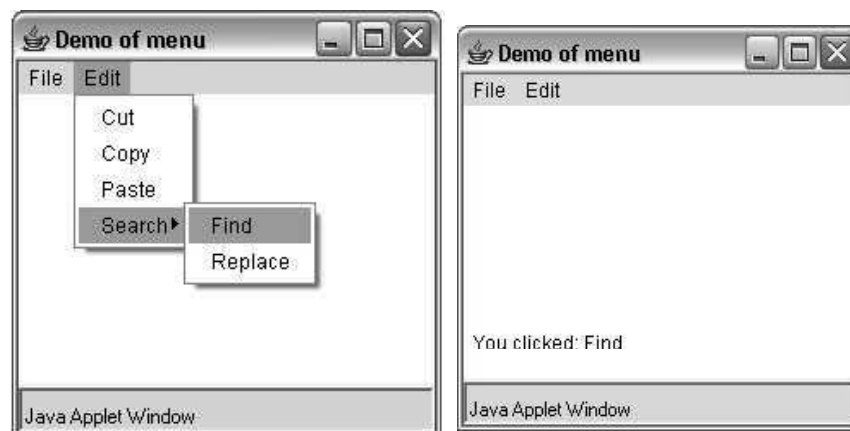
```
 if(str.equals("Find"))
 message=message+"Find";
 if(str.equals("Replace"))
 message=message+"Replace";

 mf.message=message;
 mf.repaint();
 }
}
public class MenuBarDemo extends Applet
{
 Frame f;
 public void init()
 {
 f=new Mframe("Demo of menu");
 f.setSize(250, 250);
 f.setVisible(true);
 }
}
```

The HTML code for MenuBarDemo is

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
 <CENTER>
 <APPLET
 CODE="MenuBarDemo.class"
 WIDTH=600
 HEIGHT=250>
 </APPLET>
 </CENTER>
 </BODY>
</HTML>
```

**Output of the Program:**



## 4.7.2 Dialog Class

The `java.awt` package provides a class known as `FileDialog`. It is a child class of `Dialog` class. The signature of this class is:

```
public class java.awt.FileDialog extends
java.awt.Dialog
```

This class displays a dialog box on the screen, so that the user can choose a file from it. Until the dialog window is closed; rest of the application goes to a blocked state. The `FileDialog` is known as a modal dialog.

A `FileDialog` has two modes:

- `LOAD` mode
- `SAVE` mode

### Constructors

The various constructors are:

```
FileDialog (Frame)
```

For the purpose of loading a file, a `FileDialog` is constructed by this constructor.

```
FileDialog (Frame, String)
```

This constructor also does the same action as of the above. However, here the dialog window gets a label on it.

```
FileDialog (Frame, String, int)
```

With addition to the above action, here the mode for the file is specified, i.e., either `FileDialog.LOAD` or `FileDialog.SAVE`.

### Methods

The commonly used methods of the `FileDialog` class are:

```
public java.lang.String getFile ()
```

This method is used to retrieve the file from the corresponding file dialog.

```
getFileNameFilter ()
```

It is used to get the filtered file name from the file dialog.

```
public int getMode ()
```

This method is used to get the mode of the concerned file dialog, which is either `LOAD` or `SAVE`.

```
getDirectory ()
```

It is used to get the directory name of the concerned file dialog.

```
setDirectory (java.lang.String)
```

The directory name of the concerned file dialog can be set by this method with the specified name.

```
public void setFile (java.lang.String)
```

One can set the specified file for the concerned file dialog by this method.

```
public void setMode (int)
```

One can set the specified mode for the concerned file dialog by this method.

### NOTES

## NOTES

### Check Your Progress

16. Explain the term drawing lines.
17. Define the term color class.
18. What is meant by AWT controls?
19. Elaborate the term layout manager.
20. What does a `MenuBar` represent?

## 4.8 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. By the use of `File` class, one can directly deal with the files, directories and file system of the platform. Actually, Java does not provide a crystal clear view of how things are done in the background when the programmer uses the `File` class. A programmer has to create the `File` object through the constructors provided by the `File` class.
2. A directory is a collection of files and directories. In Java, directories are also treated as files. If one wishes to deal with the directories, then the `list ()` method can be used. When the `list ()` method is invoked by the directory object (created through `File` class constructor), then the list of other files and directories are extracted from it. This method is overloaded. One of them is `String [] list ()`.
3. A stream means a channel or a pipe. Like flow of water in a pipe, data flows from the source to the destination through the channels in Java.
4. There are two types of streams: input streams and output streams.
5. The Internet provides TCP/IP protocols for delivery services.
6. The combination of IP address and port number is known as socket.
7. Domain name is very difficult to remember the IP address to connect to the Internet. The Domain Name System (DNS) is used to overcome this problem. DNS maps one particular IP address to a string of characters, which is popularly known as domain name.
8. The `InetAddress` class is used to resolve the domain names to their IP addresses and vice versa.
9. If any error occurs while creating the datagram socket, `SocketException` is thrown.
10. An applet is a dynamic and interactive program that can run inside a web page displayed by a Java-capable browser, such as a HotJava Browser or an Internet Explorer browser, which are World Wide Web (WWW) browsers used to view web pages. An applet is a class present in a `java.applet` package. A special HTML tag is embedded in an applet to make it run on the web browser. The appletviewer application, present in the jdk, is used to run and check the applets. An applet has added advantages, such as

frame, event-handing facility, graphics context and surrounding user interfaces.

11. The `ComponentEvent` class represents the event that is generated when the position, size or visibility of a component alters.
12. `AudioClip` is an interface present in `java.applet` package. This interface is used to play an `AudioClip` in the background of an applet. This interface supports only `.au` extension file.
13. The `Window` class is also a subclass of `Container`. A top-level window can be created using the `Window` class. The top-level window is placed directly on the desktop and is not contained within any other object. Generally, the objects of the `Window` class are not directly created, rather, the `Frame` class, a subclass of `Window`, is used.
14. The `ActionListener` interface declares a method that is called when an action event is generated. The method defined in `ActionListener` interface is a single method, `void actionPerformed, ActionEvent`.
15. The `addMouseListener ()` method is used for monitoring an AWT object for the occurrence of a mouse event.
16. A line can be drawn using the `drawLine ()` method of the `Graphics` class.
17. The `Color` class provides various methods to use any color you want in the display. It defines the various color constants which can be directly used only by specifying the color of your choice. In addition, the `Color` class allows creation of millions of colors. The `Color` class contains three primitive colors namely, red, blue and green and all other colors are a combination of these three colors.
18. AWT controls are components which allow a user to interact with the application. Some of the AWT controls are labels, buttons, checkbox, choice, list, text field, text area, scroll bar, etc.
19. A layout manager determines the dimension and exact positioning of components within a container. For example, window is a container that may contain components, such as buttons, labels and text fields.
20. A `MenuBar` represents a list of menus which can be added to the top of a top level window.

## NOTES

---

## 4.9 SUMMARY

---

- By the use of `File` class, one can directly deal with the files, directories and file system of the platform. Actually, Java does not provide a crystal clear view of how things are done in the background when the programmer uses the `File` class. A programmer has to create the `File` object through the constructors provided by the `File` class.

## NOTES

- File class constructors are used for the creation of the object of the File class.
- A directory is a collection of files and directories. In Java, directories are also treated as files. If one wishes to deal with the directories, then the list () method can be used. When the list () method is invoked by the directory object (created through File class constructor), then the list of other files and directories are extracted from it. This method is overloaded. One of them is String [] list ().
- A stream means a channel or a pipe. Like flow of water in a pipe, data flows from the source to the destination through the channels in Java.
- Java provides two types of streams: input streams and output streams.
- High-level input streams take their input from other input streams, whereas high level output streams direct their output to other output streams.
- High-level input streams are: `BufferedInputStream`, `DataInputStream` and `ObjectInputStream`, etc. High-level output streams are: `BufferedOutputStream`, `DataOutputStream`, `ObjectOutputStream`, `PrintStream`, etc.
- The network systems comprises a server, client and communication media.
- A machine running a process that sends request for the services is known as client. On the other hand, a machine running a process that responds to the client's request by offering requested services is known as server.
- The three most commonly used protocols within the TCP/IP suite are IP, TCP and UDP.
- TCP is a trustworthy and connection oriented protocol that permits the data that originates from source machine to be delivered without error to the destination. TCP sets up a connection between the source machine and the destination machine by transmitting control information before initiating the communication. This mechanism is known as handshake.
- The UDP (User Datagram Protocol) is considered as an untrustworthy and connectionless protocol which enables application to send independent, self-contained messages known as datagrams over the network.
- Every machine on the Internet is identified by a numerical address known as IP address.
- The grouping of IP address and port number is acknowledged as socket. A socket identifies an endpoint of a two way communication link between two programs running on the network.
- Domain name is very difficult to remember the IP address to connect to the Internet. The Domain Name System (DNS) is used to overcome this problem. DNS maps one particular IP address to a string of characters, which is popularly known as domain name.
- The `InetAddress` class is used to resolve the domain names to their IP addresses and vice versa.

- If any error occurs while creating the datagram socket, `SocketException` is thrown.
- An applet is a dynamic and interactive program that can run inside a web page displayed by a Java-capable browser, such as a HotJava Browser or an Internet Explorer browser, which are World Wide Web (WWW) browsers used to view web pages. An applet is a class present in a `java.applet` package.
- A special HTML tag is embedded in an applet to make it run on the web browser. The applet viewer application, present in the `jdk`, is used to run and check the applets. An applet has added advantages, such as frame, event-handling facility, graphics context and surrounding user interfaces.
- The applet class is a member of the Java Application Programming Interface (API) package, `Java.applet`. The applet class is used for creating a Java program that displays an applet.
- Java 1.1 event classes encapsulate all types of events occurring in the system. A successful handling of the events requires in-depth understanding of these classes.
- `AudioClip` is an interface present in `java.applet` package. This interface is used to play an `AudioClip` in the background of an applet. This interface supports only `.au` extension file.
- The `Window` class is also a subclass of `Container`. A top-level window can be created using the `Window` class. The top-level window is placed directly on the desktop and is not contained within any other object. Generally, the objects of the `Window` class are not directly created, rather, the `Frame` class, a subclass of `Window` is used.
- The `ActionListener` interface declares a method that is called when an action event is generated. The method defined in `ActionListener` interface is a single method, `void actionPerformed(ActionEvent ae)`.
- The `addMouseListener ()` method is used for monitoring an AWT object for the occurrence of a mouse event.
- A line can be drawn using the `drawLine ()` method of the `Graphics` class.
- The `Color` class provides various methods to use any color you want in the display. It defines the various color constants which can be directly used only by specifying the color of your choice. In addition, the `Color` class allows creation of millions of colors. The `Color` class contains three primitive colors namely, red, blue and green and all other colors are a combination of these three colors.
- The `Font` class is used to apply different font styles to the text. To select or apply a new font, a font object should to be constructed.
- AWT controls are components which allow a user to interact with the application. Some of the AWT controls are labels, buttons, checkbox, choice, list, text field, text area, scroll bar, etc.

## NOTES

## NOTES

- Scrollbars are horizontally or vertically oriented bars which allow the user to select items between a specified minimum and maximum values.
- A layout manager determines the dimension and exact positioning of components within a container. For example, window is a container that may contain components, such as buttons, labels and text fields.
- A `MenuBar` represents a list of menus which can be added to the top of a top level window.
- A `MenuBar` is represented by the object of class `MenuBar` in which only the default constructor is defined. A `MenuBar` may consist of one or more menus represented by the objects of the class `Menu`. A menu includes a list of menu items represented by the objects of the class `MenuItem`.

---

## 4.10 KEY TERMS

---

- **File class:** A programmer has to create the File object through the constructors provided by the File class.
- **Directory:** A directory is a collection of files and directories.
- **Stream:** In Java I/O (Input/Output) streams are flow of data you can either read from, or write to. Streams are typically linked to a data source, or data destination, like a file, network connection, etc.
- **TCP/IP:** It is a reliable and connection oriented protocol that allows the data that originates from a source machine to be delivered without error to the destination.
- **IP Address:** A numerical address by which every machine on the Internet is identified.
- **InetAddress Class:** A class used in Java to resolve domain names to their IP addresses and vice versa.
- **Applet:** A Java program that is compiled on one computer and can be run on other computers through Java-enabled web browsers or Java tools, such as applet viewer.
- **Event source:** It is an object that produces a certain type of event and provides methods to either add or remove listeners from the registered list of listeners.
- **AWT:** Refers to Abstract Window Toolkit and is responsible for communicating actions between the program and the user.
- **Action listener:** It is an interface that is used to declare a method that is called when an action event is generated.
- **Font class:** The `Font` class is used to apply different font styles to the text. To select or apply a new font, a font object should be constructed.
- **Layout classes:** Classes used for arranging, positioning and determining the shape and size of the various components held by the container.
- **CardLayout:** Used to manage a large number of components and organizes the components in layers.



---

## 4.11 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

*Input/Output Classes,  
Networking, AWT Graphics  
and Text, Controls, Layouts  
and Menus*

### Short-Answer Questions

1. How will you rename a file?
2. Explain the term dataflow in stream.
3. Define the term `SequenceInputStream`.
4. How a file is append?
5. Differentiate between `DataInputStream` and `DataOutputStream`.
6. What do you understand by the term `PrintStream` and `StreamTokenizer`.
7. What is network?
8. Why do you need sockets in networking?
9. State about the `InetAddress` Class.
10. How is an applet added to an HTML file?
11. Write the advantage and disadvantage of Java Applet.
12. What is the difference between an event source and an event listener?
13. How will you play an `AudioClip`?
14. Define the term AWT Class.
15. Write the definition of the term frame window.
16. What do you understand by the term drawing arcs?
17. What is the difference between `Choice` and `List`?
18. What is dialog class.

### Long-Answer Questions

1. Briefly explain the file and file name method giving appropriate example programs.
2. How will you create directory? Explain with the help of example program.
3. Discuss about the stream and hierarchy of stream class with the help of relevant examples.
4. Describe the `FileInputStream` and `FileOutputStream` with the help of example programs.
5. Explain the `ByteArrayInputStream` and `ByteArrayOutputStream` giving appropriate examples.
6. Differentiate between `BufferedInputStream` and `BufferedOutputStream`.
7. Explain the difference between TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) protocols.

### NOTES

## NOTES

8. Write the steps involved in creating TCP Server/Client programs by using socket.
9. Discuss briefly the networking classes and interfaces.
10. Describe the datagram packet network giving appropriate example programs.
11. Analyse the Applet life cycle and its method with the help of diagram.
12. Briefly explain the simple banner Applet with the help of example program.
13. List and describe the classes and interfaces which support event handling in Java.
14. Discuss about the window fundamentals and working with frame window with the help of diagram.
15. Analyse the event handling in frame window giving appropriate example program.
16. Write a program an applet code to demonstrate the use of various methods of graphics class.
17. Briefly explain the working of color and fonts with the help of example program.
18. What do you mean by control? Explain the different controls provided by AWT.
19. Describe the AWT layout managers with the help of example program.
20. Briefly explain the menus and dialog class giving appropriate example program.

---

## 4.12 FURTHER READING

---

- Balagurusamy, E. 2007. *Programming with Java*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Naughton, Patrick and Herbert Schidt. 1999. *Java 2: The Complete Reference*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Das, Rashmi Kanta. 2013. *Core Java for Beginners*, 3rd Edition. New Delhi: Vikas Publishing House Pvt. Ltd.
- Schildt, Herbert. 2006. *Java: The Complete Reference*, 7th Edition. New Delhi: Tata McGraw-Hill.
- Hunter, Jason and William Crawford. 2001. *Java Servlet Programming*, 2nd Edition. California: O'Reilly Media.
- Arnold, Ken, James Gosling and David Holmes. 2005. *The Java Programming Language*, 4th Edition. Boston: Addison-Wesley.
- Wigglesworth, Joe and Paula Lumby. 1999. *Java Programming Advanced Topics*, 2 Edition. Boston: Course Technology.
- Deitel, Paul and Harvey Deitel. 2011. *Java: How to Program*, 9th Edition. New Delhi: Prentice-Hall of India.

---

# UNIT 5 IMAGES, JDBC, JAVA BEANS, SERVLET API AND CORBA CONNECTIVITY

---

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

## NOTES

### Structure

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Images in Java
  - 5.2.1 File Formats
  - 5.2.2 Image Fundamentals
  - 5.2.3 ImageObserver and MediaTracker
- 5.3 JDBC: An Introduction
- 5.4 Swings
  - 5.4.1 Components of Swing
- 5.5 Java Beans
  - 5.5.1 What is Java Beans?
  - 5.5.2 JAR Files and Introspection
- 5.6 Basic Servlet API
  - 5.6.1 MIME Content Types
- 5.7 CORBA Connectivity in Java
  - 5.7.1 Working CORBA System
  - 5.7.2 Simple CORBA Service
- 5.8 Answers to 'Check Your Progress'
- 5.9 Summary
- 5.10 Key Terms
- 5.11 Self Assessment Questions and Exercises
- 5.12 Further Reading

---

## 5.0 INTRODUCTION

---

The Image class is used to load and display images. To load an image the `getImage ()` method of the Image class is used and to display the image the `drawImage ()` method of the Graphics class is used. JDBC (Java DataBase Connectivity) defines an API (Application Programming Interface) designed to support basic SQL (Structured Query Language) functionality independent of any specific SQL implementation. This means the focus is on executing SQL statements and retrieving their results. JDBC is an international standard for programming access to SQL databases. It was developed by JavaSoft, a subsidiary of Sun Microsystems. Relational Database Management System supports SQL. As we know that Java is platform independent, so JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

A Java Bean is a specially constructed java class written in the Java and coded according to the JavaBeans API specifications. A bean encapsulates many objects into one object, so we can access this object from multiple places. Moreover, it provides the easy maintenance. JavaBeans can also be referred to as

## NOTES

Java classes which adhere to an extremely simple coding convention. The Java Servlet API is not included in the core Java framework and is Standard Java Extension API. It comprises two packages `javax.servlet` and

`javax.servlet.http`, which contain classes and interfaces that are used to create servlets. Servlets are platform independent and can work with almost all the Web servers. They can be executed on any Web server that supports the servlet API. The Common Object Request Broker Architecture (CORBA) is a specification that helps to integrate heterogeneous systems that have different hardware, operating systems, networks and different programming languages.

In this unit, you will study about the image in Java, file formats, image fundamentals, image observer and mediatracker, JDBC an introduction, register drive, establish a session, `ResultSet`, closing the session, swings, JAPPLET, Java beans, application builder tools, Bean Development Kit (BDK), JAR files and introspection, bean info interface, constrained properties, persistence and customisers, basic servlet API, Get and Post method, MIME context types, and CORBA connectivity in Java, working CORBA system, CORBA servers, CORBA clients, simple CORBA service, application and CORBA.

---

## 5.1 OBJECTIVES

---

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

- Understand the image in Java
- Explain the file formats
- Analyse the image fundamentals
- Elaborate on the image observer and mediatracker
- Discuss the basic concept of JDBC and register drive
- Establish a session and `ResultSet`
- Describe the closing of the session
- Understand the swings and JAPPLET
- Define the Java beans and application builder tools
- Analyse the Bean Development Kit (BDK)
- Explain the JAR files and introspection
- Describe the bean info interface and constrained properties
- Discuss the persistence and customisers
- Understand the basic servlet API and Get and Post method
- Define the MIME context types
- Analyse the CORBA connectivity in Java
- Explain the working of CORBA system and CORBA servers
- Define the CORBA clients and simple CORBA service
- Describe the application of CORBA

---

## 5.2 IMAGES IN JAVA

---

The **java.awt** package contains an abstract class, **Image**. This class provides the mechanism to handle operations that are related to an image. The signature for the **Image** class is: `public abstract class java.awt.Image extends java.lang.Object`

The **Image** class cannot be instantiated. Generally, the **Image** class is used to draw an image on the applet. **Image** class supports only two types of **image** files:

- JPEG Format (Joint Photographic Expert Group Format)
- GIF (Graphic Interchange Format)

The **Image** class also provides some constants which are used to set the scale for the **image**. These are:

- `public static final int SCALE_DEFAULT`
- `public static final int SCALE_FAST`
- `public static final int SCALE_SMOOTH`
- `public static final int SCALE_REPLICATE`
- `public static final int SCALE_AREA_AVERAGING`

One can create an empty image by the help of a predefined method of Component class. The method is:

```
public Image createImage(int imgwidth, int imgheight)
```

Example: `Image img1=createImage(200,200);`

One can draw an **image** on the screen, by the help of one predefined method of Graphics class. The method is:

```
public void drawImage(Image img, int i, int j, ImageObserver imob)
```

One can retrieve the **image** from a desired location by the help of methods, which are present in both the Applet and Toolkit class. The methods are:

- `public Image getImage(URL url)`
- `public Image getImage(URL url, String imgpath)`

### How to Draw Image in Frame?

#### Program 5.1

```
import java.awt.*;
public class ImageDemo extends Canvas
{
 public ImageDemo()
 {
 setSize(300,300);
 setBackground(Color.cyan);
 }
 public static void main(String args[])
 {
 ImageDemo id=new ImageDemo();
 Frame f=new Frame("ImageDemo");
 f.setSize(300,300);
 }
}
```

### NOTES

## NOTES

```
f.add(id);
f.setVisible(true);
}
public void paint(Graphics g)
{
 Image im=Toolkit.getDefaultToolkit().getImage("Water
lilies.jpg");
 g.drawImage(im,30,40,this);
}
}
```

This example illustrates the use of Image class by using `getImage()` method of Toolkit class. Toolkit is a predefined class present in `java.awt` package. This is an abstract class so it can't be instantiated. If the programmer wants to instantiate the Toolkit class then the programmer has to call a static method of Toolkit class:

```
public static Toolkit getDefaultToolkit()
```

In the above example we create the object of image class by this method. After that the programmer extracts the image by the `getImage()` method of Toolkit class. The programmer draws the image in the frame by `drawImage()` method of Graphics class. Finally, the drawing is reflected on the frame by the help of `paint()` method. Hence the output is as shown.

### Output of the program:



### Program 5.2

```
import java.awt.*;
import java.applet.*;
public class Image123 extends Applet
{
 Image img;
 public void init()
 {
 img=getImage(getDocumentBase(),"CAKE.jpg");
 }
 public void paint(Graphics gph)
 {
 gph.drawImage(img,30,50,this);
 }
}
```

```
/*<applet code="Image123" width=300 height=300>
</applet>*/
```

*Images, JDBC, Java  
Beans, Servlet API and  
CORBA Connectivity*

### Output of the program:



### NOTES

This example is also an illustration of the use of Image class. But, here the object of the image class is not got by the help of the `createImage()` method of Component class. Rather, the abstract Image class object is created by invoking the `getImage()` method of the Applet class, which takes the path and name of an existing image as parameter. The object created by this method refers to the specified image. The image is reflected on the applet screen by the help of the `paint()` method.

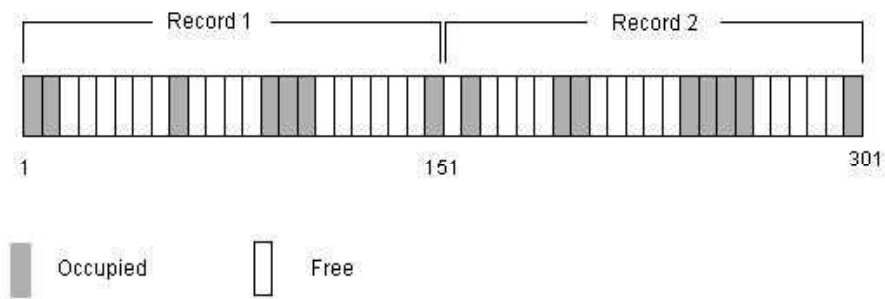
### 5.2.1 File Formats

There are two approaches to organize records in a file. In the first approach, all the records of a file are of a fixed-length. However, in the second approach, the records of a file vary in size. A file of a fixed-length records is simple to implement as all the records are of fixed length. However, deletion of record results in fragmented memory. On the other hand, in case of files of variable-length records, the memory space is efficiently utilised. However, locating the start and end of record is not simple.

#### Fixed-Length Records

All the records in a file of fixed-length records are of the same length. In a file of fixed-length records, every record consists of the same number of fields and the size of each field is fixed for every record. It ensures an easy location of field values, as their positions are predetermined. Since each record occupies equal memory, as shown in Figure 5.1, identifying the start and the end of the record is relatively simple.

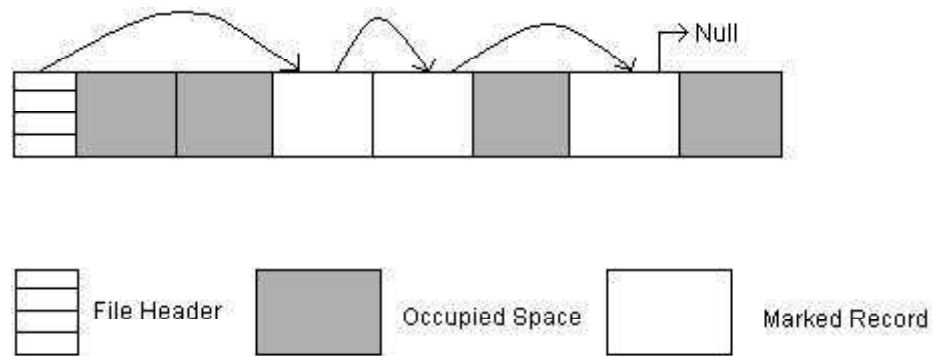
**NOTES**



**Fig. 5.1** Location of Fixed-Length Records

A major drawback of fixed-length records is that a lot of memory space is wasted. Since a record may contain some optional fields and space is reserved for optional fields as well—it stores null value if no value is supplied by the user for that field. Thus, if certain records do not have values for all the fields, the memory space is wasted. In addition, it is difficult to delete a record as deletion of a record leaves blank spaces in between the two records. To fill up that blank space, all the records following the deleted record need to be shifted.

It is undesirable to shift a large number of records to fill up the space freed by a deleted record, since it requires additional disk access. Alternatively, the space can be reused by placing a new record at the time of insertion of new records, since insertions tend to be more frequent. However, there must be some way to mark the deleted records so that they can be ignored during the file scan. In addition to a simple marker on the deleted record, some additional structure is needed to keep track of the free space created by the deleted or marked records. Thus, certain number of bytes is reserved in the beginning of the file for a **file header**. The file header stores the address of the first marked record, which further points to the second marked record and so on. As a result, a linked list of marked slots is formed, which is commonly termed as a free list. Figure 5.2 shows the record of a file with the file header pointing to the first marked record and so on. A new record is placed at the address pointed by the file header and the header pointer is changed to point towards the next available marked record. In case no marked record is available, the new record is appended at the end of the file.



**Fig. 5.2** Fixed-Length Records with Free List of Marked Records



## Variable-Length Records

Variable-length records may be used to utilise the memory more efficiently. In this approach, the exact length of the field is not fixed in advance. Thus, to determine the start and end of each field within the record, special separator characters, which do not appear anywhere within the field value, are required (Refer Figure 5.1). Locating any field within the record requires a scan of record until the field is found.

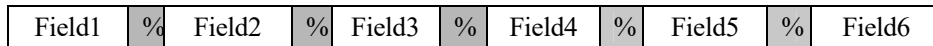


Fig. 5.3 Organization of Variable-Length Records with '%' Delimiter

Alternatively, an array of integer offsets could be used to indicate the starting address of fields within a record. The  $i^{\text{th}}$  element of this array is the starting address of the  $i^{\text{th}}$  field value relative to the start of the record. An offset to the end of the record is also stored in this array, which is used to recognise the end of the last field. The organization is shown in Figure 5.4. For null value, the pointer to starting and end of the field is set same. That is, no space is used to represent a null value. This technique is a more efficient way to organize the variable-length records. Handling such an offset array is an extra overhead; however, it facilitates direct access to any field of the records.

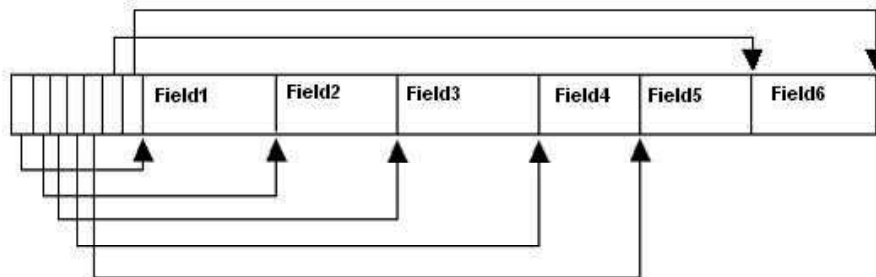


Fig. 5.4 Variable-Length Record Organization Using an Array of Field Offsets

Sometimes there may be a possibility that the values for a large number of fields are not available or are null. In that case, we can store the sequence of pair <field name, field value> instead of just field values in each record. In Figure 5.5, three separator characters are used—one for separating two fields, second one for separating field value from field name, and third one for separating two records.

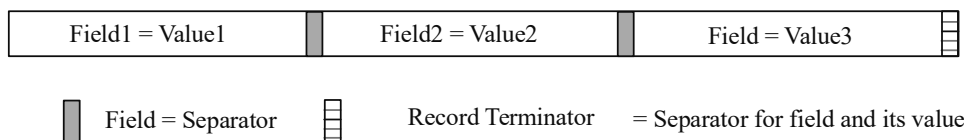


Fig. 9.5 Organization for Variable-Length Record

It is clear from the discussion that the memory is utilised efficiently but processing the variable-length records require a complicated program. Moreover, modifying the value of any field might require shifting of all the following fields, since the new value may occupy more or less space than the space occupied by the existing value.

## NOTES

## NOTES

### 5.2.2 Image Fundamentals

The `Image` class is used to load and display images. To load an image the `getImage()` method of the `Image` class is used and to display the image the `drawImage()` method of the `Graphics` class is used. Example 5.1 illustrates loading and viewing of image.

The general form of the `drawImage()` method is as follows:

```
boolean drawImage (Image image, int startx, int starty, int width, int
height, ImageObserver img_obj)
```

where,

`image` is the image to be loaded in the applet.

`startx` is the pixels space from the left corner of the screen.

`starty` is the pixels space from the upper corner of the screen.

`width` is the width of the image.

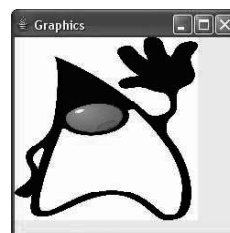
`height` is the height of the image.

`img_obj` is the object of the class that implements `ImageObserver` interface.

**Example 5.1:** A program to demonstrate loading and viewing of image is as follows:

```
import java.awt.*;
import javax.swing.*;
public class ImageExample extends JPanel
{
 public void paintComponent (Graphics gra)
 {
 super.paintComponent (gra);
 Graphics2D gra2D = (Graphics2D) gra;
 Image pic = new ImageIcon ("java.gif").getImage ();
 gra2D.drawImage (pic, 0, 0, 200, 200, null);
 }
 public static void main (String [] args)
 {
 ImageExample ge = new ImageExample ();
 JFrame fr = new JFrame ("Graphics");
 fr.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
 fr.add (ge);
 fr.setSize (250, 250);
 fr.setLocationRelativeTo (null);
 fr.setVisible (true);
 }
}
```

**The output of the programs is:** illustrated in the following screenshot:



### 5.2.3 ImageObserver and MediaTracker

Since Java programming has its roots in the Internet applications, such as web browsers, therefore its image handling APIs (Application Programming Interface) were typically designed because the images may take time to load over a slow network, providing for detailed information about image-loading progress. In Java, the Swing toolkit adds its specific layer of image handling, such as `ImageIcon`, which encapsulates an image source for the user.

The `Image` class and the `java.awt.image` package provide support for imaging, i.e., the display and manipulation of graphical images. An **image** is simply a graphical object and also the key component of web design. Java allows images to be managed under Java program control and hence Java provides extensive support for imaging.

Images are objects of the `Image` class, which is part of the `java.awt` package. Images are manipulated using the classes predefined in the `java.awt.image` package.

Principally, Java uses the following interfaces:

```
ImageConsumer
ImageObserver
ImageProducer
```

The `MediaTracker` class is also part of `java.awt`.

Here we will discuss about the `ImageObserver` interface and `MediaTracker` class.

#### 1. ImageObserver

All operations on image data, such as loading, drawing, scaling, etc., allow the user to specify an **'Image Observer' object** as a member. An image observer implements the `ImageObserver` interface, which allows it to obtain notification by means of information about the image that is available. Principally, the image observer is referred as a *callback* which is notified progressively as and when the image is loaded. For a static image, such as a GIF (Graphics Interchange Format) or JPEG (Joint Photographic Experts Group) data file, the user is notified as chunks of image data arrive and also when the entire image is complete. In addition, for a video source or animation (e.g., GIF89), the image observer is notified at the end of each frame as the continuous stream of pixel data is generated.

The image observer can do required modifications and necessary changes using this information. To use an image observer, implement the `imageUpdate()` method, which is defined by the `java.awt.image.ImageObserver` interface. Following is the syntax for `imageUpdate()` method:

```
public boolean imageUpdate(Image image, int flags, int x, int
y, int width, int height)
```

When required the `imageUpdate()` is called by the graphics system for passing the user information regarding the structure and construction of the images. The image parameter holds a reference to the `Image` object. The

## NOTES

## NOTES

`flags` is referred as an integer whose bits specify what information regarding the image is currently available. The flag values are defined as `static` variables in the `ImageObserver` interface. The following example illustrates this concept:

### Program 5.3

```
//file: ObserveImageLoad.java
import java.awt.*;
import java.awt.image.*;

public class ObserveImageLoad {

 public static void main(String [] args)
 {
 ImageObserver myObserver = new ImageObserver() {
 public boolean imageUpdate(
 Image image, int flags, int x, int y, int width, int
 height)
 {
 if ((flags & HEIGHT) !=0)
 System.out.println("Image height = " + height);
 if ((flags & WIDTH) !=0)
 System.out.println("Image width = " + width);
 if ((flags & FRAMEBITS) != 0)
 System.out.println("Another frame finished.");
 if ((flags & SOMEBITS) != 0)
 System.out.println("Image section : "
 + new Rectangle(x, y, width, height));
 if ((flags & ALLBITS) != 0)
 System.out.println("Image finished!");
 if ((flags & ABORT) != 0)
 System.out.println("Image load aborted...");
 return true;
 }
 };

 Toolkit toolkit = Toolkit.getDefaultToolkit();
 Image img = toolkit.getImage(args[0]);
 toolkit.prepareImage(img, -1, -1, myObserver);
 }
}
```

When you run the example to provide an image file as the command line argument then you will find numerous incremental messages by the Java compiler regarding loading the image.

The `flags` integer determines which of the other parameters, the `x`, `y`, `width`, and `height` hold valid data and what that data means. To test whether a particular flag in the `flags` integer is set, we use the `&` (AND) operator. The `width` and `height` parameters have a dual role. If `SOMEBITS` is set, then they represent the size of the chunk of the image that has just been provided. The `HEIGHT` or `WIDTH` is set because they represent the overall image dimensions. Finally, `imageUpdate()` returns a `Boolean` value indicating whether or not it is required for the future updates.

In the above given Java example, once the `Image` object is defined with `getImage()`, the loading process starts with the `Toolkit`'s

prepareImage () method, which considers the image observer as an argument. Using an Image API method, such as drawImage (), scaleImage (), or requesting for image dimensions with getWidth () or getHeight () will be sufficient to start the operation.

Remember that even though the getImage () method can create the image object, but it does not load the required data until one of the image operations needs it.

The above Java example demonstrates the lowest-level general mechanism to start and monitor the process of loading image data.

## 2. MediaTracker

The java.awt.MediaTracker class is a general utility that tracks the loading of a number of images or other media types for the users.

Basically, the java.awt.MediaTracker is a utility class that simplifies the problems when the user have to wait for one or more images to be loaded completely before they are actually displayed. A MediaTracker monitors the loading of an image or a group of images and helps the user to check those either periodically or to wait until the loading is completed. MediaTracker implements the ImageObserver interface which allows it to receive image updates.

The following example code illustrates how the MediaTracker is used while an image is being prepared.

### Program 5.4

```
//file: StatusImage.java
import java.awt.*;
import javax.swing.*;

public class StatusImage extends JComponent
{
 boolean loaded = false;
 String message = "Loading...";
 Image image;

 public StatusImage(Image image) { this.image = image; }
 public void paint(Graphics g) {
 if (loaded)
 g.drawImage(image, 0, 0, this);
 else {
 g.drawRect(0, 0, getSize().width - 1, getSize().height -
1);
 g.drawString(message, 20, 20);
 }
 }
 public void loaded() {
 loaded = true;
 repaint();
 }
 public void setMessage(String msg) {
 message = msg;
 repaint();
 }
}
```

## NOTES

## NOTES

```
 }

 public static void main(String [] args) {
 JFrame frame = new JFrame("TrackImage");
 Image image = Toolkit.getDefaultToolkit().getImage(args[0]
);
 StatusImage statusImage = new StatusImage(image);
 frame.add(statusImage);
 frame.setSize(300, 300);
 frame.setVisible(true);

 MediaTracker tracker = new MediaTracker(statusImage);
 int MAIN_IMAGE = 0;
 tracker.addImage(image, MAIN_IMAGE);
 try {
 tracker.waitForID(MAIN_IMAGE); }
 catch (InterruptedException e) {}
 if (tracker.isErrorID(MAIN_IMAGE))
 statusImage.setMessage("Error");
 else
 statusImage.loaded();
 }
}
```

In the above example, a trivial component called `StatusImage` is created that accepts an image and draws a text status message until it is told that the image is loaded. It then displays the image. Remember that a `MediaTracker` is used to load the image data.

The user first creates a `MediaTracker` to manage the image. The `MediaTracker` constructor takes a `Component` as an argument which is the component onto which the image is drawn. When the user do not have the component reference accessible, then the user can simply substitute a generic component reference as shown below:

```
Component comp = new Component();
```

After creating the `MediaTracker`, the user assigns it images to manage. Each image is associated with an integer that identifier the user can use later for checking on its status or to wait for its completion. Multiple images can be associated with the same identifier, letting the user manage those as a group. The value of the identifier is also meant to prioritize loading when waiting on multiple sets of images; lower IDs have higher priority. To manage only a single image, one identifier is created which is called `MAIN_IMAGE` and passed it as the ID for our image in the call to `addImage()`.

Then the `MediaTracker` `waitForID()` routine is called, which blocks on the image, waiting for it to finish loading. Another `MediaTracker` method is `waitForAll()`, which waits for all images to be completed, not just a single ID. It is possible that the loading can be interrupted by an `InterruptedException`. The user should test for errors during image preparation with `isErrorID()`.

The `MediaTracker` `checkID()` and `checkAll()` methods may be used to check periodically the status of images loading, returning `true` or `false` to indicate whether loading is finished. The `checkAll()` method does this for the union of all images being loaded. Additionally, the `statusID()` and `statusAll()` methods return a constant indicating the status or final condition of an image loading. The value is one of the `MediaTracker` constant values: `LOADING`, `ABORTED`, `ERROR`, or `COMPLETE`. For `statusAll()`, the value is the bitwise OR value of all of the various statuses.

## NOTES

---

### 5.3 JDBC: AN INTRODUCTION

---

JDBC (Java DataBase Connectivity) defines an API (Application Program Interface) designed to support basic SQL (Structured Query Language) functionality independent of any specific SQL implementation. This means the focus is on executing SQL statements and retrieving their results.

JDBC is an international standard for programming access to SQL databases. It was developed by **JavaSoft**, a subsidiary of **Sun Microsystems**.

Relational Database Management System (RDBMs) supports SQL. As we know that Java is platform independent, so JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

Java Database Connectivity is similar to Open DataBase Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not depended upon any language.

In short JDBC helps the programmers to write Java applications that manage these three programming activities:

- Establishing a connection with a database or other tabular data source.
- Sending SQL commands to the database.
- Processing the results.

#### What is API?

API is the abbreviation of *Application Program Interface*, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together. Simply, it provides a set of rules for performing a particular task but in case of JDBC API the task is connect to the database. They are completely invisible to users and Web surfers. Their primary role is to provide a channel for applications to work with each other.

#### Components of JDBC

The following are the components of JDBC:

## NOTES

### JDBC API

The JDBC application programming interface provides the facility for accessing the relational database from the Java programming language. The API technology provides the industrial standard for independently connecting Java programming language and a wide range of databases. The user not only execute the SQL statements but can also access it anywhere within a network because of its “Write Once, Run Anywhere” (WORA) capabilities. Due to JDBC API technology, user can also access other tabular data sources like spreadsheets or flat files even in the heterogeneous environment.

The application programming interface is divided into two packages:

- **java.sql**
- **javax.sql**

### Driver Manager

The JDBC Driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver. Usually Driver Manager is the backbone of the JDBC architecture. It's very simple and small that is used to provide a means of managing the different types of JDBC database driver running on an application. The main responsibility of JDBC database driver is to load all the drivers found in the system properly as well as to select the most appropriate driver from opening a connection to a database. The Driver Manager also helps to select the most appropriate driver from the previously loaded drivers when a new open database is connected.

### JDBC Test Suite

The function of JDBC driver test suite is ensure whether the JDBC drivers will run user's program or not . The test suite of JDBC application program interface is very useful for testing a driver based on JDBC technology during testing period.

### Types of Driver

A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver accepts generic commands from a program and then translates them into specialized commands for the device.

### Type 1: JDBC-ODBC Bridge Driver

The JDBC Type-1 driver also known as the JDBC-ODBC bridge driver, is a database driver implementation that employs the ODBC driver connect to the database. The driver converts JDBC method calls into ODBC function calls.

The driver is platform dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system. Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver. The other implication is that any application using a Type 1 driver is non-portable given the binding between the driver and platform.



### Steps to Perform the Task

- Translates query obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver.
- Client -> JDBC Driver -> ODBC Driver -> Database.
- There is some overhead associated with the translation work to go from JDBC to ODBC.

### NOTES

### Advantages

- Almost any database, for which ODBC driver is installed, can be accessed.
- A Type 1 driver is easy to install.

### Disadvantages

- Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native database connectivity interface.
- The ODBC driver needs to be installed on the client machine.
- Considering the client-side software needed, this might not be suitable for applets.
- Will not be suitable for Java applets.
- It is not platform independent because of ODBC.

Architecture:

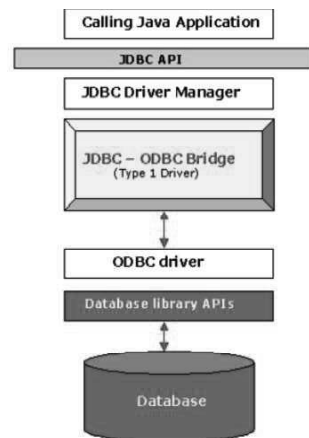


Fig. 5.6 Type 1 Driver

The above is the architecture of Type 1 driver.

### Program 5.5 (Type-1 Driver)

```
import java.sql.*;
public class Type1
{
 public static void main (String args [])
 {
 try{
 Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
 Connection con=
 DriverManager.getConnection ("jdbc:odbc:omm",
 "scott","tiger");
```

## NOTES

```
Statement st=con.createStatement();
ResultSet rs=st.executeQuery("select * from emp");
while(rs.next())
{
String name=rs.getString("ename");
int salary=rs.getInt("sal");
System.out.println("Employee Name is: "+name+" and
"+"Salary is: "+salary);
}
} catch (Exception e1)
{
System.out.println(e1.getMessage());
}
}
```

### Type 2: Native-API/Partly Java Driver

The JDBC Type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

The Type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the final database calls. The driver is compiled for use with the particular operating system.

However the Type 2 driver provides more functionality and better performance than the Type 1 driver as it does not have the overhead of the additional ODBC function calls.

#### Advantages

Better performance than the Type 1 driver as it does not have the overhead of the additional ODBC function calls.

#### Disadvantages

- The vendor client library needs to be installed on the client machine.
- Cannot be used in web-based application due the client side software needed.
- Not all databases have a client side library.
- This driver is platform dependent.

#### Architecture:

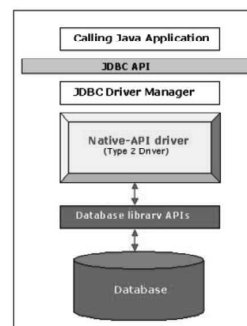


Fig. 5.7 Type 2 Driver

The above is the architecture of Type 2 driver.

### Program 5.6 (Type-2 Driver)

```
import java.sql.*;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
public class Type2
{
 public static void main (String[] args)
 {
 try{
 String datasource="ds1";
 String pool="pool1";
 Properties p=new Properties ();

 p.put (Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.
 WLInitialContextFactory");
 p.put (Context.PROVIDER_URL,"t3://localhost:7001");
 InitialContext ctx=new InitialContext (p);
 DataSource source=(DataSource)ctx.lookup (datasource);
 Connection con=source.getConnection ();
 Statement st=con.createStatement ();
 ResultSet rs=st.executeQuery ("select * from emp");
 while (rs.next ())
 {
 String name=rs.getString ("ename");
 int salary=rs.getInt ("sal");
 System.out.println ("Employee Name is: "+name+"
 and "+ "Salary is: "+salary);
 }
 catch (Exception e) {
 }
 }
 }
}
```

### Type 3: All Java/Net-Protocol Driver

The JDBC Type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

This differs from the Type 4 driver in that the protocol conversion logic resides not in the middle-tier like Type 4 drivers. The Type 3 driver is written entirely in Java. The same driver can be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The Type 3 driver is platform-independent as the platform-related differences are taken care by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

## NOTES

## NOTES

### Steps to Perform the Task

- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- The JDBC Client driver written in Java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- Thus the client driver to middleware communication is database independent.
- Client -> JDBC Driver -> Middleware-Net Server -> Any Database.

### Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the vendor database library on the client machine. Also the client to middleware need not be changed for a new database.
- The Middleware Server (which can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing, etc.

For the above include JDBC driver features in Weblogic to perform the following:

- o Can be used in Internet since there is no client side software needed.
- o At client side a single driver can handle any database. It works provided the middleware supports that database!

### Disadvantages

- Requires database specific coding to be done in the middle tier.
- An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware services described above.

### Architecture

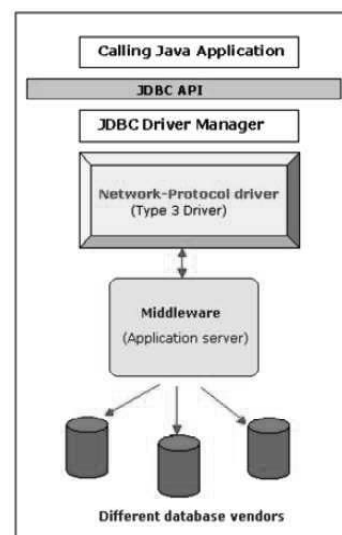


Fig. 5.8 Type 3 Driver

The above is the architecture of Type 2 driver.

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

### Program 5.7 (Type-3 Driver)

```
import java.sql.*;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
public class Type3
{
 public static void main (String[] args)
 {
 try{
 String datasource="ds3";
 String pool="pool3";
 Properties p=new Properties ();

 p.put (Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.
 WLInitialContextFactory");
 p.put (Context.PROVIDER_URL,"t3://localhost:7001");
 InitialContext ctx=new InitialContext (p);
 DataSource source=(DataSource) ctx.lookup (datasource);
 Connection con=source.getConnection ();
 Statement st=con.createStatement ();
 ResultSet rs=st.executeQuery ("select * from emp");
 while (rs.next ())
 {
 String name=rs.getString ("ename");
 int salary=rs.getInt ("sal");
 System.out.println ("Employee Name is: "+name+" and
 "+"Salary is:"+salary);
 }
 } catch (Exception e)
 {
 }
 }
}
```

### NOTES

### Type 4 Driver-Native-Protocol Driver

The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into the vendor-specific database protocol.

The Type 4 driver is written completely in Java and is hence platform independent. It is installed inside the Java Virtual Machine of the client. It provides better performance over the Type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the Type 3 drivers, it does not need associated software to work.

### Steps to Perform the Task

- Type 4 drivers are entirely written in Java that communicates directly with a vendor's database, usually through socket connections. No translation or middleware layers are required, improving performance.

## NOTES

- The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
- Completely implemented in Java to achieve platform independence.
- To include the widely used Oracle thin driver - oracle.jdbc.driver.OracleDriver which connect to jdbc:oracle:thin URL format.
- Client -> Native protocol JDBC Driver -> Database server.

### Advantages

- These drivers do not translate the requests into an intermediary format, (such as ODBC), nor do they need a middleware layer to service requests. Thus the performance may be considerably improved.
- All aspects of the application to database connection can be managed within the JVM; this can facilitate easier debugging.

### Disadvantage

- At client side, a separate driver is needed for each database.

### Architecture

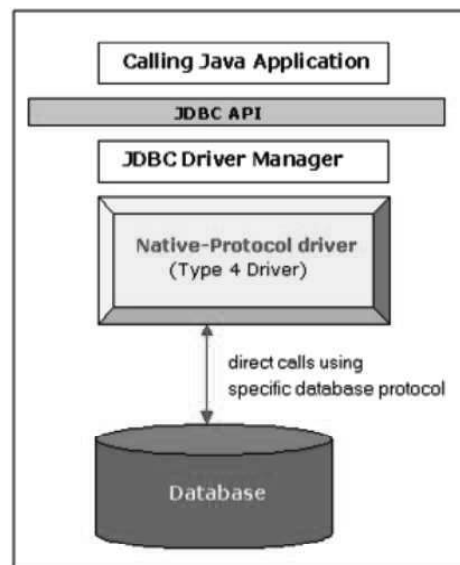


Fig. 5.9 Type 4 Driver

The above is the architecture of Type 4 driver.

### Program 5.8 (Type-4 Driver)

```
import java.sql.*;
class Customer
{
 public static void main (String args[]) throws SQLException
 {
 DriverManager.registerDriver (new
 oracle.jdbc.driver.OracleDriver ());
 System.out.println ("Connection to the database.....");
 try{
 Connection cn=DriverManager.getConnection
```

```
("jdbc:oracle:thin:@rashmi:1521:orc11","sai","sai");
 Statement st=cn.createStatement();
 ResultSet rs=st.executeQuery("select * from emp1");
 while(rs.next())
 {
 String s=rs.getString(1);
 System.out.println(s);
 }
 st.close();
 cn.close();
} catch (Exception ex)
{
 System.out.println("The exception raised is: "+ex);
}
}
```

## NOTES

In the URL, **thin** is the JDBC driver, **rashmi** is the database name, **1521** is the port on which the connection is to be established, and **orc11** is the system ID.

“sai” is the user id and “sai” is password.

### JDBC Basics

To store data in the database, you create tables. Later, when you learn about creating the tables used as examples, the tables will be in the default database. We purposely kept the size and number of tables small to keep things manageable.

Once you have created the database and tables to store the data, you'll open a connection with your DBMS. You also need to know some SQL code. After that, discover how easy it is to use JDBC to pass SQL statements to your DBMS and then process the results that are returned.

### Loading the Driver

We need to connect to the database, which the program uses to connect. Typically, a JDBC application connects to a target data source using one of two mechanisms:

**DriverManager:** The DriverManager class loads a specific Driver class by using the URL. That allows user to customize the JDBC Drivers used by user applications.

Loading the driver by the following code:

```
Class.forName("Database driver name");
```

**Class-** It is a class of **java.lang** package.

**forName()-** It is a static method of a class known as **Class**.

Responsibility of the method is to register a **.class** file at runtime.

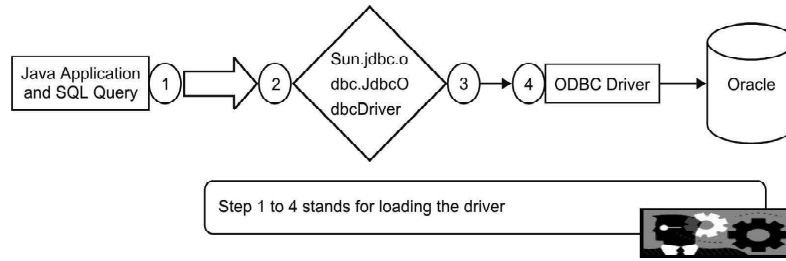
**Database driver name-** It is a class file which is loaded at runtime.

The **Class** file will vary from driver to driver.

For example, if it is Type-1 then

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

**NOTES**



*Fig. 5.10 Loading driver*

**Establishing a Connection**

The second step is to establish a connection with the appropriate driver to get connection with the required database.

We need to establish a connection through the following line of code:

```
Connection conn = DriverManager.getConnection("JDBC URL",
"User id", "Password");
```

**Connection**—It is an interface present in **java.sql** package. A connection is a session in a specific database engine. Here, connection is established at runtime.

**Conn**—It is a reference of Connection Interface.

**DriverManager**—It is a predefined class present in **java.sql** package.

**getConnection ()**—It is a predefined static method present in DriverManager class.

**JDBC URL**—It has three components as follows:

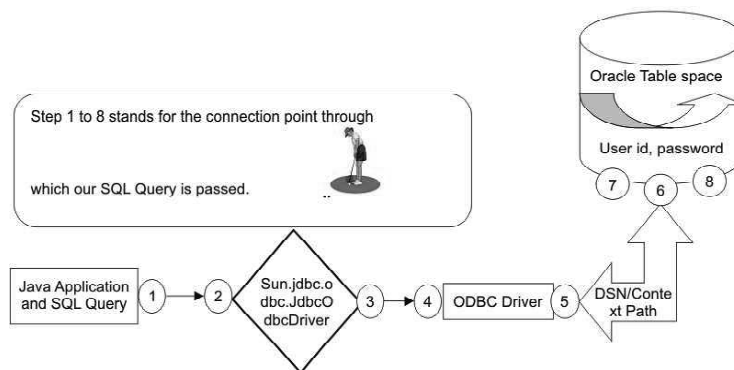
- Protocol
- Sub-Protocol
- Context Path

Altogether these three are known as JDBC URL.

**“Protocol: Subprotocol:ContextPath”**

**User id**—Here we have to put the concerned database user ID as required by the program.

**Password**—In this case we have to put the concerned database password which follows the user ID previously.



*Fig. 5.11 Establishing Connection*



## Creating the Statement Object

We can create the object by the following line of code:

```
Statement st = conn.createStatement ();
```

**Statement**—It is predefined interface present in java.sql package. This interface used to execute the SQL statements, by which we can implement DML, DDL operation (i.e., SELECT, CREATE, INSERT, UPDATE, DELETE, etc).

**conn**—Here conn is the object of Connection interface, which we got in earlier step.

**createStatement()**—It is the method of Connection interface.

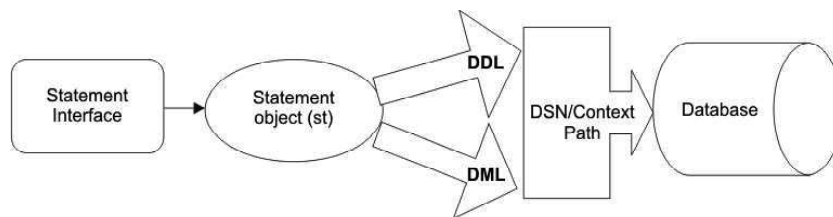


Fig. 30.7 Creating statement object

## Retrieving Values from ResultSet

It is an interface present in java.sql package. The number of rows returned in a result set can be zero or more. A user can access the data in a result set using a cursor one row at a time from top to bottom. The JDBC API supports a cursor to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

Methods present in the ResultSet

These are some methods to retrieve the value from the ResultSet:

- `next()`—Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.
- `previous()`—Moves the cursor backwards one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
- `first()`—Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
- `last()`—Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
- `beforeFirst()`—Positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.
- `afterLast()`—Positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.

## NOTES

- `relative(int rows)`—Moves the cursor relative to its current position.

We can get the PreparedStatement object by using the following line of code:

```
ResultSet rs = st.executeQuery("Select * from emp<any table name>");
```

## NOTES

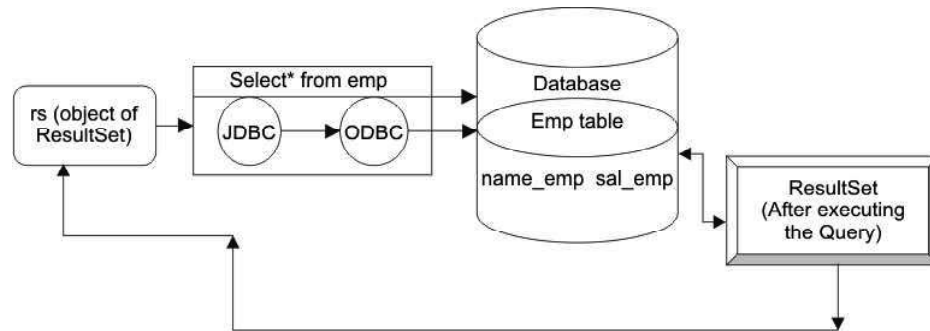
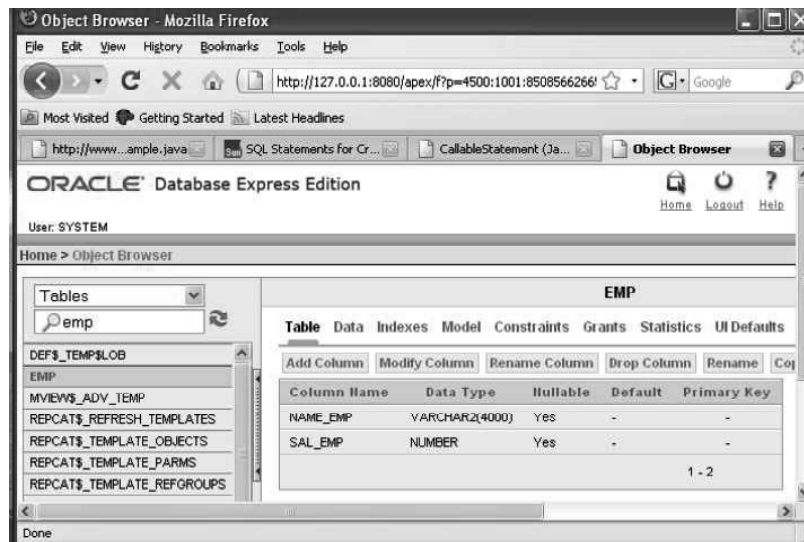


Fig. 5.13 Database Retrieval Result

### Example Using The Type-1 Driver

For this example, we need to create a table named as Emp having two fields (columns). One is name\_emp (VARCHAR2) and second one is sal\_emp (NUMBER). This table must contain some data.



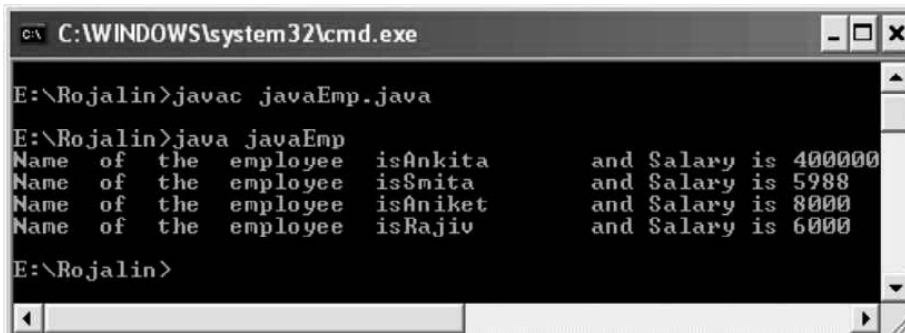
### Program 5.9

```
import java.sql.*;
public class javaEmp
{
 public static void main (String args []) throws Exception
 {
 Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
 Connection
 jdbcconn=DriverManager.getConnection ("jdbc:odbc:xyz", "system", "manager");
 Statement st=jdbcconn.createStatement ();
```

```
ResultSet rs=st.executeQuery("Select * fromEmp");
while(rs.next())
{
String s1=rs.getString("name_emp");
int s2=rs.getInt("sal_emp");
System.out.println("Name of the employee is"+s1+"\t"+"andSalary
is"+s2);
}}}
```

## NOTES

### Output of the program:

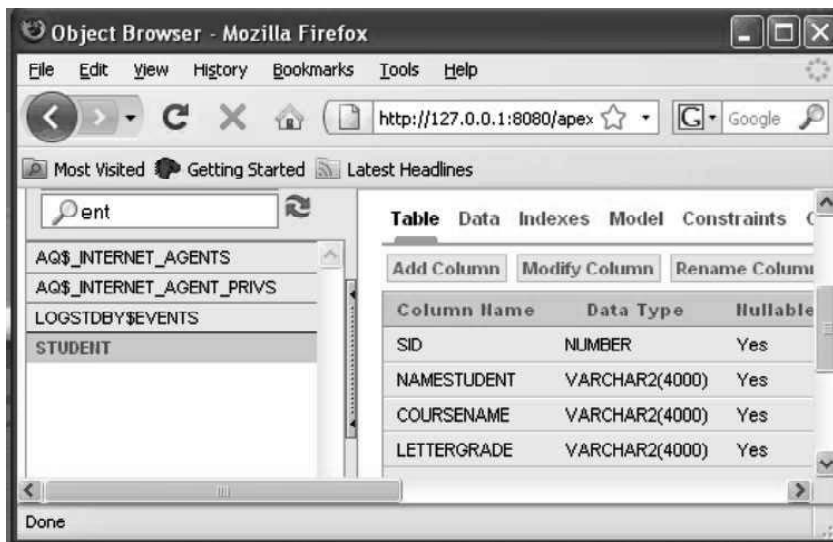


```
C:\WINDOWS\system32\cmd.exe
E:\Rojalin>javac javaEmp.java
E:\Rojalin>java javaEmp
Name of the employee isAnkita and Salary is 400000
Name of the employee isSmita and Salary is 5988
Name of the employee isAniket and Salary is 8000
Name of the employee isRajiv and Salary is 6000
E:\Rojalin>
```

### For Inserting the Data in the Table

For this example we require one Student table, where we need to insert data through the Java application. The following fields are in the Student table: sid(NUMBER), namestudent (VARCHAR2), course name (VARCHAR2), lettergrade (VARCHAR2).

### Output of the program:



### Java Application for this Example (InsertionClass.java)

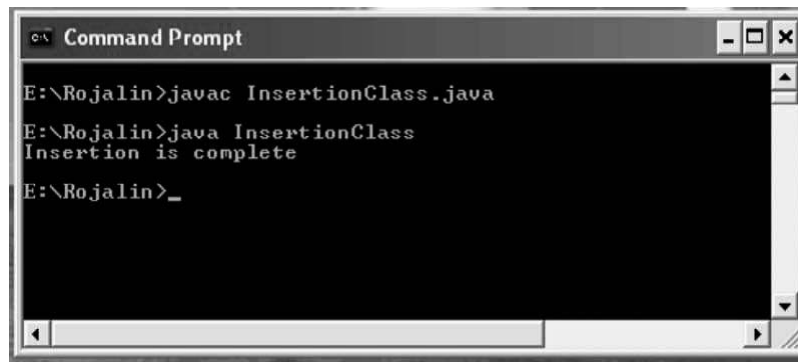
#### Program 5.10

```
import java.sql.*;
import java.io.*;
public class InsertionClass {
public static void main (String[] args) throws Exception
{
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Connection connection =
```

## NOTES

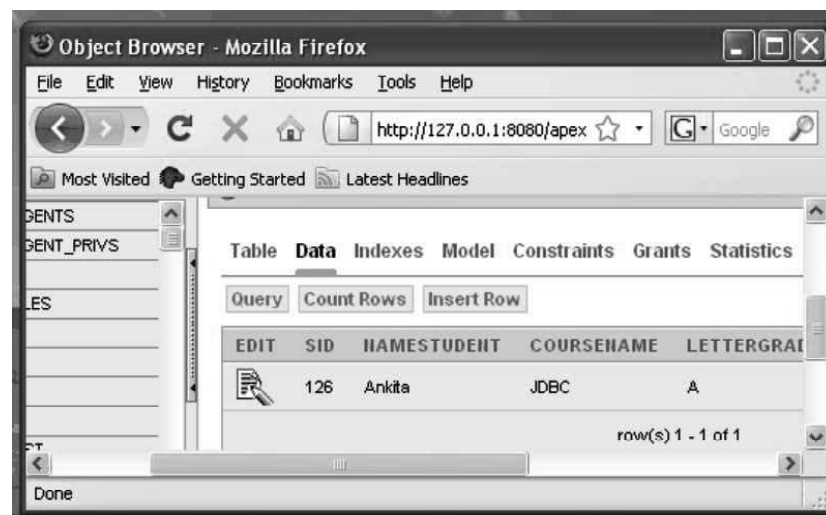
```
DriverManager.getConnection("jdbc:odbc:xyz", "system", "manager");
 int sid=126;
 String nameStudent="Ankita";
 String courseName="JDBC";
 String letterGrade="A";
 String qs = "insert into student values (" + sid + ", " +
 "" + nameStudent + "", "" + courseName + "", "" + letterGrade + "")";
 Statement stmt=connection.createStatement();
 stmt.executeUpdate(qs);
 System.out.println("Insertion is complete");
 stmt.close();
 connection.close();
}
}
```

### Output of the program:



```
c:\ Command Prompt
E:\Rojalin>javac InsertionClass.java
E:\Rojalin>java InsertionClass
Insertion is complete
E:\Rojalin>_
```

After inserting the record into the table, the table structure will be



### PreparedStatement

It is an interface present in java.sql package. Main objective of this interface is to retrieve the data at runtime. The **PreparedStatement** object represents the precompiled SQL statement. It reduces the time duration of execution. The **PreparedStatement** uses the '?' with SQL statement that provides the facility for setting an appropriate conditions in it.

We can get the PreparedStatement object by using the following line of code:

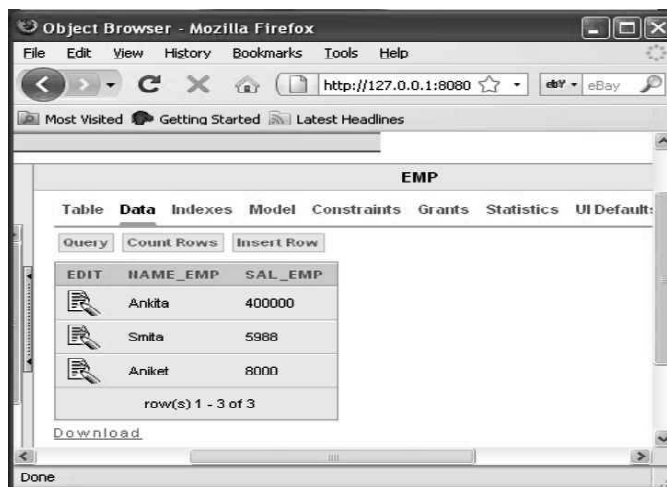
```
PreparedStatement ps = conn.prepareStatement
("insert into javaemp <table name> values (?....)");
```

Here '?' mark varies with respect to the variables.

#### Methods present in PreparedStatement:

- **executeQuery()**—Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.
- **executeUpdate()**—Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement.
- **execute()**—Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
- **setBoolean(int i, boolean b)**—Sets the designated parameter to the given Java Boolean value.
- **setInt(int a, int b)**—Sets the designated parameter to the given Java int value.
- **setDouble(int i, double d)**—Sets the designated parameter to the given Java double value.
- **setString(int i, String s)**—Sets the designated parameter to the given Java String value.
- **getMetaData()**—Retrieves a ResultSetMetaData object that contains information about the columns of the ResultSet object that will be returned when this PreparedStatement object is executed.

For this example we required one **EMP** table, where we need to insert data to the database through the commandprompt. These are the following fields in the **EMP** table **NAME\_EMP** (VARCHAR2), **SAL\_EMP** (NUMBER).



#### NOTES

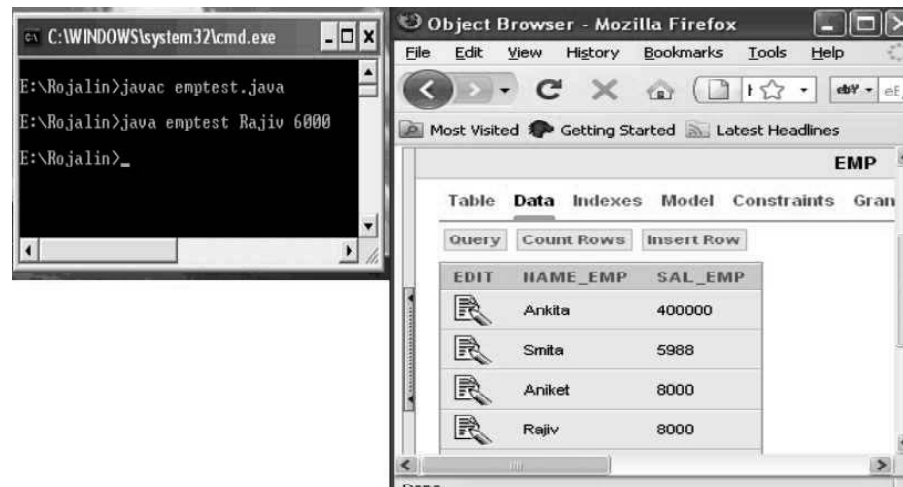
## NOTES

### Insert Data into the EMP Table (Using PreparedStatement)

#### Program 5.11

```
import java.sql.*;
class emptest
{
public static void main(String args[] throws Exception
{
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con=
DriverManager.getConnection ("jdbc:odbc:xyz", "system", "manager");
PreparedStatement ps = con.prepareStatement ("insert into emp
(NAME_EMP, SAL_EMP) values (?, ?)");
ps.setString (1, args [0]);
ps.setInt (2, Integer.parseInt (args [1]));
ps.executeUpdate ();
}
}
```

#### Output of the program:



### Use of ResultSet Interface

#### Program 5.12

```
import java.sql.*;
public class resultset
{
public static void main(String args[] throws Exception
{
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn=
DriverManager.getConnection ("jdbc:odbc:xyz", "system", "manager");
Statement st=conn.createStatement ();
ResultSet rs=st.executeQuery ("select * from student");
ResultSetMetaData rsmd=rs.getMetaData ();
int x=rsmd.getColumnCount ();
for (int i=1; i<=x; i++)
{
String y=rsmd.getColumnName (i);
System.out.println (y);
}
}
```

```

System.out.println(x);
}
}

```

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

### Output fo the program:

```

C:\WINDOWS\system32\cmd.exe
E:\Rojalin>javac resultset.java
E:\Rojalin>java resultset
SID
NAMESTUDENT
COURSENAME
LETTERGRADE
4
E:\Rojalin>

```

### NOTES

### Use of DatabaseMetaData Interface

#### Program 5.13

```

import java.sql.*;
public class database
{
public static void main (String args []) throws Exception
{
Connection con=null;
ResultSet rs=null;
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
con=
DriverManager.getConnection ("jdbc:odbc:xyz", "system", "manager");
DatabaseMetaData dbmt = con.getMetaData ();
String s1=dbmt.getDatabaseProductName ();
System.out.println (s1);
}
}

```

### Output of the program:

```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Prasant>E:
E:\>cd Rojalin
E:\Rojalin>javac database.java
E:\Rojalin>java database
Oracle

```

### How to create one table via JDBC?

#### Program 5.14

```

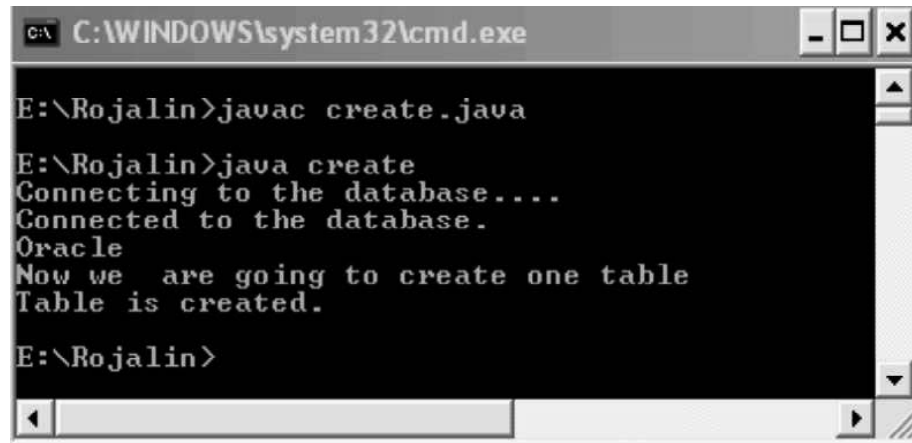
import java.sql.*;
public class create
{
public static void main (String args []) throws Exception
{

```

## NOTES

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
System.out.println("Connecting to the database....");
Connection con =
DriverManager.getConnection("jdbc:odbc:xyz","system","manager");
System.out.println("Connected to the database.");
DatabaseMetaData dbmt = con.getMetaData();
String s1 = dbmt.getDatabaseProductName();
System.out.println(s1);
System.out.println("Now we are going to create one table");
Statement st = con.createStatement();
st.executeUpdate("create table Interface_employee (emp_id
varchar2(15), emp_name varchar2(15), emp_addr varchar2(32))");
System.out.println("Table is created.");
}
}
```

### Output of the program:



```
C:\WINDOWS\system32\cmd.exe
E:\Rojalin>javac create.java
E:\Rojalin>java create
Connecting to the database....
Connected to the database.
Oracle
Now we are going to create one table
Table is created.
E:\Rojalin>
```

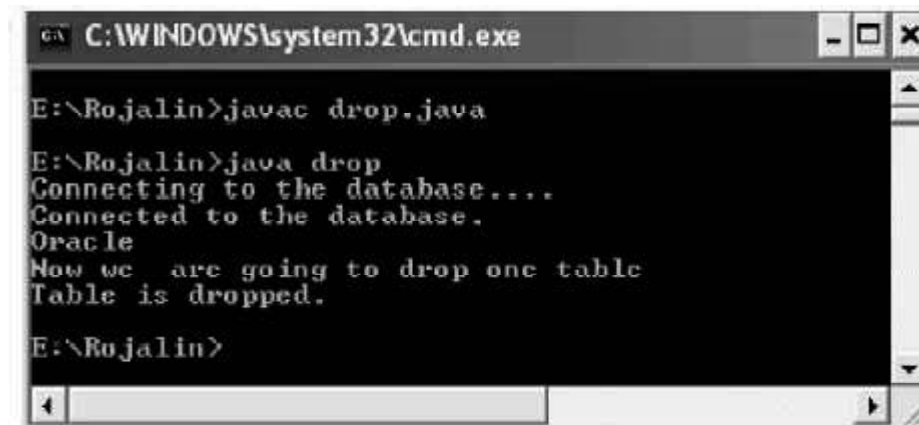
### How to drop one table via JDBC?

#### Program 5.15

```
import java.sql.*;
public class drop
{
public static void main(String args[]) throws Exception
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
System.out.println("Connecting to the database....");
Connection con =
DriverManager.getConnection("jdbc:odbc:xyz","system","manager");
System.out.println("Connected to the database.");
DatabaseMetaData dbmt = con.getMetaData();
String s1 = dbmt.getDatabaseProductName();
System.out.println(s1);
System.out.println("Now we are going to drop one table");
Statement st = con.createStatement();
st.executeUpdate("drop table Interface_employee");
System.out.println("Table is dropped.");
}
}
```



## Output of the program:



```
C:\WINDOWS\system32\cmd.exe

E:\Rojalin>javac drop.java

E:\Rojalin>java drop
Connecting to the database...
Connected to the database.
Oracle
Now we are going to drop one table
Table is dropped.

E:\Rojalin>
```

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

## NOTES

### Using Joins

Joins are one of the basic constructions of SQL and Databases as such - they combine records from two or more database tables .

### Types of Joins

- **Inner Join**
- **Outer Join**
- **Cross Join**
- **Natural Join**
- **Equi Join**

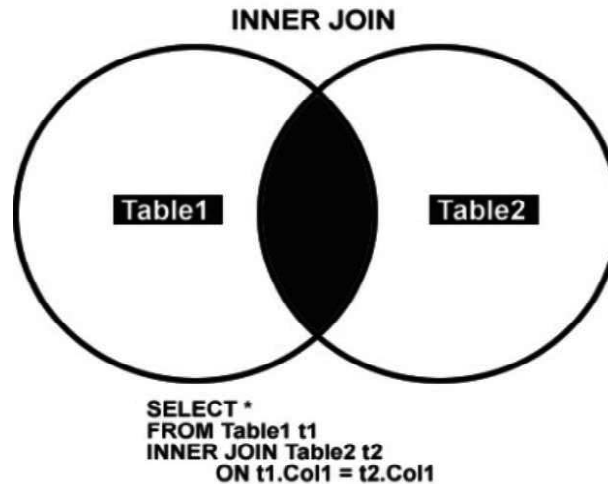
### Inner Join

An inner join is the most common join operation used in applications, and represents the default join-type. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row. The result of the join can be defined as the outcome of first taking the Cartesian product (or cross-join) of all records in the tables (combining every record in table A with every record in table B)—then return all records which satisfy the join predicate.

### Syntax

```
SELECT column_name (s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

## NOTES



*Fig. 5.14 Inner Join*

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.P_Id=Orders.P_Id
ORDER BY Persons.LastName
```

<b>LastName</b>	<b>FirstName</b>	<b>OrderNo</b>
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678

### Program 5.16

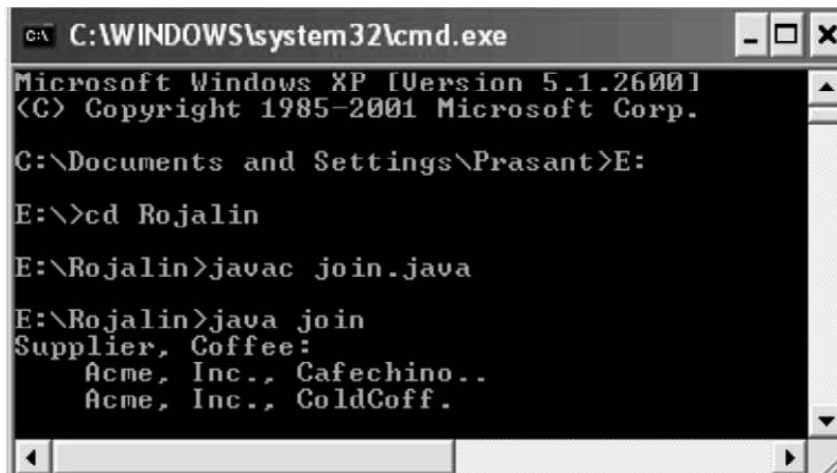
```
import java.sql.*;
public class join {
 public static void main (String args []) {
 String url = "jdbc:odbc:xyz";
 Connection con;
 String query = "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME" +
 "from COFFEES, SUPPLIERS" +
 "where SUPPLIERS.SUP_NAME like 'Acme, Inc.'" +
 "SUPPLIERS.SUP_ID=COFFEES.SUP_ID";
 Statement stmt;

 try {
 Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
 } catch (java.lang.ClassNotFoundException) {
 System.err.println ("ClassNotFoundException:");
 System.err.println (e.getMessage ());
 }
 try {
 con = DriverManager.getConnection (url,
 "system", "manager");
 stmt = con.createStatement ();
 ResultSet rs = stmt.executeQuery (query);
 System.out.println ("Supplier, Coffee:");
 while (rs.next ()) {
 String supName = rs.getString (1);
 String cofName = rs.getString (2);
 System.out.println (" "+ supName + ", " + cofName);
 }
 }
 }
}
```

```
 }
 stmt.close();
 con.close(); }
 catch (SQLException ex) {
 System.err.println("SQLException:"); }
 }
}
}
```

## NOTES

### Output of the program:



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
<C> Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Prasant>E:

E:\>cd Rojalin

E:\Rojalin>javac join.java

E:\Rojalin>java join
Supplier, Coffee:
Acme, Inc., Cafechino..
Acme, Inc., ColdCoff.
```

### Cross Join

A **cross join**, **cartesian join** or **product** provides the foundation upon which all types of inner joins operate. A cross join returns the cartesian product of the sets of records from the two joined tables. Thus, it equates to an inner join where the join-condition always evaluates to *True* or where the join-condition is absent from the statement.

If A and B are two sets, then the cross join is written as  $A \times B$ .

### Syntax

```
SELECT *
FROM employee CROSS JOIN department
```

### Implicit Cross Join

```
SELECT *
FROM A, B
```

### JDBC Example with Respect to Cross Join

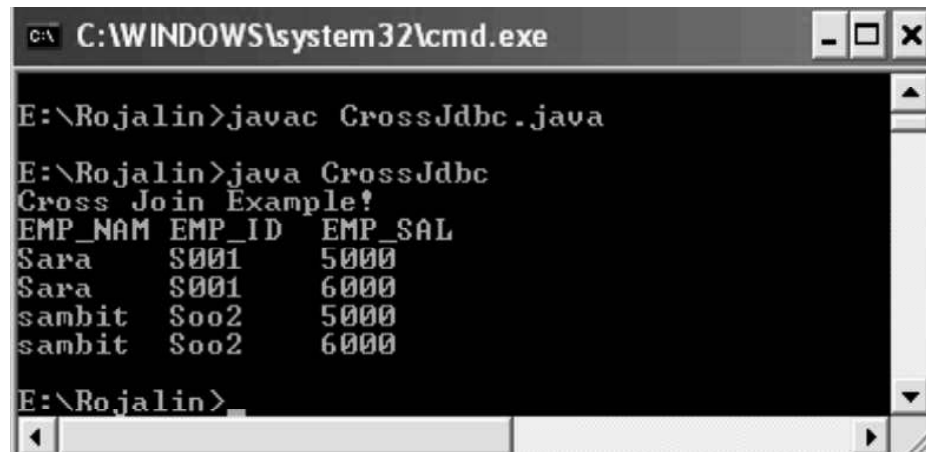
#### Program 5.17

```
import java.sql.*;
class CrossJdbc {
 public static void main (String[] args)
 {
 System.out.println("Cross Join");
 Connection con=null;
 try {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 con=DriverManager.getConnection
```

## NOTES

```
("jdbc:odbc:xyz", "system", "manager");
try{
 Statement st=con.createStatement();
 ResultSet res=st.executeQuery
("SELECT *FROM" + "INTER_EMP1" + "," + "INTER_SAL");
 System.out.println("EMP_NAM" + "\t" + "EMP_ID" + "\t"
+ "EMP_SAL"); while(res.next()) {
 String name=res.getString("EMP_NAM");
 String ed=res.getString("EMP_ID");
 String sal=res.getString("EMP_SAL");
 System.out.println(name + "\t" + ed + "\t" + sal);
 }
}
catch (SQLExceptions) {
 System.out.println("SQL statement is not executed!");
}
}
catch (Exception e) {
 e.printStackTrace(); } }
}
```

### Output of the program:



```
C:\WINDOWS\system32\cmd.exe
E:\Rojalin>javac CrossJdbc.java
E:\Rojalin>java CrossJdbc
Cross Join Example!
EMP_NAM EMP_ID EMP_SAL
Sara S001 5000
Sara S001 6000
sambit Soo2 5000
sambit Soo2 6000
E:\Rojalin>
```

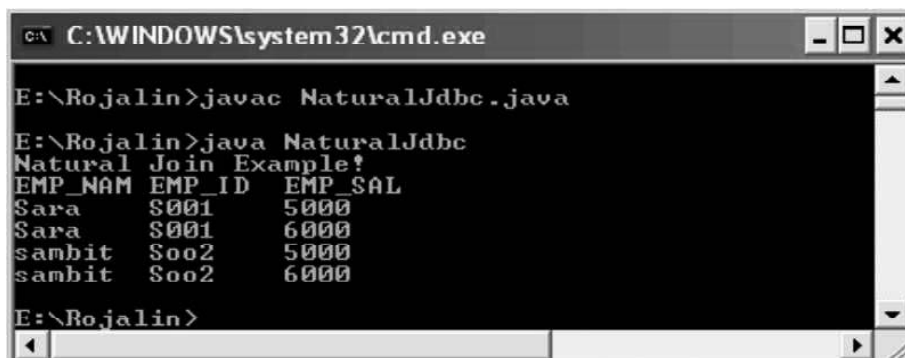
### Natural Join

#### Program 5.18

```
import java.sql.*;
class NaturalJdbc {
 public static void main (String[] args)
 {
 Connection con=null;
 try{
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 con=DriverManager.getConnection
("jdbc:odbc:xyz", "system", "manager");
 try{
 Statement st=con.createStatement();
 ResultSet res=st.executeQuery
("SELECT *FROM" + "INTER_EMP1" + "NATURAL LEFT JOIN" +
"INTER_SAL");
 System.out.println("EMP_NAM" + "\t" + "EMP_ID" + "\t" + "EMP_SAL");
 }
 }
 }
}
```

```
while (res.next()) {
 String name = res.getString("EMP_NAM");
 String id = res.getString("EMP_ID");
 String sal = res.getString("EMP_SAL");
 System.out.println(name + "\t" + id + "\t" + sal);
}
}
catch (SQLExceptions) {
 System.out.println("SQL statement is not executed!");
}
}
catch (Exception e) {
 e.printStackTrace();
}}
```

### Output of the program:



```
C:\WINDOWS\system32\cmd.exe
E:\Rojalin>javac NaturalJdbc.java
E:\Rojalin>java NaturalJdbc
Natural Join Example!
EMP_NAM EMP_ID EMP_SAL
Sara S001 5000
Sara S001 6000
sambit Soo2 5000
sambit Soo2 6000
E:\Rojalin>
```

## Using Transactions

### Program 5.19

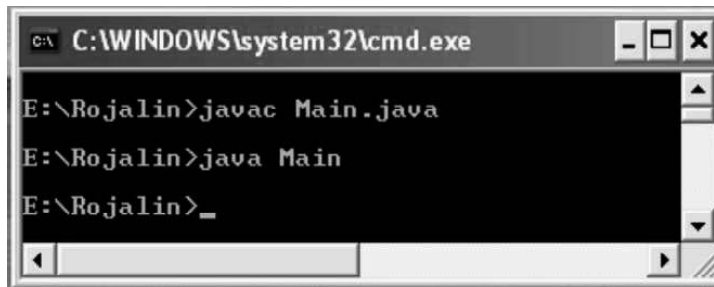
```
import java.sql.*;
public class Main {
 public void transaction() {
 Connection con = null;
 Statement statement = null;
 try {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 con = DriverManager.getConnection("jdbc:odbc:xyz",
 "system", "manager");
 con.setAutoCommit(false);
 statement = con.createStatement();
 statement.executeUpdate("UPDATE Table1 SET Value=1
 WHERE Name='fool'");
 statement.executeUpdate("UPDATE Table2 SET Value=2
 WHERE Name='Sunidhi'");
 con.commit();
 } catch (ClassNotFoundException ex) {
 ex.printStackTrace();
 } catch (SQLException ex) {
 ex.printStackTrace();
 }
 try {
 con.rollback();
 } catch (SQLException ex1) {}
 } finally {
 try {
 if (statement != null)
 statement.close();
 }
 }
}
```

## NOTES

## NOTES

```
 if (con != null)
 con.close();
 } catch (SQLException ex) { }
 }
} public static void main (String [] args) {
 new Main ().Transaction ();
} }
```

### Output of the program:



```
C:\WINDOWS\system32\cmd.exe
E:\Rojalin>javac Main.java
E:\Rojalin>java Main
E:\Rojalin>_
```

### Program on Applet to JDBC Communication

#### Program 5.20

```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.applet.Applet;
public class Login extends Applet implements ActionListener
{
 Panel p, p1, p2, p3;
 Label l, l1, l2;
 TextField tf1, tf2;
 Button b1, b2;
 Font f1, f2;
 public void init ()
 {
 setLayout (new GridLayout (4, 1));

 p=new Panel ();
 p1=new Panel ();
 p2=new Panel ();
 p3=new Panel ();

 f1=new Font ("Verdana", Font.BOLD, 24);
 f2=new Font ("Arial", Font.BOLD, 18);

 l=new Label ("Login");
 l.setFont (f1);
 l1=new Label ("User Id");
 l1.setFont (f2);

 l2=new Label ("Password");
 l2.setFont (f2);

 tf1=new TextField (12);
 tf2=new TextField (12);
 tf2.setEchoChar ('*');
 tf1.setFont (f2);
```



## NOTES

### Output of the program:



### Running Procedure of Applet to JDBC using Type-1 Driver

Before run the program first create the DSN for Type-1 driver then set the policytool for applet communication to JDBC

### How to Create the DSN?

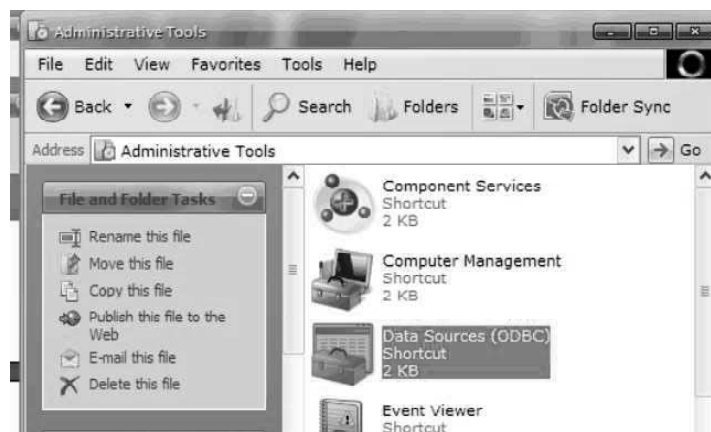
#### Step 1

First open control panel inside control panel click on “Administrative Tool”.



#### Step 2

Inside administrative tool choose “Data Sources(ODBC)”.

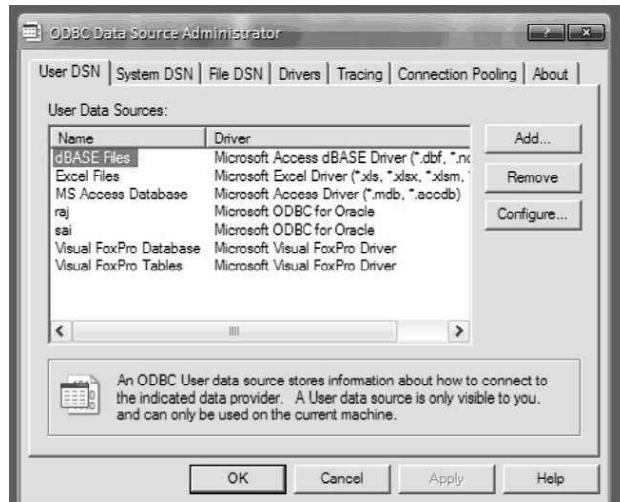




### Step 3

After Step 2 click on “Add” button from the window given below.

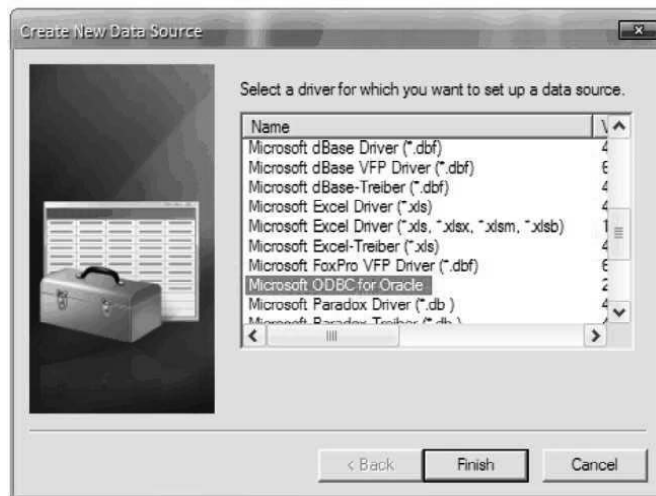
*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*



### NOTES

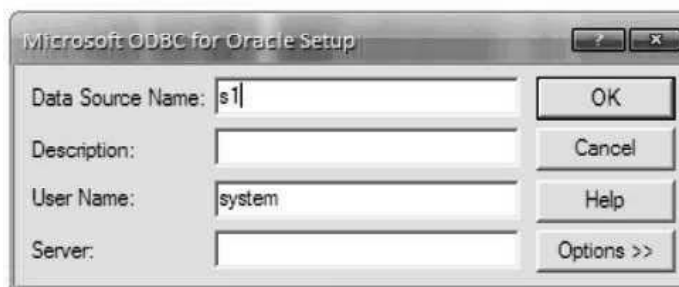
### Step 4

After clicking on “Add” button choose “Microsoft ODBC for Oracle”.



### Step 5

After selecting “Microsoft ODBC for Oracle” it will open a new window. Inside that window pass the name of “DSN” (i.e., s1).

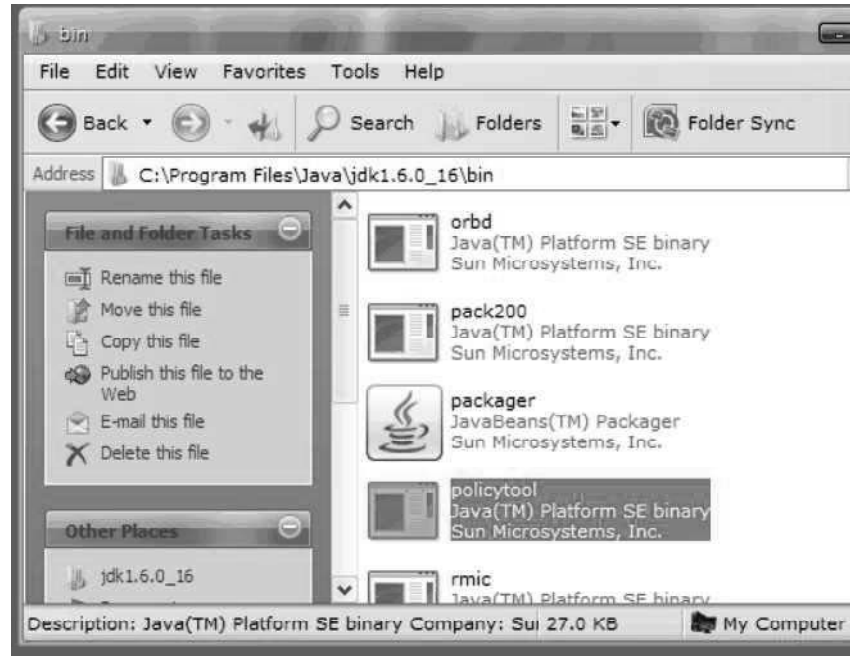


## NOTES

### How to set Policy Tool?

#### Step 1

First click on “Policy Tool” present in “C:\Program Files\Java\jdk1.6.0\_16\bin”



#### Step 2

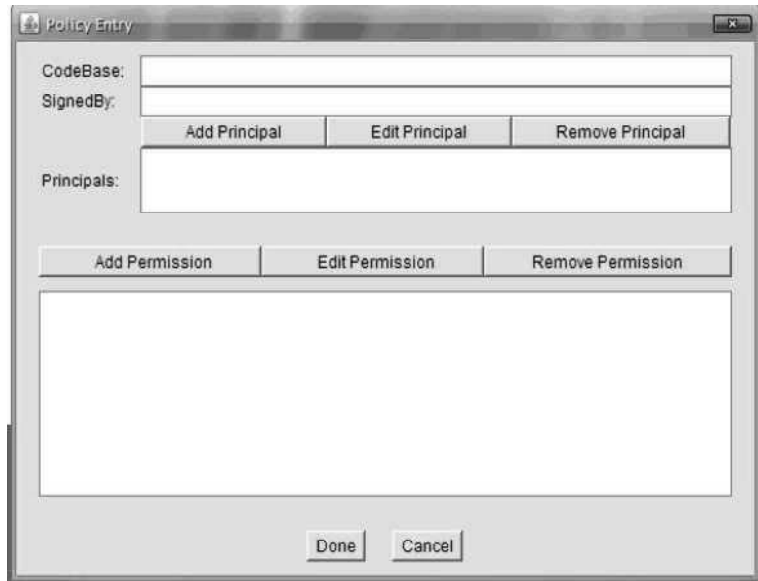
After click it will open a new window from that window we choose the “Add Policy Entry” button.



#### Step 3

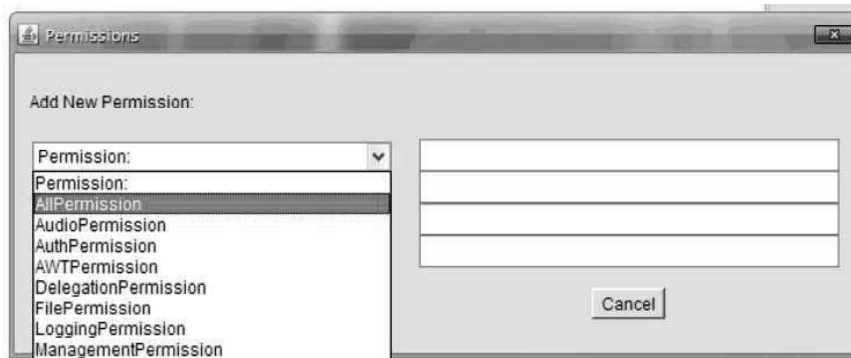
After click “Add Policy Entry” it will open a window from that window we choose “Add Permission” button.

## NOTES



### Step 4

After Step 3 from next screen we choose "All Permission"



### Step 5

After choose "AllPermission" save the file name as ".java.policy" in "C:\Documents and Settings\Administrator\.java.policy"



## NOTES

### An Introduction to SQL

**SQL (Structured Query Language)** is a database computer language designed for managing data in relational database management systems or RDBMS. The most common operation in SQL is the query, which is performed with the declarative SELECT statement. SELECT retrieves data from one or more tables.

SELECT in a statement, is used to retrieve information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT in a statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car\_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First\_Name and Last\_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

```
FIRST_NAME LAST_NAME

Axel Washington
Florence Wojokowski
```

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no WHERE clause). Note that SELECT \* means “SELECT all columns.”

```
SELECT *
FROM Employees
```

### WHERE Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last\_Name begins with the string ‘Washington’.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

### Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars, as shown below.

**Table 5.1: Cars Table**

Car Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

**NOTES**

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car\_Number, which is the primary key for the table Cars and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car\_Number 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car\_Number) in Employees would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car\_Number column in the table Employees because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the FROM clause lists both Employees and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name, Cars.Make,
Cars.Model, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

FIRST_NAME YEAR	LAST_NAME	MAKE	MODEL
----- -----	-----	-----	-----
Axel 1996	Washington	Honda	CivicDX
Florence 1999	Wojokowski	Toyota	Corolla

**Common SQL Commands**

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

## NOTES

A list of the more common DML commands follows:

- **SELECT**—used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.
- **INSERT**—adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- **DELETE**—removes a specified row or set of rows from a table
- **UPDATE**—changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- **CREATE TABLE**—creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- **DROPTABLE**—deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional. In addition, the behavior of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.
- **ALTER TABLE**—adds or removes a column from a table. It also adds or drops table constraints and alters column attributes.

### Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

## **Transactions**

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

## **Stored Procedures**

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are byte code portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

## **Metadata**

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface Database Metadata, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

## **NOTES**

## NOTES

### Contents of java.sql package

It has already been seen that the JDBC API defines a set of interfaces and classes, which are found in the **java.sql package**.

#### Interfaces in java.sql package

The various interfaces are as follows:

##### Array

This interface is used to map Java array into SQL type **Array**. This interface contains some predefined methods. These are:

- `getArray()`
- `getBaseType()`
- `getBaseTypeName()`
- `getResultSet()`

##### Blob

It is an interface present in **java.sql package**. An SQL **Blob** is a built-in data type that stores a binary large object in a database table. The methods of this interface are:

- `getBinaryStream()`
- `getBytes()`
- `length()`
- `position()`

##### CallableStatement

It is an interface present in **java.sql package** and is used to call SQL stored procedures. A **CallableStatement** may return a **ResultSet** or multiple **ResultSets**. Escape syntax is used for procedures that return a parameter. This interface extends the **PreparedStatement** interface. The methods in this interface include `getXXX` (where **XXX** stands for any datatype) methods and the following:

- `registerOutParameter()`
- `wasNull()`

##### Clob

It is an interface present in **java.sql package**. In SQL, **Clob** is a built-in data type that stores a character large object in a database table. The methods of this interface are:

- `getAsciiStream()`
- `getCharacterStream()`
- `getSubString()`
- `length()`
- `position()`

##### Connection

It is an interface present in **java.sql package**. A **Connection** is a session in a specific database engine. Information such as database tables, stored procedures and other database objects may be obtained from a **Connection** with the `getMetaData()` methods. Some of the important methods in this interface are:

- `commit()`
- `createStatement()`
- `getAutoCommit()`
- `isClosed()`
- `isReadOnly()`
- `prepareCall()`



- `prepareStatement()`
- `rollback()`
- `setAutoCommit()`
- `setReadOnly()`

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

## DatabaseMetaData

This interface provides information regarding the database itself, such as, version information, table names, and supported functions. Many of the methods of this interface return lists of information in the form of `ResultSet` objects. Some of the important methods of this interface are:

- `getCatalogs()`
- `getConnection()`
- `getDatabaseProductVersion()`
- `getDriverName()`
- `getMaxRowSize()`
- `getColumns()`
- `getDriverVersion()`
- `isReadOnly()`

## Driver

Every driver must implement the **Driver** interface. This interface is used to create connection objects. When a **Driver** class is loaded, first it must create an instance of the **Driver** and then it is registered in the `DriverManager`. The following are the methods present in the **Driver** interface.

- `acceptsURL()`
- `connect()`
- `getMajorVersion()`
- `getMinorVersion()`
- `getPropertyInfo()`
- `jdbcCompliant()`

## Ref

It is an interface present in **java.sql package**. This interface is a reference to an SQL structure type value in the database. The reference of the interface is saved in the persistent storage mechanism. The method present in this interface is `getBaseTypeName()`.

## ResultSet

This interface provides methods for the retrieval of data returned by an SQL statement execution. A **ResultSet** maintains a cursor pointing to its current row of data. The most often used methods, namely, **getXXX** and **updateXXX** methods are present in this interface. The other important methods present in this interface are:

- `absolute()`
- `afterLast()`
- `beforeFirst()`
- `cancelRowUpdate()`
- `close()`
- `deleteRow()`
- `insertRow()`
- `next()`
- `previous()`
- `wasNull()`

## ResultSetMetadata

This interface is used for the collection of **meta data** information associated with last **ResultSet** object. Some of the important methods of this interface are:

## NOTES

- getCatalogName ()
- getColumnCount ()
- isReadOnly ()
- getColumnName ()
- isNullable ()

## NOTES

### SQLData

It is an interface present in **java.sql package**. This interface is used to map the SQL user-defined data types. Some important methods of this interface are:

- getSQLTypeName ()
- writeSQL ()
- readSQL ()

### SQLInput

It is an interface present in **java.sql package**. It contains an `InputStream` that contains stream-oriented values. The programmer does not invoke this interface. Rather, the driver uses it. The `readXXX` methods (where `XXX` represents any data type) of this interface are used to read the attributes from the input stream. Another method in this interface is `wasNull()`.

### SQLOutput

It is an interface present in **java.sql package**. This interface uses the `OutputStream` for writing the attributes of user-defined data types in the data base. This interface is also used by the driver. However, the programmer does not invoke it directly. The **writeXXX** methods (where `XXX` represents any data type) of this interface are used to write data on **SQLData object**.

### Statement

The methods of the **Statement** interface are used to execute SQL statements and to retrieve data into the `ResultSet`. A **Statement** can open only one `ResultSet` at a time. Some of the important methods of this interface are:

- cancel ()
- execute ()
- executeUpdate ()
- getFetchSize ()
- getAttributes ()
- getSQLTypeName ()
- close ()
- executeBatch ()
- getConnection ()
- getMaxRow ()

### Classes in java.sql

The various classes in Java are given here.

### Date

The **Date** class contains methods to perform conversion of SQL date formats and Java **Date** objects. This class contains the following important methods:

- getHours ()
- getSeconds ()
- setMinutes ()
- setTime ()
- valueOf ()
- getMinutes ()
- setHours ()
- setSeconds ()
- toString ()

## DriverManager

This class is used to load and unload the **drivers** and establish the connection with the database. The important methods of this class are:

- getConnection()
- getDriver()
- getLogStream()
- println()
- registerDriver()

## DriverPropertyInfo

The methods of the **DriverPropertyInfo** class are used for the insertion and retrieval of driver properties. It is useful for advanced programmers. This class inherits its methods from the java.lang.Object class.

## Time

The **Time** class extends the Date class. It allows the JDBC to identify java.util.Date as an SQL Time value. The methods of this class are used to perform SQL time and Java time object conversions. The methods available in this class are:

- getDate()
- getDay()
- getMonth()
- getMonth()
- getYear()
- setDate()
- setMonth()
- setTime()
- setYear()
- toString()
- valueOf()

## TimeStamp

The **TimeStamp** class also extends the Date class. It provides additional precision to the Java Date object by adding a nanosecond field. The methods of this class are:

- after()
- before()
- equals()
- getNanos()
- setNanos()
- toString()
- valueOf()

## Types

The **Types** class extends the java.lang.Object class. This class defines constants that are used to identify generic SQL types called JDBC types. Its methods are inherited from the class object.

## Exceptions in java.sql package

There are three types of **Exceptions** in **java.sql package**. Each of them is given below in detail.

### BatchUpdateException

It extends the SQLException. When an error occurs in the batch update operation, then the **BatchUpdateException** is thrown at runtime.

## NOTES

## NOTES

### SQLException

The **SQLException** class extends Exception class of java.lang package. It provides information on a database access error. The information given in the exception includes a string describing the error, a string describing the SQLState, the error code and a chain to the next exception.

### SQLWarning

The **SQLWarning** extends SQLException. It provides information on database access warnings and is chained to the object whose method caused it to be reported.

### Steps for using JDBC

There are seven basic steps for using **JDBC** to access a database. These are:

- Import the **java.sql package**
- Register the driver
- Connect to the database
- Create a statement
- Execute the statement
- Retrieve the results
- Close the statement and the connection

### Import the java.sql Package

The interfaces and classes of the JDBC API are present inside the package called **java.sql**. When the programmer wants to make Java-database connectivity, he is bound to import **java.sql package**. Its syntax is `import java.sql.*;`

### Register the Driver

In Java, if the programmer wants to **register** the driver, he calls the static method of DriverManager class. Its syntax is:

```
DriverManager.registerDriver(Driver dr);
```

### Connect to the Database

The next step is to **connect to the database**. The `getConnection()` method is used to establish the connection. Its syntax is:

```
DriverManager.getConnection(String url, String user, String passwd);
```

Where `url` is the database. The `url` is of the form `jdbc:subprotocol:subname.user` is the database user, and `passwd` is the password to be supplied to get connected to the database. The return value is connected to the `url`.

### Creating a statement

A statement can be **created** using three methods, namely, `createStatement()`, `prepareStatement()` and `prepareCall()`. The syntax of each of these is given below:

```
createStatement()
```

Its syntax is:

```
cn.createStatement();
```

This is used where **cn** is a connection object. This method creates and returns a statement object for sending SQL statement to the database.

```
cn.createStatement(int rsType, int rsConcur)
```

This is used where **cn** is a connection object, **rsType** and **rsConcur** are type and concurrency of ResultSet respectively. This method creates a statement object that will generate ResultSet objects with the given type and concurrency.

```
cn.prepareStatement(String str)
```

This is used where **cn** is a connection object and **str** is an **SQL** statement that may contain one or more IN parameter place holders. This method creates and returns a PreparedStatement object for sending **SQL** statements with parameters to the database.

```
cn.prepareStatement(String str, int rsType, int rsConcur)
```

This is used where **cn** is a connection object, **str** is a SQL statement, **rsType** is a result set type and **rsConcur** is a concurrency type. This method creates a PreparedStatement object that will generate ResultSet objects with the given concurrency.

```
cn.prepareCall(string str)
```

This is used where **cn** is a connection object and **str** is an SQL statement that may contain one or more IN parameter placeholders. This method creates and returns a CallableStatement object for calling database-storing procedures.

```
cn.preparecall(String str, int retype, int reConcur)
```

This is used where **cn** is a connection object, **str** is an SQL statement, **retype** is a result set type and **rsConcur** is a concurrency type. It creates a CallableStatement object that will generate ResultSet objects with the given type and concurrency.

SQL statements without parameters are normally executed using Statement objects. If the same SQL statement is executed many times, it is more efficient to use a PreparedStatement.

### Executing the Statement

There are three methods to **execute** the statement. These are `execute()`, `executeQuery()` and `executeUpdate()`. Its syntax is:

```
stmt.execute();
```

This is used where `stmt` is a statement object. This method returns a boolean value and is used to execute any SQL statement.

### **executeQuery()**

Its syntax is:

```
stmt.executeQuery(String str);
```

This is used where `stmt` is a statement object and `str` is an SQL statement. It is used to execute an SQL statement that may return multiple results. The return value is a boolean, which is true if the next result is a ResultSet and false if it is an update count or there are no more results. This is used where `stmt` is a PreparedStatement object. The method returns a ResultSet generated by executing the query in `stmt`. This method takes query as string and is invoked by the statement object `stmt` and returns the ResultSet.

## NOTES

## NOTES

### **executeUpdate ()**

Here **stmt** is an object of PreparedStatement. By the object of PreparedStatement, this method executes SQL statements. In case of insert, update, delete statement, this method's return value is an int, which counts the number of rows which are affected. The syntax is:

```
stmt.executeUpdate(String str);
```

Here **stmt** is a statement object and **str** is a SQL statement for performing insert, update or delete functions.

### **Retrieving the Results**

The results of the SQL statements (in particular queries are) are stored in a ResultSet object. To **retrieve** the data from the ResultSet, we need to use the getXXX methods. These methods **retrieve** the data and convert it to a Java data type. There is a separate getXXX method for each data type. For example, getString is used to retrieve the string value and getDate () is used to **retrieve** a date value. The getXXX takes one argument which is the index of the column in the ResultSet and returns the value of the column. To move into the next row in the ResultSet, one makes use of the ResultSet.next () method.

### **Closing the Statement and Connection**

The various methods are shown below.

#### **The close () Method**

It is not absolutely necessary to **close** the connection. However, since an open connection can cause problem, it is better to **close** the connections. The close() method is used to close the statements and connection. The syntax for closing an object is: stmt.close();

This is used where stmt is a statement object to be closed. This method releases stmt database. The return type is void. The syntax for closing a connection is:

```
cn.close ();
```

This is used where **cn** is the connection to be closed. The return type of this method is void.

### **Executing DDL and DML Commands**

Once the connection with the database is established, the user can start creating and working with the objects of the database. In this part, the way to execute **Data Definition Language (DDL)** and **Data Manipulation Language (DML)** commands is learnt.

#### **DDL Commands**

The **DDL commands** are create, alter and drop. The methods to execute each of these are given below. The create command is used to create database tables. The following tables would be needed in case of the rhythm.

<b>Customer</b>	
CustId	Number(3)
CustName	varchar2(15)
Address	varchar2(30)
<b>Product</b>	
ProdId	Number(3)
ProdName	varchar2(10)
Price	Number(5,2)
Stock-on-hand	Number(4)
<b>Transaction</b>	
TranDt	Date
TranId	Number(3)
ProdId	Number(3)
CustId	Number(3)
Qty	Number(2)

## NOTES

The create statements for creating the above three tables are as follows:

- Create a table named Customer (CustId Number(3), CustName varchar2(15), Address varchar2(30));
- Create a table named Product (ProdId Number(3), ProdName varchar2(10), Price Number(5,2), Stock- on- hand Number(4) );
- Create a table named Transaction (CustId Number(3), ProdId Number(3), tranId Number(3), Qty Number(2), TranDt Date);

The following example clarifies this further:

### Create the Table

#### Program 5.21

```
import java.sql.*;
public class Customer1
{
 public static void main (String args []) throws SQLException
 {
 DriverManager.registerDriver (new oracle.jdbc.driver.
 OracleDriver ());
 System.out.println ("Connecting to the database...");
 Connection cn=DriverManager.getConnection
 ("jdbc:oracle:thin:@rashmi:1521:orcl1", "sai", "sai");
 System.out.println ("Connected to the database.");
 Statement st =cn.createStatement ();
 try{
 st.executeUpdate ("create table Customer (CustId
number (3) , CustName varchar2 (15) , Address varchar2 (30))");
 System.out.println ("Table Customer Created");
 } catch (SQLException ex)
 {
 System.out.println ("The Exception raised is" + ex);
 }
 st.close ();
 cn.close ();
 }
}
```

## NOTES

### Output of the program:

```
Connecting to the database...
Connected to the database.
Table Customer Created
The next example shows altered table transaction.
```

### Altered Table

#### Program 5.22

```
import java.sql.*;
public class Customer_alt
{
 public static void main (String args []) throws SQLException
 {
 DriverManager.registerDriver (new
oracle.jdbc.driver.OracleDriver ());
 try{
 Connection cn=DriverManager.getConnection
("jdbc:oracle:thin:@rashmi:1521:orc11","sai","sai");
 System.out.println ("Connected to the database");
 Statement st=cn.createStatement ();
 st.executeUpdate ("alter table Transaction modify (Qty
Number (4))");
 System.out.println ("Table Transaction altered");
 } catch (Exception ex)
 {
 System.out.println ("The Exception raised is:" +ex);
 }
 }
}
```

### Output of the program:

```
Connected to the database
Table Transaction altered
Consider the following example:
```

### Table Dropped

#### Program 5.23

```
import java.sql.*;
public class Customer_drop
{
 public static void main (String args []) throws SQLException
 {
 DriverManager.registerDriver (new
oracle.jdbc.driver.OracleDriver ());
 System.out.println ("Connecting to the database...");
 try{
 Connection cn=DriverManager.getConnection
("jdbc:oracle:thin:@rashmi:1521:orc11","sai","sai");
 System.out.println ("Connected to the database");
 Statement st=cn.createStatement ();
 st.executeUpdate ("drop table Trans");
 System.out.println ("Table Trans dropped");
 } catch (Exception ex)
 {

```



```

 System.out.println("The exception raised is:" + ex);
 }
}

```

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

### Output of the program:

```

Connecting to the database....
Connected to the database....
Table Trans dropped

```

### DML Commands

The data manipulation language commands are: select, insert, update and delete commands. Using these the tables are created. The insert command is used to input the data and the select command is used to retrieve the records from the tables. The following example shows the way to insert a row in the table.

### Insert Table

#### Program 5.24

```

import java.sql.*;
public class CustomerInsert
{
 public static void main (String args []) throws SQLException
 {
 DriverManager.registerDriver (new
oracle.jdbc.driver.Oracle.OracleDriver ());
 try{
 Connection cn=DriverManager.getConnection
("jdbc:oracle:thin:@rashmi:1521:orcl1","sai","sai");
 System.out.println("Connected to the database");
 Statement st=cn.createStatement ();
 st.executeUpdate ("insert into customer values (100, 'usha',
'100, Naya Bazar, Cuttack, Orissa')");
 System.out.println("One row inserted");
 st.close ();
 cn.close ();
 } catch (Exception ex)
 {
 System.out.println("The Exception raised is "+ ex);
 }
 }
}

```

### Output of the program:

```

Connected to the database
One row inserted

```

### Update a Row

#### Program 5.25

```

import java.sql.*;
import java.io.*;
public class ProductUpdate
{
 public static void main (String args []) throws
SQLException, IOException

```

### NOTES

## NOTES

```
{
 DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
 String refValue;
 String updateValue;
 String str;
 try{
 Connection cn=DriverManager.getConnection
("jdbc:oracle:thin:@:1521:orcl11","sai","sai");
 refValue=readEntry("Enter the product ID: ");
 updateValue=readEntry("Enter the new price: ");
 Statement st=cn.createStatement();
 str="update product set price="+updateValue+" "+"where
prodID="+refValue;
 st.executeUpdate(str);
 System.out.println("Row Updated");
 st.close();
 cn.close();
 }catch (Exception ex)
 {
 System.out.println("The Exception raised is "+ex);
 }
}
static String readEntry(String prompt)
{
 try{
 StringBuffer tempo=new StringBuffer();
 System.out.print(prompt);
 System.out.flush();
 int c=System.in.read();
 while (c!='\n' && c!=-1)
 {
 tempo.append((char)c);
 c=System.in.read();
 }
 return tempo.toString().trim();
 }catch (IOException ex)
 {
 return "";
 }
}
}
```

### Output of the program:

Enter the product ID: 100

Enter the new price: 110.00

Row Updated

### Delete One Row

#### Program 5.26

```
import java.sql.*;
public class CustomerDel
{
```

```
public static void main (String args []) throws SQLException
{
 DriverManager.registerDriver (new
 oracle.jdbc.driver.OracleDriver ());
 try{
 Connection cn=DriverManager.getConnection
 ("jdbc:oracle:thin:@rashmi:1521:orc11","sai","sai");
 System.out.println("Connected to the database");
 Statement st=cn.createStatement ();
 st.executeUpdate ("delete from Product where
ProdID=105");
 System.out.println("One row deleted");
 } catch (Exception ex)
 {
 System.out.println("The Exception raised is:" +ex);
 }
}
```

### Output of the program:

Connected to the database

One row deleted

### Joins and Transactions

Sometimes, it is required to use two or more tables to get the data. This is a case where a **join** is needed. A **join** is a database operation that relates to two or more tables by names of values that they share in common.

### Joins

There are different types of **joins** available in Oracle. Examples of **equi join** and **outer join** have been given.

The following example shows the way to join the two tables:

### Joining Two Tables

#### Program 5.27

```
import java.sql.*;
import java.io.*;
public class ListTran
{
 public static void main (String args []) throws
SQLException, IOException
 {
 DriverManager.registerDriver (new
oracle:jdbc:driver.OracleDriver ());
 try{
 Connection cn=DriverManager.getConnection

("jdbc:oracle:thin:@rashmi:1521:orc11","sai","sai");
 Statement st=cn.createStatement ();
 ResultSet rs=st.executeQuery ("select
product.prodID, ProdName, trained, qty from product,
transaction where product.ProdID=transaction.ProdID");
 System.out.println ("ProdID\tProdName\t\
```

## NOTES

## NOTES

```
tTranID\tQuantity");
while (rs.next ())
{
 System.out.println(rs.getInt(1)+"\t"+rs.
getString(2)+"\
t"+rs.getInt(3)+"\t"+rs.getInt(4));
}
rs.close();
st.close();
cn.close();
}catch (Exceptionex)
{
 System.out.println("theexceptionis"+ex);
}
}
}
```

### Output of the program:

<i>ProdID</i>	<i>ProdName</i>	<i>TranID</i>	<i>Quantity</i>
001	Gajani	1	2
001	Gajani	4	3
002	Lagan	2	3
004	Mann	5	1
005	Rangeela	3	1

### Transaction

In case of **transaction**, one SQL statement waits for another statement to be executed. Let us take the instance of the rhythm. Whenever there is a **transaction**, in addition to inserting the corresponding record in the **transaction** table, the corresponding row in the product table should also be updated. If either of the operations fails, then the data will become inconsistent. In order to be sure that either both the operations are executed or neither of them is executed, one can make use of **transaction**. A **transaction** is a set of one or more statements that are executed together as a unit.

```
cn.setAutoCommit (false);
```

### Transaction

#### Program 5.28

```
import java.sql.*;
import java.io.*;
public class TransCmt
{
 public static void main (String args []) throws
SQLException, IOException
 {
 DriverManager.registerDriver (new
oracle.jdbc.driver.OracleDriver ());
 try {
 Connection cn = DriverManager.getConnection
("jdbc:oracle:thin:@rashmi:1521:orc11", "sai", "sai");
 cn.setAutoCommit (false);
```

```
Statement stm=cn.createStatement();
stm.executeUpdate("Insert into transaction values (103,
101,
'3-jan-09')");
Statement st=cn.createStatement();
ResultSet rs;
rs=st.executeQuery("select stock_on_hand from product
where
ProdID=103");
rs.next();
int i=rs.getInt(1);
i=i-1;
Statement stmt=cn.createStatement();
stmt.executeUpdate("Update Product set stock_on_hand
="+
i + "where ProdID=103");
cn.commit();
System.out.println("Changes committed");
st.close();
cn.close();
} catch (Exception ex)
{
System.out.println("The Exception raised is "+ ex);
cn.rollback();
}
}
```

### Output of the program:

Changes committed

---

## 5.4 SWINGS

---

The sample program given below shows an example to write a Swing application. It uses two Swing components: JFrame (top-level container) and JButton (component which creates a push button).

**Example 5.2:** A program to demonstrate the creation of a Swing application.

```
import javax.swing.*;
class MySwingDemo extends JApplet
{
public static void main (String str[])
{
//creates a new frame
JFrame jf=new JFrame ("Swing application");
//specify the initial size of the frame.
jf.setSize (300,125);
//Terminates the program
jf.setDefaultCloseOperation (jf.EXIT_ON_CLOSE);
//create a new button
JButton b1=new JButton ("Click");
//Adding button to the frame
jf.add (b1);
//Display the frame
```

### NOTES

```
jf.setVisible(true);
}
}
```

## NOTES

Swing programs are compiled and executed like any other Java application. The program given in Example 5.2 can be compiled by using the following command.

```
javac MySwingDemo.java
```

To execute the program, following command is used.

```
java MySwingDemo
```

**The output of the program is**



The explanation of this program is as follows:

1. In the beginning of the program, the `javax.swing` package is imported.
2. Next, `MySwingDemo` class which extends the `JApplet` class is declared.

An object of `JFrame` class is created using the statement.

```
JFrame jf = new JFrame("Swing application");
```

It creates a container called `jf` that defines a rectangular window with a specified string displayed on the title bar.

3. The size of the window is specified by using the statement.

```
jf.setSize(300, 125);
```

The general form of `setSize()` method to set the size of the window is:

```
void setSize(int width, int height)
```

where,

`width` is the width of the window

`height` is the height of the window

4. The statement used for terminating the program when the window is closed is

```
jf.setDefaultCloseOperation(jf.EXIT_ON_CLOSE);
```

When the above statement is executed, the entire application terminates with the closing of window.

The general form of `setDefaultCloseOperation()` is

```
void setDefaultCloseOperation(int w)
```

where,

`w` specifies the action to be performed on the window when it is closed. It has several options which are as follows

```
JFrame.DISPOSE_ON_CLOSE
JFrame.HIDE_ON_CLOSE
JFrame.DO_NOTHING_ON_CLOSE
```

By default, when the top-level window is closed, the application is not terminated. It simply removes the window from the screen.

5. A Swing component `JButton` is created using the statement.

```
JButton b1 = new JButton("Click");
```

6. The button is added to the content pane of the frame by using the statement.

```
jf.add(b1);
```

The component is added to the frame's content pane by calling `add()` method on the `JFrame` reference (`jf`). `JFrame` inherits the `add()` method from the AWT class `Container`.

The general form of `add()` method is:

```
Component add(Component comp)
```

where,

`comp` is the object of the component to be added

7. The statement to make the window visible is

```
jf.setVisible(true);
```

The `setVisible()` method is inherited from the AWT `Component` class. If it is set to `true`, the window will be displayed, otherwise not. By default, a `JFrame` is invisible.

**Note:** Prior to JDK 5, the content pane was obtained by calling `getContentPane()` method. However, today, the use of `getContentPane()` is no longer necessary. We can call `add()`, `remove()`, and `setLayout()` directly on `JFrame` as they operate on the content pane automatically.

### 5.4.1 Components of Swing

In this section, we will present an overview of some of the components of Swing.

#### **JApplet**

`JApplet` is a class that represents the Swing applet. It is a subclass of `Applet` class and must be extended by all the applets that use Swing. It provides all the functionalities of the AWT applet as well as support for menu bars and layering of components. Whenever we require to add a component to it, the component is added to the content pane.

The `JApplet` defines the following constructor:

```
JApplet()
```

#### **Label**

A label is an object of `JLabel` class. Some of the constructors defined by `JLabel` class are as follows:

## NOTES

```
JLabel()
JLabel(String string)
JLabel(String string, int align)
```

where,

## NOTES

`string` is the text used for the label.

`align` specifies the horizontal alignment of the text contained in the label, and can have one of the following values: LEFT, RIGHT, CENTER, LEADING or TRAILING.

### Button

A push button is an object of `JButton` class. Some of the constructors defined by `JButton` class are as follows: `JButton()`

```
JButton(Icon icon)
JButton(String string)
```

where,

`string` represents the string used for the button.

`icon` represents the icon used for the button.

**Note:** All the buttons are derived from `AbstractButton` class.

### Toggle Button

Swing provides a variant of push button called toggle button which has two states: pushed and released. When toggle button is pressed for the first time, it remains pressed; it is released only when it is pressed for the second time. This button toggles between pushed and released states. Toggle button is an object of `JToggleButton` class.

Some of the constructors defined by `JToggleButton` class are as follows:

```
JToggleButton()
JToggleButton(String string)
JToggleButton(String string, boolean state)
```

where,

`string` specifies the text.

`state` can have one of the two values: `true`, if the button is initially selected, otherwise `false` (default value).

### Checkbox

A checkbox is an object of `JCheckBox` class. Some of the constructors defined by `JCheckBox` class are as follows:

```
JCheckBox()
JCheckBox(String string)
JCheckBox(String string, boolean isSelected)
```

where,

`string` is the text used as a label for the checkbox.

`isSelected` is used to set the initial state of the checkbox. If it is `true`, the checkbox is checked, otherwise it is unchecked. The default state of the checkbox is `false`.



## Text Field

Text field is an object of `JTextField` class which is a subclass of `JTextComponent`. Some of the constructors defined by `JTextField` class are as follows:

```
JTextField()
JTextField(int cols)
JTextField(String string, int cols)
JTextField(String string)
```

where,

`string` is the initial string contained in the text field.

`cols` is the width of the text field in terms of columns.

## Text Area

The text area creates a multi-line text area. It does not provide scrolling facility; however, scroll bars can be added by adding the `JTextArea` in `JScrollPane` container. A text area is an object of `JTextArea` class which is a subclass of `JComponent`.

Some of the constructors defined by `JTextArea` class are as follows:

```
JTextArea()
JTextArea(String str1)
JTextArea(int rows, int cols)
JTextArea(String str1, int rows, int cols)
```

where,

`str1` is the initial string contained in text area.

`rows` represents the height of the text area or the maximum rows a text area can contain.

`cols` represents the width of the text area, that is, the maximum number of characters each line of the text area can contain.

## Radio Button

Radio buttons are a group of buttons, in which only one radio button can be selected at one time. That is, when you select any one radio button, then the other selected radio button will get deselected automatically. A radio button is an object of `JRadioButton` class.

Some of the constructors defined by `JRadioButton` class are as follows:

```
JRadioButton()
JRadioButton(String string)
JRadioButton(String string, boolean state)
```

where,

`string` specifies the text.

`state` can have one of the two values: `true`, if the button is initially selected, otherwise `false` (default value).

**Note:** Button groups (`javax.swing.ButtonGroup`) are used in combination with radio buttons to ensure that only one radio button is selected at a time.

## NOTES

## NOTES

### Panel

Panel is a container to hold different Swing components. One can add any number of components to a panel and there can be multiple panels in the same frame. It also supports double buffering, which is used in animation to avoid flickering. In double buffering, object is first written to an off-screen memory before display and then switched over to the panel. Flow layout is the default layout for the panel. A panel is an object of class `JPanel` which is present in package `javax.swing`.

Some of the constructors defined by `JPanel` class are as follows:

```
JPanel ()
JPanel (boolean isDoubleBuffered)
JPanel (LayoutManager layout)
JPanel (LayoutManager layout, boolean isDoubleBuffered)
```

where,

`isDoubleBuffered` defines whether the panel is double buffered or not. It can have one of the values: `true` (double buffered) or `false` (not double buffered)

`layout` defines the layout of the panel.

### Scroll Pane

A scroll pane is a container that represents a small area to view other component. If the component is larger than the visible area, scroll pane provides horizontal and/or vertical scroll bars automatically for scrolling the components through the pane. A scroll pane is an object of the `JScrollPane` class which extends `JComponent`.

Some of the constructors defined by `JScrollPane` class are as follows:

```
JScrollPane ()
JScrollPane (Component component)
JScrollPane (int ver, int hor)
JScrollPane (Component component, int ver, int hor)
```

where,

`component` is the component to be added to the scroll pane.

`ver` and `hor` specify the policies to display the vertical and horizontal scroll bar, respectively. Some of the standard policies are:

```
HORIZONTAL_SCROLLBAR_ALWAYS,
HORIZONTAL_SCROLLBAR_AS_NEEDED,
VERTICAL_SCROLLBAR_ALWAYS,
VERTICAL_SCROLLBAR_AS_NEEDED.
```

**Note:** `JScrollPane` is a lightweight container.

### List

A list is an object of `JList` class.

Some of the constructors defined by `JList` class are as follows:

```
JList()
JList(Object[] listdata)
```

where,

`listdata` represents the array of `Object` type that displays the elements.

Example 5.2 illustrates the use of Swing components.

**Example 5.2:** A program to demonstrate the use of the Swing components is as follows:

```
import java.awt.*;
import javax.swing.*;
class FormExample extends JApplet
{
 // Declaring components
 JFrame jf;
 JLabel nm, add, sex, hobbies, languages, blank;
 JButton ok, clear;
 JTextField name;
 JTextArea address;
 JRadioButton m, f;
 JCheckBox cric, bad, dance, music;
 ButtonGroup bg;
 JList list;
 JScrollPane scr;
 JPanel panel1, panel2, panel3, panel4, panel5, main;
 FormExample()
 {
 String lang[] = {"C", "C++", "Java", "Pascal",
 "Fortran", "COBOL"}; // list items
 jf = new JFrame("Demo of Swing components");
 // Instantiation of components
 ok = new JButton("OK");
 clear = new JButton("Clear");
 m = new JRadioButton("Male", true);
 f = new JRadioButton("Female");
 bg = new ButtonGroup();
 sex = new JLabel(" Sex");
 hobbies = new JLabel(" Hobbies");
 blank = new JLabel();
 cric = new JCheckBox("Cricket", true);
 bad = new JCheckBox("Badminton");
 dance = new JCheckBox("Dance");
 music = new JCheckBox("Music");
 languages = new JLabel(" Languages Known");
 list = new JList(lang); // Adding items to the list
 list.setVisibleRowCount(3); // Three items visible
 scr = new JScrollPane(list,
 ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
 ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
 // Adding list to scroll pane
 nm = new JLabel("Name");
 add = new JLabel("Address");
 name = new JTextField(20); // Textfield have 20
 / /
 columns
```

## NOTES

## NOTES

```
address = new JTextArea(3,20); // Textarea have 3 rows
//
and 20 columns
panel1 = new JPanel();
panel2 = new JPanel(new GridLayout(1,3)); // set
//layout of panel2 to GridLayout
panel3 = new JPanel();
panel4 = new JPanel(new GridLayout(1,2)); //set
//layout of panel4 to GridLayout
panel5 = new JPanel();
main = new JPanel(new GridLayout(5,1)); //hold all
//the five panels

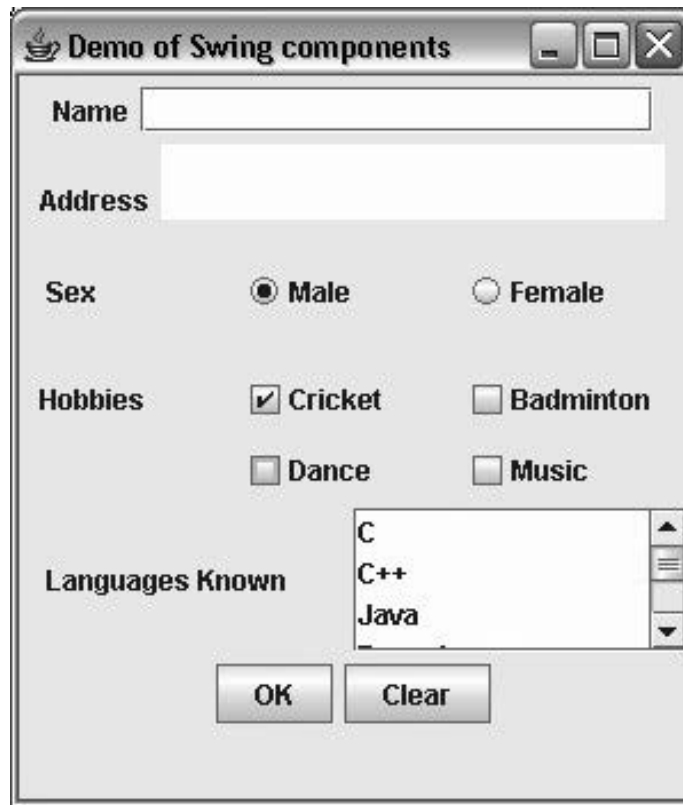
/* Adding radio buttons to the button group so that only
one radiobutton is active at a time
*/
bg.add(m);
bg.add(f);

// Adding components to the corresponding panels
panel1.add(nm);
panel1.add(name);
panel1.add(add);
panel1.add(address);
panel2.add(sex);
panel2.add(m);
panel2.add(f);
panel3.setLayout(new GridLayout(2,3));
panel3.add(hobbies);
panel3.add(cric);
panel3.add(bad);
panel3.add(blank);
panel3.add(dance);
panel3.add(music);
panel4.add(languages);
panel4.add(scr, BorderLayout.NORTH);
panel5.add(ok);
panel5.add(clear);

// Adding all the panels to the main panel
main.add(panel1);
main.add(panel2);
main.add(panel3);
main.add(panel4);
main.add(panel5);

jf.add(main); // adding main panel to the frame
jf.setDefaultCloseOperation(jf.EXIT_ON_CLOSE);
jf.setSize(300, 350);
jf.setVisible(true);
}
public static void main(String str[])
{
 FormExample f = new FormExample();
}
}
```

The output of the program is:



*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

## NOTES

### Check Your Progress

1. Define the term image class.
2. Differentiate between fixed-length records and variable-length records.
3. What are the disadvantages with fixed length records?
4. State about the image class.
5. What is MediaTracker?
6. Define the term JDBC.
7. What is API?
8. Name the container that is not a top-level container.
9. What is JApplet?

## 5.5 JAVA BEANS

The Java Beans architecture is based on a component model enabling the developers to create small software units that they call components. These smaller components can be assembled and integrated to form large components like applets, applications etc. These core components have the characteristics of reusability and are self-contained and are known as beans.

## NOTES

The official definition of a bean, as given in the Java Beans specification, is: “A bean is a reusable software component based on Sun’s JavaBeans specification that can be manipulated visually in a builder tool.”

These software components or beans are developed, but run and reused everywhere as per the requirements. There are various softwares like NetBeans, JBuilder, JDK, etc. which can help you out to unleash the power of beans – reusability. The JavaBeans component architecture is backed upon by the set of APIs called the JavaBeans API specification that deals with the internal sophistication of this model.

Java Beans are dynamic components and also very crucial implementations to be dealt with. Dynamic in the sense that you can easily change their nature and customize them as per your need. For this the design mode of a builder tool can be used for visual manipulation. Simply, choose a bean from the toolbox, drag and drop it into a form, restructure its appearance and behaviour, define its interaction with other beans, and combine it with other beans into an applet, application, or a new bean.

### Some Bean Facts

1. A Java Bean is a reusable and self-contained software component that inherits the basic implementations from sun’s java bean specification.
2. A Java Bean is not only reusable but also customizable and can be easily manipulated visually in a builder tool.
3. The Builder tool is an application development tool which helps you to build new beans or reuse the pre-existing ones to develop an application visually.
4. Java Beans help to create simple components first which can be then reutilised to produce comparatively some more complex components or applications using the builder tools like NetBeans etc.
5. These builder tools are moreover like IDEs that determine a bean’s features like its properties, methods, and events. The process of discovering these facts about the beans is called introspection. There are two ways for Beans support introspection:
  - (i) Just adhere to some specific rules, known as design patterns, when naming bean features. The Introspector class checks the beans for these design patterns explore out all the bean features.
  - (ii) And the other method is to explicitly provide all the basic information like property, method, and event information with a related bean information class. A bean information class implements the Bean Info interface. A BeanInfo class explicitly lists those bean features that are to be exposed to application builder tools.
6. The appearance and behaviour of a bean determines its properties. These properties can be manipulated and changed later during the design using some builder tools. Builder tools introspect on a bean to discover its properties and expose those properties for manipulation and further changes.

7. Beans show various properties which can be customized at design time either by using property editor tools like NetBeans, etc. or by using more advanced bean customizers.
8. Events are the mechanism of communication between two or more beans. Beans follow the event delegation model where there is a listener and the source for every event. It requires a bean builder tool to act as interface and create appropriate registration of the source (event generator), the listener (event delegator) and the event handler so that the event gets properly managed.
9. The literary meaning of “Persistence” is long lasting. So, this enables the beans to save and restore their state later when needed. After bean manipulation, beans properties get changed from the initial ones, so this need to be saved. Why? So, that if there is any error or the bean component is not performing as desired then it could be reverted back to its initial state. Java Object Serialization is used to support persistence in java beans.
10. A bean’s methods are similar to the Java methods, and can be called from other beans or a scripting environment as required. By default all public methods are exported.
11. On the basis of functionality and purpose, all the beans vary in nature from each other. Some of the bean implementations you might have probably met while programming are:
  - (a) GUI (Graphical User Interface)
  - (b) Non-Visual Beans, such as a Spelling Checker
  - (c) AnimationApplet
  - (d) Spreadsheet Application

### **Benefits of using Java Beans**

As discussed beans are reusable, self-contained software components. So, there are a lot of benefits that a programmer can utilize to create a good software. Some of the benefits have been discussed below:

- The java beans are java components hence have the “Write once and run anywhere” property.
- Beans are platform independent and hence can work in different local platforms.
- Beans follow the message passing concept and therefore can easily capture the events sent by other objects and vice versa thereby enabling effective object communication.
- The application developer can add, manipulate and configure the properties, events and methods of the bean with the help of bean auxiliary software during design time without creating any havoc at runtime.
- The beans are persistent in nature and hence their configuration settings can be saved in persistent storage and restored later during any contingency or failure.

### **NOTES**

## NOTES

### Power and Capabilities of Java Bean

A Bean can be too simple or too complex, whatever be the design and/or configurations, but both are equally powerful in implementations. A bean can be designed to cater the need of performing a simple function, like developing a spell checking component to be used in a document, or for performing a complex function, such as forecasting some performance metrics in business models. A Bean may be visible to an end user.

### Builder Tool

Builder tools are the softwares that helps a developer to examine a Java Bean by a process known as Introspection and exposes the features of the Java Bean so that it can be visually manipulated. Apart from this, it also maintains a list of all JavaBeans that are currently available in the development environment. Amongst the various features, besides customization of the beans behaviour, its appearances and properties, builder tools can be used to convert the Bean into applets, application, servlets and composite components (e.g. a JFrame), and connect different other components to the event of the Bean or vice versa.

### Points to Remember While Devising a JavaBeans

Java Beans are special java classes and hence while you are writing any of them you should always remember the following points, without which your beans will just get spoilt.

1. A Java Bean class should always be declared “Public”.
2. A Java Bean should implement the “Serializable interface”.
3. A Java Bean should have a “No-argument constructor/default constructor”.
4. A Java Bean class should be derived from “Javax.swing.JComponent or java.awt.Component” class if it is to be visually manipulated.

The java.beans package consists of the classes and interfaces that help you to create JavaBeans with proper implementations.

The Java Bean components can exist in either of the following three phases of development: viz. 1. Construction phase, 2. Build phase and 3. Execution phase.

Java Beans support the standard component architecture that comprises of components like- properties, events, methods, and persistence.

### Composition of a Java Bean

A Java Bean is composed of the followings: Properties, Methods, and Events. So, you should know the basics of all these components of the java beans.

#### 1. Properties

Java Bean properties are analogous to instance variables of a class in java. A bean *property* is a named attribute of a bean that can affect its behaviour or appearance. It means if you wish to change the behaviour or the appearance of the Java Beans,



then you can use this attribute. Examples of bean properties include colour, label, font, font size, and display size.

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

## 2. Methods

The Java Bean methods are just the same as normal **Java methods**. **Every property should have accessor (get) and mutator (set) method**. All Public methods can be identified by the introspection and also there does not exist specific naming standard for these methods. These are also known as getter and setter methods.

## NOTES

### Events

Events are similar to the events of Swing/AWT event handling. They also follow the same event delegation model like Swing components.

### The Java Bean Component Specification

The Java Beans component specification deals with following properties and capabilities like customization of Bean properties, persistence, Beans to Beans communications, etc.

1. **Beans Customization:** This is the ability of Java Bean to allow its properties to be modified or altered in build phase and/or execution phase.
2. **Beans Persistence:** It can be defined as the JavaBeans ability to save its state to disk or storage device and restore the saved state when it is reloaded.
3. **Beans Communication:** This can be defined as the ability of Java Bean to communicate about change in its properties to other JavaBeans or the container so that they can respond accordingly later on as per modifications.
4. **Introspection:** This is the ability of a Java Bean to allow an external application to examine it and know the properties, methods, and events supported by it so that it can be changed or used as needed.

### Features of a Java Bean

1. Java bean supports “Introspection”; it helps a builder tool analyse how a bean works by inspecting its features.
2. Java bean provides support for “Customization” so that a developer can easily customise the appearance and behaviour of a Java Bean and create a new from the existing one.
3. Java beans supports “Events” that act as a glue to bind two or more beans.
4. Java beans have “Properties” that can be used for both customization and for programmatic use.
5. Java beans support “Persistence”, which is a mechanism of storing the bean state initially and restoring later on need.

## NOTES

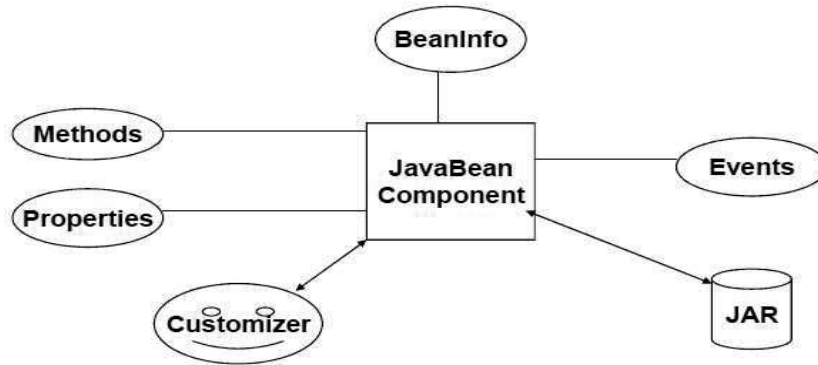


Fig. 5.16 Components of Java Bean

### Beans Development Kit

Beans Development Kit (BDK) is a development environment that can be utilized to create, configure, and test Java Beans. It has the following basic features:

- It is a graphical user interface helping a bean developer to create, configure, and test Java Beans.
- It helps the bean developer to introspect the Java Bean properties and also helps to manage and link multiple Java Beans in an application.
- It consists of a set of sample Java Beans which can be used for application development.
- It also provides a base for utilising and associating the pre-existing events with sample Java Beans to understand the event model.

### Identifying BDK Components

- Execute the run.bat file of BDK to start the BDK development environment.

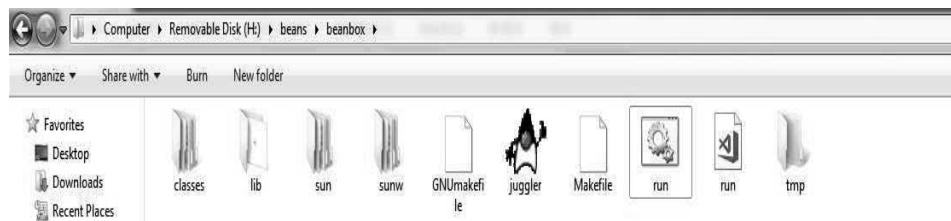
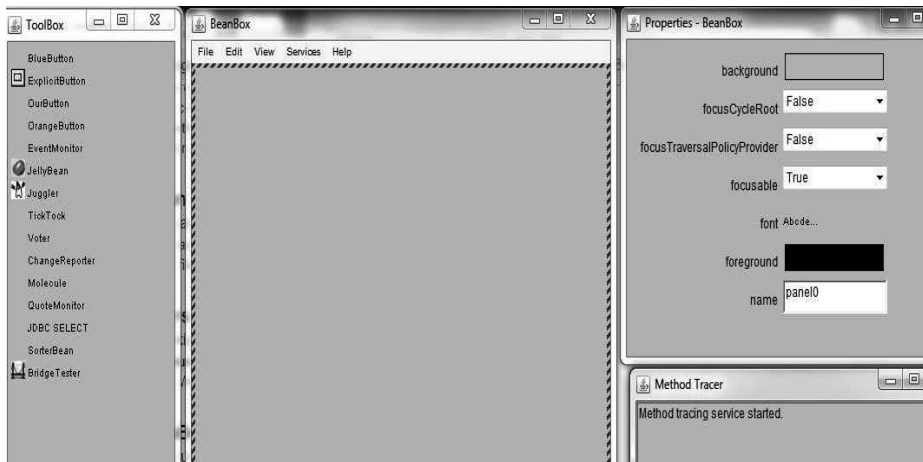


Fig. 5.16 Run.Bat File of BDK to Start the BDK Development Environment

A BDK development environment is composed of the following modules:

- ToolBox
- BeanBox
- Properties
- Method Tracer

## NOTES



*Fig. 5.17 A Complete View of Bean Development Kit*

Now, you will learn about the different components of the bean development kit.

1. **ToolBox Window:** A window that consists of the lists of sample JavaBeans of BDK available to be reused to build up an application.

Below is the figure that shows the **ToolBox** window with various pre-built beans:



*Fig. 5.18 Toolbox*

2. **BeanBox Window:** It is a workspace for creating the layout of Java Bean application. Figure given below shows the **BeanBox** window:



*Fig. 5.19 BeanBox*

## NOTES

- 3. Properties Window:** This window displays all the exposed properties of a Java Bean. You can modify Java Bean properties in the properties window. Given below is the **Properties** window showing various properties of a BeanBox:

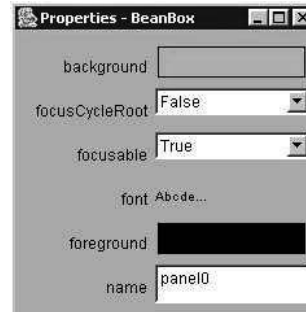


Fig. 5.20 BeanBox Properties

- 4. Method Tracer Window:** Method tracer window displays the debugging messages and method calls for a Java Bean application.

The following figure shows the **Method Tracer** window:

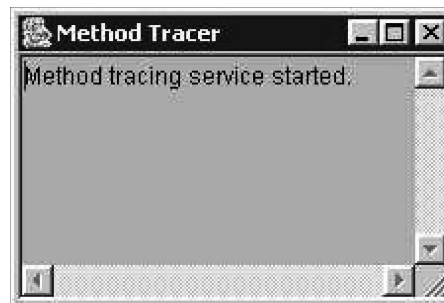


Fig. 5.21 Method Tracer

### Steps to Develop a User-Defined Java Bean

Follow the following steps to develop a user defined Java bean.

- 1. Create a directory for the new bean**

Create a directory/folder like C:\Beans

- 2. Create bean source file - MyBean.java**

```
import java.awt.*;
public class MyBean extends Canvas
{
 public MyBean()
 {
 setSize(70, 50);
 setBackground(Color.green);
 }
}
```

- 3. Compile the source file(s)**

C:\Beans>Javac MyBean.java

- 4. Create a manifest file**

## Manifest File

The manifest file for a Java Bean application contains a list of all the class files that make up a Java Bean. The entry in the manifest file enables the target application to recognize the Java Bean classes for an application. For example, the entry for the MyBean Java Bean in the manifest file is as shown:

**Note:** write that 2 lines code in the notepad and save that file as MyBean.mf

```
Manifest-Version: 1.0
Name: MyBean.class
Java-Bean: true
```

The rules to create a manifest file are:

- Press the Enter key after typing each line in the manifest file.
- Leave a space after the colon.
- Type a hyphen between Java and Bean.
- No blank line between the Name and the Java-Bean entry.

## 5. Generate a JAR file

Syntax for creating jar file using manifest file

```
C:\Beans>jar cfmMyBean.jarMyBean.mfMyBean.class
```

## 6. Start BDK

Go to->

```
C:\bdk1_1\beans\beanbox
```

Click on **run.bat** file. When we click on run.bat file the BDK software automatically started.

```
Manifest-Version: 1.0
Name: MyBean.class
Java-Bean: true
```

## 7. Load Jar file

Go to Beanbox->File->Load jar. Here we have to select our created jar file when we click on ok, ourbean (userdefined) MyBean appear in the ToolBox.

## 8. Test our created user defined bean

Select the MyBean from the ToolBox when we select that bean one + simple appear then drag that Bean in to the Beanbox. If you want to apply events for that bean, now we apply the events for that Bean.

## Steps to develop a user defined Java bean

1. Create a directory for the new bean
2. Create the java bean source file(s)
3. Compile the source file(s)
4. Create a manifest file
5. Generate a JAR file

## NOTES

6. Start BDK
7. Load Jar file
8. Test.

## NOTES

### 5.5.1 What is Java Beans

JavaBeans is a technology that allows one to build complex systems from reusable software component (called bean). It provides an architecture how these components can communicate with one another.

A bean is a software component that is designed to be reused in different environments. These components are written in the Java language like other Java programs and run in Java Virtual Machine (JVM).

Some of the advantages of JavaBeans are as follows:

- A bean has all benefits of Java's 'Write-Once, Run-Anywhere' paradigm.
- The properties events, and methods of a bean which are exposed to another application can be controlled.
- The configuration settings of a bean can be saved in the persistent storage, and restored any time later.

#### Bean Architecture

A bean consists of three general purpose interfaces listed as follows:

- **Properties:** These are the attributes that determine the bean's internal state. When values are assigned to the properties, they determine the appearance and behaviour of the component. The properties are equivalent to data fields of object in Java except that they must be declared as `private`. There are two types of properties, namely, *simple* and *indexed*.
- **Events:** Beans can communicate with objects by generating events. The events are generated upon happening of some action at some specific point of time. The generated events may be sent to other objects.
- **Methods:** Methods are the operations through which interaction with a bean can be done. A bean's methods are similar to Java methods. Some methods are special and deal with properties and events.

#### Introspection

Introspection is the automatic process of analyzing a bean's properties, events and methods. It is the important feature of JavaBeans API as it allows other application such as design tool, to obtain information about a component. The process of introspection is used at run-time as well as design time.

There are two approaches by which a developer can indicate which of its properties, events and methods are exposed. In the first approach, JavaBean's design patterns are used. The second approach is to implement the `BeanInfo` interface which gives explicit information about its associated bean.

## Design Patterns for Properties

The JavaBean's design patterns specify special methods called accessor methods; getter and setter methods to get and set a property, respectively. According to the design patterns, the getter method names must start with the `get` prefix and setter methods must start with the `set` prefix.

### Simple Properties

A simple property has a single value. The design patterns for simple property are:

```
public TP getNP ()
public void setNP (TP arg)
```

where,

NP is the property name

TP is the property type

For example, consider a simple property named `length` along with its getter and setter methods, namely, `getLength()` and `setLength()` respectively as given here.

```
private double length;

public double getLength ()
{
 return length;
}

public void setLength (double l)
{
 length=l;
}
```

### Indexed Properties

An indexed property holds the multiple values. The design patterns for an indexed property are:

```
public TP getNP (int index); //indexedgetter
public void setNP (int index, TP value); //indexedsetter
public TP [] getNP (); //arraygetter
public void setNP (TP values []); //arraysetter
```

where,

NP is the property name

TP is the property type

For example, consider an indexed property named `list` along with its getter and setter methods as shown here.

```
private double list [];

public double getList (int index)
{
 return list [index];
}
```

## NOTES

## NOTES

```
}
public void setList (int index, double value)
{
 list[index]=value;
}
public double [] getList ()
{
 return list;
}
public void setList (double [] values)
{
 list=new double [values.length];
 System.arraycopy (values, 0, list, 0, values.length);
}
```

### Design Patterns for Events

The design patterns to add a listener for the specified event are:

```
public void addTListener (TListener eventListener) //first

public void addTListener (TListener eventListener) throws
java.util.TooManyListenersException //second
```

where,

T is type of the event

The first method is used to register many listeners for notification of the events, that is, to multicast an event.

The second method of throws `TooManyListenersException`. Thus, it is used to unicast an event, that is, only one listener can register to the event.

The design pattern for removing a listener is:

```
public void removeTListener (TListener eventListener)
```

### Methods and Design Patterns

There is no design pattern for naming non-property methods. The public methods of a bean are exposed automatically by the introspection mechanism.

### Using the BeanInfo Interface

The introspection mechanism is provided by JavaBeans API architecture. The `BeanInfo` interface defined in `java.beans` package provides a set of methods that gives the explicit information about the associated beans. The name of the class that implements `BeanInfo` interface must be `beanName (BeanInfo)` where `beanName` is the name of the bean. The methods provided by `BeanInfo` interface are as follows:

- `EventSetDescriptor [] getEventSetDescriptors ()`:  
This method is used to determine a bean's events. The `getEventSetDescriptors` method returns an array of `EventSetDescriptor`.



- `PropertyDescriptor[] getPropertyDescriptors():` This method is used to determine the properties of a bean. The `getPropertyDescriptors` method returns an array of `PropertyDescriptor`.
- `MethodDescriptor[] getMethodDescriptors():` This method is used to determine the public methods of a bean. The `getMethodDescriptors` method returns an array of `MethodDescriptor`.

**Note:** A static method named `getBeanInfo(beanName)` of the `Introspector` (in API reference documentation) class is used to get detailed information about a specific bean.

### Bound and Constrained Properties

A bound property is a property that generates an event when its value is changed. A bean provides this notification by following methods.

```
public void addPropertyChangeListener (PropertyChangeListener p)
{
 changes.addPropertyChangeListener (p);
 //provides notification when an attempt is made to change the //bound
 property
}

public void removePropertyChangeListener (PropertyChangeListener
p)
{
 changes.removePropertyChangeListener (p); //to remove
 //listener
}
```

**Note:** The `PropertyChangeListener` is an interface that is declared in `java.beans` package.

A constrained property is a property that notifies other objects or components when attempts are made to change their values. When other component is not agreed to the change of the property it throws `PropertyVetoException`. Following are the methods provided by constrained properties.

```
public void addVetoableChangeListener (VetoableChangeListener v)
{
 vetos.addVetoableChangeListener (v);
}

public void removeVetoableChangeListener (VetoableChangeListener
v)
{
 vetos.removeVetoableChangeListener (v);
}
```

### NOTES

## NOTES

### Persistence

As stated earlier, a bean is a reusable software component that can be used in other applications. To use it in other applications and to make it portable, its state should be saved to the non-volatile storage from where it can be retrieved later. The process of saving the current state of bean is referred to as persistence.

Persistence can be achieved by serializing a bean. A bean can be serialized by implementing the `java.io.Serializable` interface. This interface is a marker interface that makes serialization automatic. In case a bean does not implement the `java.io.Serializable` interface, serialization must be provided explicitly by implementing the `java.io.Externalizable` interface.

**Note:** Object serialization is a method of converting an object into data stream.

### Customization

During the development process, it is required to configure the bean according to requirement. To help other developers configure the bean, `customizer` can be provided. A `customizer` is a step-by-step guide that provides the instructions to be followed for using the bean in specific context. Online documentation can also be useful to configure the bean.

### Bean Conventions

A bean is a simple Java class that follows some basic conventions which are as follows:

- A bean should implement the `Serializable` interface.
- A bean should have a no-argument constructor.
- A bean should be public and provide getter and setter methods for accessing its properties.

### A Simple Bean

```
package com.start.bean.test //package statement

public class SimpleBeanExample implements java.io.Serializable
{
 /* Properties */
 private String name = null;
 private int eid = 0;

 /* Empty constructor */
 public SimpleBeanExample ()
 {}

 /* Getter and Setter methods */
 public String getName ()
 {
 return name;
 }
 public void setName (String s)
 {
 name = s;
 }
}
```

```
}
public int getEid()
{
 return eid;
}
public void setEid(int i)
{
 eid = i;
}
}
```

## NOTES

The explanation of this program is as follows:

- The statement `package com.start.bean.test` is the package statement.
- The statement `public class SimpleBeanExample implements java.io.Serializable` defines a class named `SimpleBeanExample` that implements the `Serializable` interface to provide persistence to the bean. Note that the `Serializable` interface does not contain any method.
- The statements `private String Ename = null` and `private int Eid = 0` declare properties of the bean. The properties are declared private, thus, are not accessible directly by other classes.
- The statement `public SimpleBeanExample() {}` creates an empty constructor. For behaving Java class as a bean, this is the basic requirement.
- The methods `getEname()` and `setEname()` are the getter and setter methods respectively of property `Ename`.

Like any other Java class file, compile the bean. Upon compilation, `SimpleBeanExample.class` file will be created.

### 5.5.2 JAR Files and Introspection

JAR (Java ARchive) file allows you to efficiently deploy a set of classes and their associated resources. JAR file makes it much easier to deliver, install, and download. It is compressed. The files of a Java Bean application are compressed and grouped as JAR files to reduce the size and the download time of the files.

- The syntax to create a JAR file from the command prompt is:  
`jar <options> <file_names>`
- The `file_names` is a list of files for a Java Bean application that are stored in the JAR file.

The various options that you can specify while creating a JAR file are:

- c: Indicates the new JAR file is created.
- f: Indicates that the first file in the `file_names` list is the name of the JAR file.
- m: Indicates that the second file in the `file_names` list is the name of the manifest file.

## NOTES

- t: Indicates that all the files and resources in the JAR file are to be displayed in a tabular format.
- v: Indicates that the JAR file should generate a verbose output.
- x: Indicates that the files and resources of a JAR file are to be extracted.
- o: Indicates that the JAR file should not be compressed.
- m: Indicates that the manifest file is not created.

```
C:\Users\DevilsEye>jar
Usage: jar (ctxui)[lvfmn0Me] [jar-file] [manifest-file] [entry-point] [-c]
les ...
Options:
 -c create new archive
 -t list table of contents for archive
 -x extract named (or all) files from archive
 -u update existing archive
 -v generate verbose output on standard output
 -f specify archive file name
 -m include manifest information from specified manifest file
 -n perform Pack200 normalization after creating a new archive
 -e specify application entry point for stand-alone application
 bundled into an executable jar file
 -0 store only; use no ZIP compression
 -M do not create a manifest file for the entries
 -i generate index information for the specified jar files
 -C change to the specified directory and include the following files
 If any file is a directory then it is processed recursively.
 The manifest file name, the archive file name and the entry point name :
 specified in the same order as the 'm', 'f' and 'e' flags.

Example 1: to archive two class files into an archive called classes.jar
jar cvf classes.jar Foo.class Bar.class
Example 2: use an existing manifest file 'mymanifest' and archive all the
files in the foo/ directory into 'classes.jar':
jar cvfm classes.jar mymanifest -C foo/ .
```

*Fig. 5.22 JAR utility as Implemented on Command Prompt*

### Introspection

Introspection is a technique which helps to retrieve the vital information about the Java beans like what features does it have(bean properties), how it will behave(bean methods) and what it can really be used for (bean events) etc. Introspection is actually an analysis of bean capabilities so as to know what the bean is capable to do. Implementation of introspection is an automatic process in bean builder tool to introspect which properties, methods, and events a bean supports. It is an eminent process, without which you can never imagine to unleash the real power of a Java Bean and utilize it further.

### BDK Introspection

To automatically analyse the properties of a Java Bean, BDK Introspection is done. It allows the builder tool to analyse the functioning of the bean. It can be defined as the mechanism that allows the bean classes to publish their operations and properties that they support along with a meta-mechanism which leads to the discovery of such mechanisms. Introspection can be defined as the technique of obtaining information about bean properties, events and methods or in short, it is the analysis of bean capabilities as what it can do.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed by the builder tool.

1. First one uses the mechanism of naming conventions that allows the introspection mechanisms to infer information about a Bean.
2. Secondly, an additional class which contains the basic information is provided that explicitly provides this information to the builder tools.

## NOTES

## 5.6 BASIC SERVLET API

The Java Servlet API is not included in the core Java framework and is Standard Java Extension API. It comprises two packages `javax.servlet` and `javax.servlet.http`, which contain classes and interfaces that are used to create servlets. Each servlet must implement the `Servlet` interface included in the package `javax.servlet`. Servlets are created by extending one of the two classes: `javax.servlet.GenericServlet` or `javax.servlet.http.HttpServlet`. Both the classes implement `Servlet` interface.

### The `javax.servlet` Package

Some of the interfaces included in the `javax.servlet` package are listed in Table 5.2.

*Table 5.2 The Interfaces Included in `javax.servlet` Package*

Interface	Description
<code>Servlet</code>	It defines methods that must be implemented by all servlets.
<code>ServletConfig</code>	It represents servlet configuration object that allows servlets to receive initialization parameters.
<code>ServletContext</code>	It allows servlet to write to the log files and access information related to the environment in which servlet is executing.
<code>ServletRequest</code>	It is used to define an object that can be used to read data included in a client request
<code>ServletResponse</code>	It is used to define an object that can be used to develop response for a client.
<code>Filter</code>	It is used to define an object that is used to perform filtering tasks on either the request to the resource or on the response from a resource.
<code>RequestDispatcher</code>	It is used to define an object that is used to accept request from the client and send it to any resource on the server.

Some of the classes included in this package are listed in Table 5.3.

*Table 5.3 The Classes Included in `javax.servlet` Package*

Class	Description
<code>GenericServlet</code>	It is used to create servlet.
<code>ServletInputStream</code>	It provides an input stream that is used to read data from a client request.
<code>ServletOutputStream</code>	It provides an output stream that is used to send data to the client.
<code>ServletContextEvent</code>	It is an event class for notifications related to the changes to the servlet context of a Web application.
<code>ServletException</code>	It defines a general exception that servlet can throw when an error occurs.
<code>UnavailableException</code>	It defines an exception that a servlet can throw to indicate that it is unavailable.

## The javax.servlet.http Package

Some of the interfaces included in the javax.servlet.http package are listed in Table 5.4.

### NOTES

**Table 5.4** The Interfaces Included in javax.servlet.http Package

Interface	Description
HttpSession	It enables data of a session to be read and written.
HttpServletRequest	It extends the ServletRequest interface to read data form an HTTP request.
HttpServletResponse	It extends the ServletResponse interface to send data as response to the HTTP request.
HttpSessionBindingListener	It provides information to an object whether it is bound to or unbound from a session.

Some of the classes included in this package are listed in Table 5.5.

**Table 5.5** The Classes Included in javax.servlet.http Package

Class	Description
Cookie	It is used to create cookie that allows state information to be stored on the client machine.
HttpServlet	It is an abstract class that provides methods for handling HTTP requests and responses.
HttpSessionEvent	It represents event notifications related to the changes to sessions within a Web application.
HttpSessionBindingEvent	It provides information whether listener is bound to or unbound from a session and whether a session attribute has changed.

### Creating and Executing Servlets

Servlets are platform independent and can work with almost all the Web servers. They can be executed on any Web server that supports the servlet API. Some of the Web servers having built-in support for Java servlets are listed in Table 5.6.

**Table 5.6** Web Servers Supporting Java Servlets

Product	Vendor
Tomcat server	Apache
Java Web server	Sun Microsystems
Enterprise server	Netscape
Zeus Web server	Zeus Technology
Tengah application server	Weblogic
Sun Web server	Sun Microsystems

In this unit, we will be using the Tomcat server to explain the steps required to execute the servlet in Windows environment. Assume that the default location of Tomcat 6.0 is as follows:

C:\ProgramFiles\Apache Software Foundation\Tomcat 6.0\

The steps to create and execute the servlet are as follows:

1. Create a `.java` source file containing the code for creating servlet and save it with the name, say `Sample.java`.
2. Compile this source file, as a result `Sample.class` file will be created.
3. Copy the `Sample.class` file in the directory under the `Webapps` directory of Tomcat. The path for the directory is as follows:

```
C:\ProgramFiles\ApacheSoftwareFoundation\Tomcat
6.0\Webapps\MyApp\WEB-INF\classes
```

4. Add the name and mapping of this servlet in the `Web.xml` file. The path of this file is as follows:

```
C:\ProgramFiles\ApacheSoftwareFoundation\Tomcat
6.0\Webapps\MyApp\WEB-INF
```

Add the following statements in the section, which defines the servlets:

```
<servlet>
 <servlet-name>Sample</servlet-name>
 <servlet-class>Sample</servlet-class>
</servlet>
```

Also, add the following statements in the section which defines the servlets mappings.

```
<servlet-mapping>
 <servlet-name>Sample</servlet-name>
 <url-pattern>/servlet/Sample</url-pattern>
</servlet-mapping>
```

5. Start the Tomcat. To start the Tomcat, click the **Start** menu, point to **All Programs**, point to **Apache Tomcat 6.0** and then click **Configure Tomcat**. This displays the **Apache Tomcat Properties**. Click the **Start** button.
6. Start a Web browser and request for the servlet. Note that the HTML code can be included in the servlet code or a separate HTML file can be used. If the HTML code is included in the servlet code, enter the following URL in the address bar:

```
http://localhost:8080/MyApp/servlet/Sample
or
http://127.0.0.1:8080/MyApp/servlet/Sample
```

If separate HTML file is used for the HTML code, enter the following URL in the address bar:

```
http://localhost:8080/MyApp/Sample.html
or
http://127.0.0.1:8080/MyApp/Sample.html
```

Here, `127.0.0.1:8080` is an IP address of the local system. Note that the `.html` file must be saved in the `MyApp` directory.

The output of the servlet will be displayed in the display area of browser.

**Note:** Tomcat must be running in the background before executing the servlet.

## NOTES

## NOTES

### Using HttpServlet Class

The `HttpServlet` class is the commonly used class for developing servlets, which can handle HTTP requests. It is the subclass of `GenericServlet` class. Some of the methods defined in this class are listed in Table 5.7.

*Table 5.7 Some of the Method of HttpServlet Class*

Method	Description
<code>doDelete()</code>	It allows servlet to handle HTTP DELETE request.
<code>doGet()</code>	It allows servlet to handle HTTP GET request.
<code>doPost()</code>	It allows servlet to handle HTTP POST request.
<code>doHead()</code>	It allows servlet to handle HTTP HEAD request.
<code>doOptions()</code>	It allows servlet to handle HTTP OPTIONS request.
<code>doPut()</code>	It allows servlet to handle HTTP PUT request.
<code>doTrace()</code>	It allows the servlet to handle HTTP TRACE request.
<code>getLastModified()</code>	It returns the time when the <code>HttpServletRequest</code> object was last modified.
<code>service()</code>	It receives the HTTP request and sends it to corresponding method defined in this class.

The servlet created by using this class overrides one of these methods. When servlet is invoked, the information is passed to the `service()` method, which in turn determines the type of request sent and invokes the appropriate method. For example, if request made is of GET type, the `doGet()` method is invoked by the `service()` method. Similarly, if request is of POST type, `doPost()` method is invoked by the `service()` method and so on. The `doGet()` and `doPost()` methods are the commonly used methods. In this section, you will learn about the usage of these two methods.

#### The `doGet()` Method

The `doGet()` method is invoked by server through `service()` method to handle a HTTP GET request. This method also handles HTTP HEAD request automatically as HEAD request is nothing but a GET request having no body in the code for response and only includes request header fields. To understand the working of `doGet()` method, consider a sample program to define a servlet for handling the HTTP GET request.

**Example 5.8:** A program to define a servlet for handling HTTP GET request

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletGetExample extends HttpServlet
{
 public void doGet (HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException
 {
 PrintWriter out = res.getWriter();
 String login = req.getParameter("loginid");
 String password = req.getParameter("password");
 out.println("Your login ID is: ");
 out.println(login);
 out.println("Your password is: ");
 }
}
```



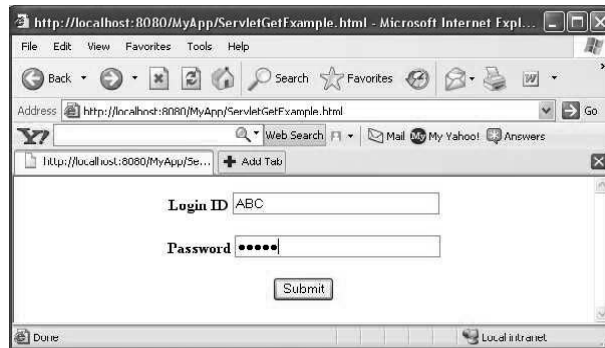
```
 out.println(password);
 out.close();
 }
}
```

In this example, the `doGet()` method of `HttpServlet` class is overridden to handle the HTTP GET request. The two parameters passed to the `doGet()` method are `req` and `res`, the objects of `HttpServletRequest` and `HttpServletResponse` interfaces respectively. The `req` object allows to read data provided in the client request and the `res` object is used to develop response for the client request.

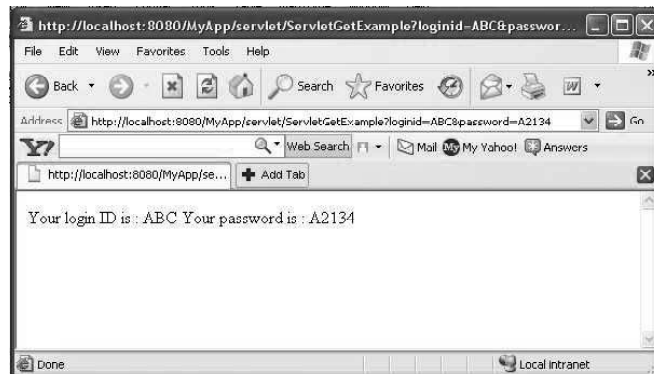
The corresponding HTML code for this servlet is as follows:

```
<HTML>
<BODY>
<CENTER>
<FORMNAME="Form1"
ACTION="http://localhost:8080/MyApp/servlet/ServletGetExample">
Login ID<INPUT TYPE="text" NAME="loginid" SIZE="30">
<P>
Password<INPUT TYPE="password" NAME="password" SIZE="30">
</P>
<P>
<INPUT TYPE=submit VALUE="Submit">
</P>
</BODY>
</HTML>
```

This HTML code creates a Web page containing a form; see the screen.



Enter the required data and press the submit button on the Web page. The browser will display the response generated dynamically by the corresponding servlet (see the following screen).



## NOTES

## NOTES

Note that the `getParameter()` method of `HttpServletRequest` interface is used to retrieve data attached to the URL sent to the server. For example, consider the URL in the address bar of the afore mentioned screenshot. The string appearing to the right of the question mark, known as the **query string**, contains the parameters for the HTTP GET request.

### The `doPost()` Method

Like `doGet()` method, the `doPost()` method is invoked by server through `service()` method to handle HTTP POST request. The `doPost()` method is used when large amount of data is required to be passed to the server, which is not possible with the help of `doGet()` method. In `doGet()` method, parameters are appended to the URL; whereas, in `doPost()` method parameters are sent in separate line in the HTTP request body. The `doGet()` method is mostly used when some information is to be retrieved from the server and the `doPost()` method is used when data is to be updated on server or data is to be submitted to the server. To understand the working of `doPost()` method, consider a sample program to define a servlet for handling the HTTP POST request.

**Example 5.9:** A program to define a servlet for handling HTTP POST request

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletPostExample extends HttpServlet
{
 public void doPost(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException
 {
 PrintWriter out = res.getWriter();
 String loginid = req.getParameter("loginid");
 String password = req.getParameter("password");
 out.println("Your Login ID is: ");
 out.println(loginid);
 out.println("Your Password is: ");
 out.println(password);
 out.close();
 }
}
```

This servlet can be tested by using HTML code

### 5.6.1 MIME Content Types

**Content** type is also known as **MIME (Multipurpose Internet Mail Extension)** type. It is a **HTTP (HyperText Transfer Protocol)** header that provides the description about what the user is sending to the browser.

MIME is an Internet standard that is used for extending the limited capabilities of e-mail by allowing the insertion of sounds, images and text in a message.

The features provided by MIME to the e-mail services are as given below:

- It supports the non-ASCII (American Standard Code for Information Interchange) characters.

- It supports the multiple attachments in a single message.
- It supports the attachment which contains executable audio, images and video files, etc.
- It supports the unlimited message length.

### List of Content Types

There are many content types. The commonly used content types are given below:

- Text/HTML (HyperText Markup Language)
- Text/Plain
- Application/Microsoft WORD
- Application/ Microsoft EXCEL
- Application/JAR
- Application/PDF
- Application/OCTET STREAM
- Application/X-ZIP
- Images/JPEG
- Images/PNG
- Images/GIF
- Audio/MP3
- Video/MP4, etc.

The MIME types file in the **config** directory contains mappings between the Multipurpose Internet Mail Extensions (MIME) types and file extensions. For example, the MIME types file maps the extensions **.html** and **.htm** to the type **Text/HTML**:

```
type=text/html exts=htm,html
```

When the Web Server receives a request from a client, it uses the MIME type mappings to determine the kind of resource that is requested.

MIME types are defined by the following three attributes:

- Language (**lang**)
- Encoding (**enc**)
- Content Type (**type**)

At least one of these attributes must be present for each type. The most commonly used attribute is '**type**'. The server frequently considers the **type** when deciding how to generate the response to the client. The **enc** and **lang** attributes are rarely used. The default MIME types file is **mime.types**.

### Determining the MIME Type

During the **ObjectType** stage in the request handling process, the server determines the MIME type attributes of the resource requested by the client. The user can use different SAFs to determine the MIME type. The most commonly

### NOTES

## NOTES

used SAF (Store-And-Forward) is type-by-extension, which tells the server to look up the MIME type according to the requested resource file extension in the MIME types table. The MIME types table is stored in a MIME type file.

The directive in **obj.conf** that tells the server to look up the MIME type according to the extension is given as,

**ObjectType fn=type-by-extension**

If the server uses a different SAF, such as force type for determining the type, then the MIME types table is not used for that particular request.

---

## 5.7 CORBA CONNECTIVITY IN JAVA

---

CORBA or Common Object Request Broker Architecture is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

The Internet is a constant reminder that we live in a highly connected world, where people and products communicate quickly and easily. This connected world depends on embedded systems ranging from massive central office switches and routers and compact cell phones. Connectivity is, however, not just geographical distribution. Many embedded products are themselves distributed systems consider a small local network to provide connectivity of processors with in the system. Distribution is often used to increase the performance scalability and availability of embedded systems. The increasing customer demand for distributed applications, coupled within advances in the enabling technologies of networking hardware and high speed processors, has made distribution a mandatory ingredient of many embedded systems. You will learn here the key issues consider for a distribution infrastructure, and how Object Request Broker (ORB) can extend operating system capability to, provide a bridge between the embedded and the connected worlds. By using commercial solutions, designers can focus on application development rather than infrastructure, thus accelerating product delivery.

### Java and CORBA Connectivity

**Java SE Leverage CORBA:** Common Object Request Broker Architecture (CORBA) technology is the open standard for heterogeneous computing. CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment, and a very robust platform. By combining the Java platform with CORBA and other key enterprise technologies, the Java Platform is the ultimate platform for distributed technology solutions.

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency, Java provides the implementation transparency.

**Object Request Broker (ORB):** An Object Request Broker (ORB) is part of the Java Platform Standard Edition (SE), since version 1.3. The ORB is a runtime component that can be used for distributed computing using IIOP communication.

CORBA is a standard architecture for distributed object systems as it permits a distributed, heterogeneous collection of objects to interoperate. The CORBA is a specific standard defined by the OMG (Object Management Group) and describes an architecture, interfaces, and protocols that distributed objects use for interacting with each other.

Part of the CORBA standard is the Interface Definition Language (IDL), which is an implementation-independent language for describing the interfaces of remote objects. The OMG comprises over 700 companies and organizations, including almost all the major vendors and developers of distributed object technology, including platform, database, and application vendors as well as software tool and corporate developers.

The Java IDL is an implementation of the standard IDL-to-Java mapping and is provided by Sun in version 1.3 of Java 2 and is compliant with CORBA 2.x specification. Java IDL provides an Object Request Broker or ORB. The ORB is a class library that enables low-level communication between Java-IDL applications and other CORBA-compliant applications.

The key components that frame the CORBA architecture include the following:

1. Interface Definition Language (IDL) states how CORBA interfaces are defined.
2. Object Request Broker (ORB) is responsible for all interactions between remote objects and the applications that use them.
3. Portable Object Adaptor (POA) is responsible for object activation/deactivation, mapping object IDs to actual object implementations.
4. 'Naming Service' is a standard service in CORBA that helps the remote clients in finding remote objects on the networks.
5. Internet Inter-ORB Protocol (IIOP) is an Internet communications protocol that runs on distributed platforms.

### **Interface Definition Language (IDL)**

An Interface Description Language or Interface Definition Language (IDL), is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language. IDLs describe an interface in a language-independent way, enabling communication between software components that do not share one language, for example, between those written in C++ and those written in Java.

IDLs are commonly used in remote procedure call software. In these cases the machines at either end of the link may be using different operating systems and computer languages. IDLs offer a bridge between the two different systems.

Software systems based on IDLs include Sun's ONC RPC or Open Network Computing (ONC) / Remote Procedure Call (RPC), Distributed

## **NOTES**

## NOTES

Computing Environment (DCE) of the Open Group, IBM's System Object Model, the Object Management Group's (OMG) CORBA, which implements OMG IDL (Interface Definition Language) and an IDL based on DCE/RPC (Distributed Computing Environment / Remote Procedure Call) and Data Distribution Service, Mozilla's XPCOM (Cross Platform Component Object Model), Microsoft's Microsoft RPC (which evolved into COM (Component Object Model) and DCOM (Distributed Component Object Model)), Facebook's Thrift and WSDL (Web Services Description Language) for Web services.

Open Network Computing (ONC) / Remote Procedure Call (RPC), commonly known as Sun RPC is a remote procedure call system. ONC was originally developed by Sun Microsystems in the 1980s as part of their Network File System (NFS) project. Distributed Component Object Model (DCOM) is a proprietary Microsoft technology for communication between software components on networked computers. DCOM, which originally was called 'Network OLE', extends Microsoft's COM, and provides the communication substrate under Microsoft's COM+ application server infrastructure. The addition of the 'D' to COM was due to extensive use of DCE/RPC (Distributed Computing Environment/Remote Procedure Calls) – more specifically Microsoft's enhanced version, known as MSRPC.

Cross Platform Component Object Model (XPCOM) is a cross-platform component model from Mozilla. It is similar to Microsoft Component Object Model (COM) and Common Object Request Broker Architecture (CORBA). It features multiple language bindings and Interface Description Language (IDL) descriptions; thus programmers can plug their custom functions into the framework and connect it with other components. The most prominent usage of XPCOM is within the Firefox web browser. Many of its internal components interact via XPCOM interfaces.

The Web Services Description Language (WSDL) is an XML-based (Extensible Markup Language-based) interface description language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service, also referred to as a WSDL file, which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. Therefore, its purpose is roughly similar to that of a type signature in a programming language.

Typically, an Interface Definition Language (IDL) is a language that is used to define the interface between a client and server process in a distributed system. Each IDL also has a set of associated IDL compilers, one per supported target language. An IDL compiler compiles the interface specifications, listed in an IDL input file, into source code (e.g., C/C++, Java) that implements the low-level communication details required to support the defined interfaces. IDL can also be used to populate an implementation repository, which other programs can use to look up information on an interface at runtime. This is necessary when a program, such as a debugger or interface browser, does not have access to an application's IDL file.

One advantage of an interface definition language is that it does not contain any mechanism for specifying computational details. The stubbed out routines,

generated by the IDL compiler, must be filled in with implementation specific details provided by the application developer. Thus, an IDL clearly enforces the separation of a distributed application's interface from its implementation.

Another advantage of an IDL is the productivity enhancement provided by the IDL compiler. Without the IDL compiler, the developer would have to custom craft the network protocol for each distributed application developed, which would be both time consuming and error prone. The IDL compiler frees the developer from these low-level details, thus providing more time for the developer to focus on the application's core functionality.

IDL provides a basic set of atomic data types (e.g., **long**, **double**, **string**) and a mechanism, **struct**, for combining these atomic types into more complex structures. The **typedef** command can be used to create a new name for a data type.

A client and server are typically on different machines, so message passing must be used to pass parameters, including any return value, between them. Thus, the client sends a request message to the server object, which sends a reply message back (if required). However, it is unnecessary for both messages to contain all parameters, so CORBA IDL introduces constructs to deal with this.

**IDL Compilation:** An IDL compiler takes as input an IDL file, with its associated interface definitions, and produces a set of output files for both the client and server application. The names and number of generated files varies from one development environment to another. The client side code consists of a set of routines that transparently access the server. On the server side, the IDL compiler generates a skeleton framework that must be fleshed out with application specific implementation details.

### Internet Inter-ORB Protocol (IIOP)

Internet Inter-ORB Protocol (IIOP) is a transport level protocol used by both Remote Method Invocation (RMI) over IIOP and Common Object Request Broker Architecture (CORBA).

Principally, the Internet Inter-ORB Protocol (IIOP) is an Internet communications protocol that runs on distributed platforms. Using this protocol, software programs written in different programming languages and running on distributed platforms can communicate over the Internet.

IIOP, a part of the CORBA standard, is based on the client/server computing model, in which a client program makes requests of a server program that waits to respond to client requests. With IIOP, you can write client programs that communicate with your site's existing server programs wherever they are located without having to understand anything about the server other than the service it performs and its address, called the Interoperable Object Reference, IOR, which comprises the server's port number and IP (Internet Protocol) address. An IP address is a unique address that identifies a device on the Internet or a local network. IP is the set of rules governing the format of data sent via the Internet or local network.

Between the ORBs (Object Request Brokers), communication proceeds by means of a shared protocol, the 'IIOP' or the Internet Inter-ORB Protocol.

## NOTES

## NOTES

IIOP is based on the standard TCP/IP (Transmission Control Protocol/Internet Protocol) which works across the Internet and defines how CORBA-compliant ORBs pass information back and forth.

Like CORBA and IDL, the IIOP standard is defined by OMG or the Object Management Group. IIOP allows clients using a CORBA product from one vendor to communicate with objects using a CORBA product from another vendor thus permitting interoperability, which is one of the goals of the CORBA standard. The ORB provided with Java IDL supports one optional service, i.e., the ability to locate objects by name.

### Java RMI over IIOP

Java Remote Method Invocation (RMI) over Internet Inter-ORB Protocol (IIOP) or simply the 'RMI-IIOP' technology is part of the Java Platform Standard Edition (Java SE). The RMI programming model enables the programming of CORBA servers and applications via the `rmi` API (Application Programming Interface). The user can select to work completely within the Java programming language using the Java Remote Method Protocol (JRMP) as the transport or to work with other CORBA-compliant programming languages using the Internet Inter-ORB Protocol (IIOP).

RMI-IIOP utilizes the Java CORBA Object Request Broker (ORB) and IIOP, therefore the user can write all of the programming code in the Java programming language, and use the `rmic` compiler to generate the code necessary for connecting the user's applications via the Internet Inter-ORB Protocol (IIOP) to others written in any CORBA-compliant language. To work with CORBA applications in other languages, IDL can be generated from Java programming language interfaces using the `rmic` compiler with the `-idl` option. To generate IIOP stubs and connect classes, use the `rmic` compiler with the `-iiop` option.

### RMI over IIOP

Java™ Remote Method Invocation (RMI) provides a simple mechanism for distributed Java programming. RMI over IIOP (RMI-IIOP) uses the Common Object Request Broker Architecture (CORBA) standard Internet Inter-ORB Protocol (IIOP) to extend the base Java RMI to perform communication. This allows direct interaction with any other CORBA Object Request Brokers (ORBs), whether they were implemented in Java or another programming language.

## 5.7.1 Working CORBA System

The Common Object Request Broker Architecture (CORBA) is a specification that helps to integrate heterogeneous systems that have different hardware, operating systems, networks and different programming languages.

### CORBA Components

There are several CORBA components available. Typical CORBA components are as follows:

- **Object Request Broker (ORB):** It is the core component of CORBA that acts as communication channel between client and server. With the



help of ORB, client application can invoke a method on a server object anywhere across the internet. The client application is totally unaware of the location of the server object; neither it has any knowledge of server systems hardware, operating systems, nor the programming language in which server object is implemented. ORB keeps track of object's location, type of platform, programming language, types of requests and responses and exceptions between client objects and server objects. It taps the method call from the client application and gives information about server objects (see Figure 5.23) as it maintains repository of server objects.

## NOTES

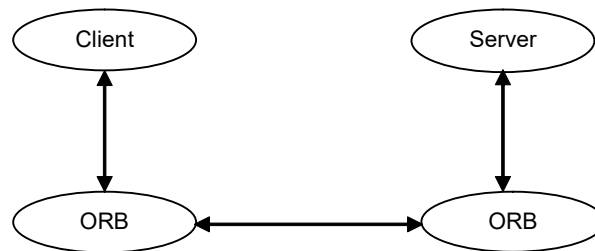


Fig. 5.23 Client-Server Communication through ORB

- **CORBA Services:** ORB uses the CORBA services to make CORBA implementations. Some of the CORBA services are as follows:
- **Life Cycle Service:** This service is used to define operations for creating, deleting and moving objects in a CORBA application.
- **Persistence Service:** This service provides capability to store objects persistently. The objects may be stored on servers like files, relational database and object database.
- **Naming Service:** Naming service helps the CORBA client in finding the objects on the network. A reference to the object name is requested by clients using a naming Service.
- **Event Service:** This service enables a client or object to send messages in the form of event objects. The event object may be sent to more than one receiver.
- **Security Service:** It supports access control, confidentiality and authorization there by provides security to the distributed objects.
- **Query Service:** This service provides query operations for the objects.
- **CORBA Facilities:** CORBA facilities define application level services. These services include firewalls, data interchange, work flow, business object frame works, etc.
- **Internet Inter-ORB Protocol (IIOP):** This protocol specifies the specifications about how ORBs communicate, how messages are sent and how parameters and return values are marshaled in remote object invocation.
- **Interface Definition Language (IDL):** It provides a language construct with which interfaces can be defined independent of any programming language. The IDL language constructs are similar to C and C++ syntax but they cannot be compiled directly into the binary code. The interface

## NOTES

specifications defined by IDL language constructs are compiled by special tool called IDL compiler. The IDL compiler translates the general constructs to the specific programming language constructs. For example, an IDL to Java compiler converts the IDL specifications to the Java files. In the same way, IDL to C++ compiler converts the IDL specifications to C++ files. Nowadays, CORBA vendors provide tools to translate the IDL specification to C, C++, Java, SmallTalk, ADA and COBOL programming languages.

### 5.7.2 Simple CORBA Service

Although the implementation of an ORB is complex, how it operates is fairly simple. Each node on the network has an ORB library or core that is linked into the application at build time as shown in Figure 5.24. The stubs and skeletons created as part of the IDL translation processes isolate the clients and servers from the actual location of the objects with which they are interacting, thus providing location transparency; for example, to communicate with an object using CORBA, the source object only needs a reference to the target object it wants to communicate with.

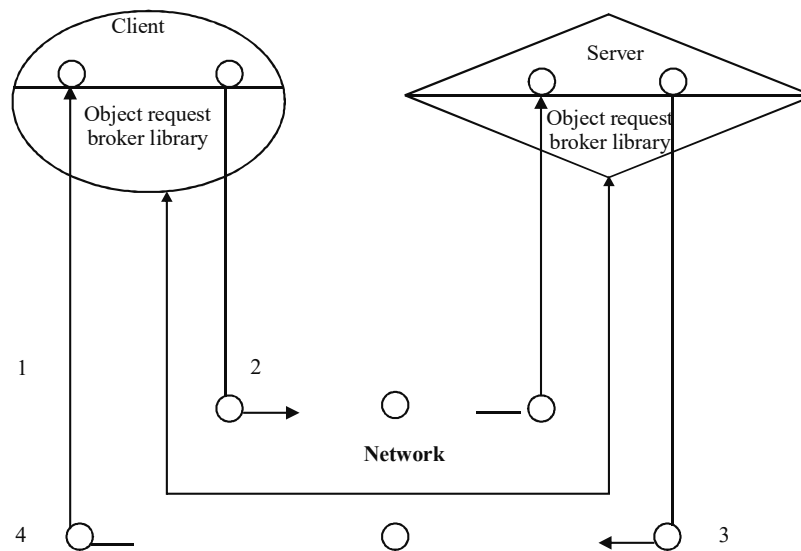


Fig. 5.24 Operations of ORB

#### Check Your Progress

10. Give the definition of Java Beans architecture.
11. Define the term introspection.
12. What is a JAR file?
13. Define basic servlet API.
14. Define the term MIME content.
15. What is CORBA?

---

## 5.8 ANSWERS TO ‘CHECK YOUR PROGRESS’

---

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

1. The `java.awt` package contains an abstract class, `Image`. This class provides the mechanism to handle operations that are related to an image.
2. All the records in a file of fixed-length record are of same length and number, size of each field is fixed whereas, exact length of field is not fixed in advance in variable length records. Hence, special separator is used to determine the start and end of each field within the record.
3. A fixed length record may contain some optional fields and space is reserved for optional fields as well. These fields store null value if no value is supplied which results in wastage of lots of memory space. Also, it is difficult to delete a record as deletion of a record leaves blank space in between the two records.
4. The `Image` class is used to load and display images. To load an image the `getImage ()` method of the `Image` class is used and to display the image the `drawImage ()` method of the `Graphics` class is used.
5. The `java.awt.MediaTracker` class is a general utility that tracks the loading of a number of images or other media types for the users. Basically, the `java.awt.MediaTracker` is a utility class that simplifies the problems when the user have to wait for one or more images to be loaded completely before they are actually displayed. A `MediaTracker` monitors the loading of an image or a group of images and helps the user to check those either periodically or to wait until the loading is completed.
6. JDBC (Java DataBase Connectivity) defines an API (Application Program Interface) designed to support basic SQL (Structured Query Language) functionality independent of any specific SQL implementation. This means the focus is on executing SQL statements and retrieving their results. JDBC is an international standard for programming access to SQL databases. It was developed by JavaSoft, a subsidiary of Sun Microsystems.
7. API is the abbreviation of Application Program Interface, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together. Simply, it provides a set of rules for performing a particular task but in case of JDBC API the task is connect to the database. They are completely invisible to users and Web surfers. Their primary role is to provide a channel for applications to work with each other.
8. `JPanel` is not a top-level container.
9. `JApplet` is a class that represents the Swing applet. It is a subclass of `Applet` class and must be extended by all the applets that use Swing. It provides all the functionalities of the AWT applet as well as support for menu bars and layering of components. Whenever we require to add a component to it, the component is added to the content pane.

### NOTES

## NOTES

10. The Java Beans architecture is based on a component model enabling the developers to create small software units that they call components. These smaller components can be assembled and integrated to form large components like applets, applications etc. These core components have the characteristics of reusability and are self-contained and are known as beans.
11. Introspection is the automatic process of analyzing a bean's properties, events and methods. It is the important feature of JavaBeans API as it allows other application, such as design tool, to obtain information about a component. The process of introspection is used at run-time as well as design time.
12. JAR (Java ARchive) file allows you to efficiently deploy a set of classes and their associated resources. JAR file makes it much easier to deliver, install, and download. It is compressed. The files of a Java Bean application are compressed and grouped as JAR files to reduce the size and the download time of the files.
13. The Java ServletAPI is not included in the core Java framework and is Standard Java Extension API. It comprises two packages `javax.servlet` and `javax.servlet.http`, which contain classes and interfaces that are used to create servlets. Each servlet must implement the `Servlet` interface included in the package `javax.servlet`. Servlets are created by extending one of the two classes: `javax.servlet.GenericServlet` or `javax.servlet.http.HttpServlet`. both the classes implement `Servlet` interface.
14. Content type is also known as MIME (Multipurpose Internet Mail Extension) type. It is a HTTP (HyperText Transfer Protocol) header that provides the description about what the user is sending to the browser. MIME is an Internet standard that is used for extending the limited capabilities of e-mail by allowing the insertion of sounds, images and text in a message.
15. CORBA (Common Object Request Broker Architecture) is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

---

## 5.9 SUMMARY

---

- The `java.awt` package contains an abstract class, `Image`. This class provides the mechanism to handle operations that are related to an image.
- All the records in a file of fixed-length record are of same length and number, size of each field is fixed whereas, exact length of field is not fixed in advance in variable length records. Hence, special separator is used to determine the start and end of each field within the record.
- A fixed length record may contain some optional fields and space is reserved for optional fields as well. These fields store null value if no value is supplied which results in wastage of lots of memory space. Also, it is difficult to

delete a record as deletion of a record leaves blank space in between the two records.

- Variable-length records may be used to utilise the memory more efficiently. In this approach, the exact length of the field is not fixed in advance.
- The `Image` class is used to load and display images. To load an image the `getImage ()` method of the `Image` class is used and to display the image the `drawImage ()` method of the `Graphics` class is used.
- The `java.awt.MediaTracker` class is a general utility that tracks the loading of a number of images or other media types for the users. Basically, the `java.awt.MediaTracker` is a utility class that simplifies the problems when the user have to wait for one or more images to be loaded completely before they are actually displayed. A `MediaTracker` monitors the loading of an image or a group of images and helps the user to check those either periodically or to wait until the loading is completed.
- JDBC (Java DataBase Connectivity) defines an API (Application Program Interface) designed to support basic SQL (Structured Query Language) functionality independent of any specific SQL implementation. This means the focus is on executing SQL statements and retrieving their results.
- JDBC is an international standard for programming access to SQL databases. It was developed by JavaSoft, a subsidiary of Sun Microsystems.
- API is the abbreviation of Application Program Interface, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together. Simply, it provides a set of rules for performing a particular task but in case of JDBC API the task is connect to the database. They are completely invisible to users and Web surfers.
- The JDBC driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver.
- SQL (Structured Query Language) is a database computer language designed for managing data in relational database management systems or RDBMS.
- The two components of swing: `JFrame` (top-level container) and `JButton` (component which creates a push button).
- `JApplet` is a class that represents the Swing applet. It is a subclass of `Applet` class and must be extended by all the applets that use Swing. It provides all the functionalities of the AWT applet as well as support for menu bars and layering of components. Whenever we require to add a component to it, the component is added to the content pane.
- Radio buttons are a group of buttons, in which only one radio button can be selected at one time. That is, when you select any one radio button, then the other selected radio button will get deselected automatically.
- Panel is a container to hold different Swing components. One can add any number of components to a panel and there can be multiple panels in the same frame. It also supports double buffering, which is used in animation to

## NOTES

## NOTES

- avoid flickering. In double buffering, object is first written to an off-screen memory before display and then switched over to the panel.
- The Java Beans architecture is based on a component model enabling the developers to create small software units that they call components. These smaller components can be assembled and integrated to form large components like applets, applications etc. These core components have the characteristics of reusability and are self-contained and are known as beans.
  - JavaBeans is a technology that allows one to build complex systems from reusable software component (called bean).
  - Introspection is the automatic process of analyzing a bean's properties, events and methods. It is the important feature of JavaBeans API as it allows other application, such as design tool, to obtain information about a component. The process of introspection is used at run-time as well as design time.
  - JAR (Java ARchive) file allows you to efficiently deploy a set of classes and their associated resources. JAR file makes it much easier to deliver, install, and download. It is compressed. The files of a Java Bean application are compressed and grouped as JAR files to reduce the size and the download time of the files.
  - Introspection is a technique which helps to retrieve the vital information about the Java beans like what features does it have(bean properties), how it will behave(bean methods) and what it can really be used for (bean events) etc.
  - The Java Servlet API is not included in the core Java framework and is Standard Java Extension API.
  - Servlets are platform independent and can work with almost all the Web servers. They can be executed on any Web server that supports the servlet API.
  - The `doGet ()` method is invoked by server through `service ()` method to handle a HTTP GET request.
  - Like `doGet ()` method, the `doPost ()` method is invoked by server through `service ()` method to handle HTTP POST request. The `doPost ()` method is used when large amount of data is required to be passed to the server, which is not possible with the help of `doGet ()` method.
  - Content type is also known as MIME (Multipurpose Internet Mail Extension) type. It is a HTTP (HyperText Transfer Protocol) header that provides the description about what the user is sending to the browser. MIME is an Internet standard that is used for extending the limited capabilities of e-mail by allowing the insertion of sounds, images and text in a message.
  - CORBA (Common Object Request Broker Architecture) is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

- Common Object Request Broker Architecture (CORBA) technology is the open standard for heterogeneous computing.

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

---

## 5.10 KEY TERMS

---

## NOTES

- **Image class:** The java.awt package contains an abstract class, Image. This class provides the mechanism to handle operations that are related to an image.
- **Graphics class:** Provides different methods to draw and fill various shapes.
- **JDBC (Java DataBase Connectivity):** JDBC (Java DataBase Connectivity) defines an API (Application Program Interface) designed to support basic SQL (Structured Query Language) functionality independent of any specific SQL implementation.
- **JDBC driver:** The JDBC driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver.
- **SQL (Structured Query Language):** SQL (Structured Query Language) is a database computer language designed for managing data in relational database management systems or RDBMS.
- **Beans customization:** This is the ability of Java Bean to allow its properties to be modified or altered in build phase and/or execution phase.
- **Beans persistence:** It can be defined as the JavaBeans ability to save its state to disk or storage device and restore the saved state when it is reloaded.
- **JavaBeans:** It is a technology that allows one to build complex systems from reusable software component (called bean). It provides an architecture how these components can communicate with one another.
- **Introspection:** This is the ability of a Java Bean to allow an external application to examine it and know the properties, methods, and events supported by it so that it can be changed or used as needed.
- **Servlets:** These are small programs written in Java, which are loaded and executed by Web server.
- **MIME type:** Content type is also known as MIME (Multipurpose Internet Mail Extension) type.
- **CORBA:** CORBA (Common Object Request Broker Architecture) is a standard architecture for distributed object systems.

---

## 5.11 SELF ASSESSMENT QUESTIONS AND EXERCISES

---

### Short-Answer Questions

1. Write the two types of image files.
2. What is file format?

## NOTES

3. What do you mean by `ImageObserver`?
4. Write the application of JDBC.
5. Elaborate on the term `ResultSet`.
6. What do you mean by JDBC metadata API?
7. State about the radio button.
8. Define the term Bean.
9. What do you understand by persistence?
10. Elaborate on the term introspection in JAR file.
11. What does `javax.servlet` consist of?
12. Name the list of MIME types.
13. Explain the Interface Definition Language (IDL).
14. Write the application of CORBA.

### Long-Answer Questions

1. Explain how an image is drawn in a frame with the help of a diagram and examples.
2. Briefly explain the fixed-length records and variable-length records with the help of relevant examples.
3. Analyse the image class with the help of an example program.
4. Discuss about the `ImageObserver` and `MediaTracker` with the help of example programs.
5. Explain in detail about JDBC drivers and its types giving examples.
6. Discuss about the JDBC Type 2 and JDBC Type 4 drivers implementation. Support your answer with the help of relevant examples.
7. Describe the components of swing with the help of example programs.
8. Briefly explain the Beans Development Kit (BDK) and its components giving appropriate examples.
9. Discuss about the components of Java Bean with the help of examples.
10. Briefly discuss the constrained and bound properties.
11. Describe the JAR files with the help of examples.
12. Explain the basic servlet API with the help of a program.
13. Differentiate between `doGet()` and `doPost()` methods giving Java programs.
14. Discuss the features of the MIME content with the help of an example program.
15. Briefly explain the Java and CORBA connectivity.
17. Discuss in detail the basic concept of CORBA system with the help of examples.



---

## 5.12 FURTHER READING

---

*Images, JDBC, Java Beans, Servlet API and CORBA Connectivity*

- Balagurusamy, E. 2007. *Programming with Java*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Naughton, Patrick and Herbert Schidt. 1999. *Java 2: The Complete Reference*, 3rd Edition. New Delhi: Tata McGraw-Hill.
- Das, Rashmi Kanta. 2013. *Core Java for Beginners*, 3rd Edition. New Delhi: Vikas Publishing House Pvt. Ltd.
- Schildt, Herbert. 2006. *Java: The Complete Reference*, 7th Edition. New Delhi: Tata McGraw-Hill.
- Hunter, Jason and William Crawford. 2001. *Java Servlet Programming*, 2nd Edition. California: O'Reilly Media.
- Arnold, Ken, James Gosling and David Holmes. 2005. *The Java Programming Language*, 4th Edition. Boston: Addison-Wesley.
- Wigglesworth, Joe and Paula Lumby. 1999. *Java Programming Advanced Topics*, 2 Edition. Boston: Course Technology.
- Deitel, Paul and Harvey Deitel. 2011. *Java: How to Program*, 9th Edition. New Delhi: Prentice-Hall of India.

### NOTES



**NOTES**

---

**NOTES**

---