

Threads AWT
DYNAMIC BINDING
Swing Event Handling
PACKAGE Event Handling
Robust
SECURE AWT
Platform Independent Polymorphism
Applets DYNAMIC BINDING
Threads SECURE Threads
Robust Event Handling
Data Abstraction AWT
Inheritance
Multithreading
Event Handling
AWT
V
Y
Z

JAVA PROGRAMMING

Hari Mohan Pandey

MIC BINDING
Platform Independent
Handling Applets Swing
AGE Threads
Robust
eritance SECURE AWT
ect Oriented Polymorphism
ta Abstraction Robust
MULTITHREADING
IDING Object Oriented Applets Swing
SECURE AWT Platform Independent

Threads Inheritance
Platform Independent
SECURE Applets
DYNAMIC BINDING
AWT
Robust Event Handling
Polymorphism AWT
SECURE Applets Swing Robust

JAVA

Java Programming

Hari Mohan Pandey

Lecturer, Department of Computing
Middle East College of Information Technology
Muscat, Oman

PEARSON

Delhi • Chennai • Chandigarh

Copyright © 2012 Dorling Kindersley (India) Pvt. Ltd.

Licensees of Pearson Education in South Asia

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version. The publisher reserves the right to remove any material present in this eBook at any time.

ISBN 9788131733110

eISBN 9789332510326

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India
Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

Brief Contents

- 1 Introduction to OOPs 1
- 2 Marching Towards Java and Java Bytecodes 9
- 3 Introduction to Operators and Expression in Java 49
- 4 Working with Decision-making Statement in Java 77
- 5 Working with Array in Java 116
- 6 Functions in Java 140
- 7 Classes and Objects 157
- 8 Inheritance in Java 207
- 9 Packages and Interfaces 240
- 10 String and String Buffer 265
- 11 Exception Handling 284
- 12 Threads in Java 309
- 13 Streams and Files 336
- 14 Applet and Graphics Programming 361
- 15 Event Handling 405
- 16 Working with AWT 446
- 17 Working with Layout 518
- 18 The Collection Framework 534
- 19 Basic Utility Classes 590
- 20 Networking in Java 641
- 21 Miscellaneous Topics: JNI, Serialization and RMI 684
- 22 Working with Images 702
- 23 Introduction to Swing 729
- 24 Introduction to Virtual Machine and API Programming 782

This page is intentionally left blank.

Contents

Preface	xvii
Acknowledgements	xxi

1 INTRODUCTION TO OOPS

1

1.1 Structured Introduction	1
1.1.1 Sequence Structure	1
1.1.2 Loop or Iteration Structure	1
1.1.3 Decision Structure	1
1.2 Procedural Programming	2
1.3 Programming Methodology	2
1.4 Object-oriented Programming	4
1.5 Basic Concepts of OOPs	4
1.6 Characteristics of OOPs	6
1.7 Advantages of OOPs	6
1.8 Object-oriented Languages	7
1.9 Object-based Languages	7

2 MARCHING TOWARDS JAVA AND JAVA BYTECODES

9

2.1 Introduction: What is Java?	9
2.2 Evolution of Java	9
2.3 Main Benefits of Using Java	10
2.4 Key Features of Java	10
2.5 Java Character Set	12
2.5.1 Java Tokens	12
2.6 Variables	15

2.7	How to Put Comments in Java?	15
2.8	Data Types in Java	16
2.9	Structure of a Java Program	17
2.10	Your First Program in Java	18
	<i>2.10.1 Compiling and Running the Program (DOS Based)</i>	19
2.11	Java is Free Form	23
2.12	Programming Examples	23
2.13	Reading Using Scanner	32
2.14	Command Line Arguments	33
2.15	Using Constants	35
2.16	Bytecode and JVM	35
	<i>2.16.1 What is the Java Virtual Machine? Why is it Here?</i>	35
	<i>2.16.2 What are Java Bytecodes?</i>	35
	<i>2.16.3 Virtual Parts</i>	36
	<i>2.16.4 The Proud, the Few, the Registers</i>	36
	<i>2.16.5 The Method Area and the Program Counter</i>	37
	<i>2.16.6 The Java Stacks and Related Registers</i>	37
	<i>2.16.7 The Garbage-collected Heap</i>	37
	<i>2.16.8 What is in a Class File?</i>	37
2.17	Why Understand Bytecode?	38
	<i>2.17.1 How to Get Bytecode?</i>	38
	<i>2.17.2 Implementation of Bytecode Works</i>	40
2.18	The Java Platform	44
2.19	Java, Internet and WWW	44
2.20	JDK Tools	45
2.21	Ponderable Points	46

3 INTRODUCTION TO OPERATORS AND EXPRESSION IN JAVA

49

3.1	Introduction	49
	<i>3.1.1 Binary Operators</i>	49
	<i>3.1.2 Unary Operators</i>	49
	<i>3.1.3 Expressions</i>	50
3.2	Arithmetic Operators	50
3.3	Relation and Ternary Operator	53
3.4	Logical Operator	56
	<i>3.4.1 Logical AND (&&)</i>	56
	<i>3.4.2 Logical OR</i>	57
	<i>3.4.3 Logical NOT (!)</i>	58
3.5	Assignment Operator	58
3.6	Increment (++) and Decrement (--) Operator	59

- 3.7 Bitwise Operators 61
 - 3.7.1 *Bitwise AND (&)* 61
 - 3.7.2 *Bitwise OR (|)* 62
 - 3.7.3 *Bitwise XOR (^)* 63
 - 3.7.4 *I's Complement (~)* 63
 - 3.7.5 *Left Shift Operator (<<)* 64
 - 3.7.6 *Right Shift Operator (>>)* 65
 - 3.7.7 *Right Shift with Zero Fill (>>>)* 66
- 3.8 The InstanceOf Operator 67
- 3.9 The Comma Operator 68
- 3.10 The `sizeof()` Operator 69
- 3.11 Precedence of Operator 69
- 3.12 Type Conversion and Typecasting 70
- 3.13 Mathematical Functions 71
- 3.14 Scope and Lifetime 73
- 3.15 Ponderable Points 74

4 WORKING WITH DECISION-MAKING STATEMENT IN JAVA

77

- 4.1 Introduction 77
- 4.2 The `if` Statement 77
- 4.3 The `if-else` Statement 79
- 4.4 Nesting of `if-else` Statement 81
- 4.5 `else-if` Ladder 83
- 4.6 Switch-Case Statement 86
- 4.7 Introduction of Loops 89
 - 4.7.1 *The while Loop* 89
 - 4.7.2 *The for Loop* 96
- 4.8 Different Syntaxes of `for` Loop 97
- 4.9 Programming Examples 98
 - 4.9.1 *Nesting of for Loop* 100
 - 4.9.2 *The do-while Statement* 105
- 4.10 `break` and `continue` Statement 108
 - 4.10.1 *The break Statement* 108
 - 4.10.2 *The continue Statement* 110
 - 4.10.3 *Labelled break and continue Statement* 111
- 4.11 Ponderabale Points 113

5 WORKING WITH ARRAY IN JAVA

116

- 5.1 Introduction 116
- 5.2 Creating Arrays in Java 116

5.3	Some Important Points About Array	117
5.4	Initializing 1-D Array	118
5.5	Programming Examples (Part-1)	119
5.6	Two-dimensional (2-D) Array	125
5.7	Three-dimensional (3-D) and Variable Column Length Arrays	134
5.7.1	<i>3-D Array</i>	134
5.8	Ponderable Points	137

6 FUNCTIONS IN JAVA

140

6.1	Introduction	140
6.2	Function Declaration and Definition	140
6.3	No Return Type but Arguments	142
6.3.1	<i>Returning Prematurely from Function</i>	145
6.4	Function with Parameters and Return Type	146
6.5	Recursion	149
6.6	Function Overloading	151
6.7	Ponderable Points	154

7 CLASSES AND OBJECTS

157

7.1	Introduction	157
7.2	Programming Examples	160
7.3	Accessing Private Data	164
7.4	Passing and Returning Objects	167
7.5	Copying Objects	171
7.6	Array of Objects	174
7.7	Static Class Members	177
7.7.1	<i>Static Member Functions</i>	178
7.7.2	<i>Static Data Members</i>	179
7.8	Constructors	181
7.8.1	<i>Constructors with Parameters</i>	182
7.9	Copy Constructor	195
7.10	The <code>this</code> Reference	196
7.11	Garbage Collection and Finalize Method	201
7.12	The Final Keyword Revisited	201
7.12.1	<i>Blank Finals</i>	202
7.12.2	<i>Final Arguments to Methods</i>	203
7.13	Ponderable Points	204

8 INHERITANCE IN JAVA**207**

- 8.1 Introduction 207
- 8.2 Types of Inheritance 207
 - 8.2.1 *Single-level Inheritance* 208
 - 8.2.2 *Multilevel Inheritance* 208
 - 8.2.3 *Multiple Inheritance* 209
 - 8.2.4 *Hierarchical Inheritance* 209
 - 8.2.5 *Hybrid Inheritance* 210
- 8.3 Programming Examples 211
- 8.4 Method Overriding 218
- 8.5 Dynamic Method Dispatch 219
- 8.6 Hierarchical Inheritance Revisited 221
- 8.7 The Super Keyword 223
- 8.8 Constructor and Inheritance 224
- 8.9 Object Slicing 229
- 8.10 Final Class 231
- 8.11 Abstract Class 231
- 8.12 Ponderable Points 237

9 PACKAGES AND INTERFACES**240**

- 9.1 Introduction 240
- 9.2 Package Types 240
 - 9.2.1 *Built-in Packages* 240
 - 9.2.2 *User-defined Packages* 241
- 9.3 Interfaces 252
 - 9.3.1 *Interface Declaration* 253
 - 9.3.2 *Interface Implementation* 253
 - 9.3.3 *Programming Examples* 254
 - 9.3.4 *Extending Classes and Interfaces* 260
- 9.4 Ponderable Points 262

10 STRING AND STRING BUFFER**265**

- 10.1 The String Class 265
- 10.2 Constructors for String Class 266
- 10.3 Methods of String Class 267
- 10.4 Programming Examples (Part 1) 272
- 10.5 The StringBuffer Class 276
- 10.6 Constructor of StringBuffer Class 277

- 10.7 Methods of StringBuffer Class 277
- 10.8 Programming Examples (Part 2) 280
- 10.9 Ponderable Points 281

11 EXCEPTION HANDLING

284

- 11.1 Introduction 284
- 11.2 What is Exception? 284
- 11.3 Basis for Exception Handling 285
- 11.4 Exception-Handling Mechanism 286
- 11.5 Exception Classes in Java 286
 - 11.5.1 Runtime Exception* 287
 - 11.5.2 Checked versus Unchecked Exception* 287
- 11.6 Without Try-Catch 289
- 11.7 Exception Handling Using Try and Catch 290
- 11.8 Creating Your Own Exception 304
- 11.9 Ponderable Points 306

12 THREADS IN JAVA

309

- 12.1 Introduction 309
- 12.2 Creating Threads 309
 - 12.2.1 Extending Thread Class* 310
 - 12.2.2 Thread Methods* 313
 - 12.2.3 Implementing Thread Using Runnable* 318
- 12.3 Thread Priority 320
- 12.4 Thread Synchronization 322
 - 12.4.1 The Synchronized Method* 322
 - 12.4.2 The Synchronized Statement* 324
- 12.5 Thread Communication 324
- 12.6 Suspended and Resuming Threads 327
- 12.7 Daemon Threads 331
- 12.8 Ponderable Points 333

13 STREAMS AND FILES

336

- 13.1 Introduction 336
- 13.2 Working with Files 336
 - 13.2.1 Method of File Class* 337
- 13.3 Types of Streams 344
 - 13.3.1 Character Streams* 345
 - 13.3.2 Programming Examples* 348
 - 13.3.3 Byte Stream* 351

- 13.4 Reading from Console 352
- 13.5 Writing to Console 353
- 13.6 Reading and Writing Files 354
- 13.7 Ponderable Points 359

14 APPLET AND GRAPHICS PROGRAMMING

361

- 14.1 Introduction 361
- 14.2 Applet versus Application Programs 361
- 14.3 The Applet Class 362
- 14.4 Writing the First Applet 363
- 14.5 Life Cycle of an Applet 366
- 14.6 Applet Tag and Applet Parameters 368
- 14.7 Passing Parameter to Applet 369
- 14.8 The Paint, Update and Repaint Method 372
 - 14.8.1 *The Paint Method* 372
 - 14.8.2 *The Update Method* 373
 - 14.8.3 *The Repaint Method* 373
- 14.9 Getdocumentbase() and Getcodebase() Methods 375
- 14.10 The AppletContext Interface 377
- 14.11 Playing Audio in Applet 378
 - 14.11.1 *Playing Audio Using Play Method of Applet Class* 379
 - 14.11.2 *Playing Audio Using AudioClip Interface* 380
- 14.12 Working with Graphics Class 383
 - 14.12.1 *Drawing Lines and Rectangles* 383
 - 14.12.2 *Drawing Ovals and Circles* 384
 - 14.12.3 *Drawing Polygon* 385
 - 14.12.4 *Using Color* 386
 - 14.12.5 *Setting the Drawing Mode* 387
 - 14.12.6 *Programming Example* 391
- 14.13 Working with Fonts 397
- 14.14 The FontMetrics Class 400
- 14.15 Ponderable Points 402

15 EVENT HANDLING

405

- 15.1 Introduction 405
- 15.2 The Delegation Event Model 405
- 15.3 The Event Classes 406
- 15.4 Event Listeners 407
- 15.5 Event Sources 407

15.6	Discussion of Event Classes	407
15.7	The Listeners Interfaces	414
15.8	Programming Example	417
15.8.1	<i>Handling Mouse Events</i>	417
15.8.2	<i>Handling Keyboard Events</i>	424
15.9	Adapter Classes	428
15.10	Nested and Inner Classes	431
15.10.1	<i>Inner Classes in Methods and Scopes</i>	436
15.10.2	<i>Static Nested Class</i>	438
15.10.3	<i>Inner Classes in Event Handling</i>	439
15.10.4	<i>Anonymous Inner Class</i>	441
15.11	Ponderable Points	444

16 WORKING WITH AWT

446

16.1	Introduction to AWT	446
16.2	Structure of the AWT	446
16.3	AWT Window Hierarchy	449
16.3.1	<i>The Component Class</i>	449
16.3.2	<i>The Container Class</i>	449
16.3.3	<i>The Panel Class</i>	450
16.3.4	<i>The Window Class</i>	450
16.3.5	<i>The Frame Class</i>	450
16.3.6	<i>Applet Frame Class</i>	450
16.4	AWT Controls	458
16.4.1	<i>The Button Control</i>	459
16.4.2	<i>The Label Class</i>	464
16.4.3	<i>The Checkbox Class</i>	466
16.4.4	<i>The CheckboxGroup Class</i>	470
16.4.5	<i>The List Class</i>	473
16.4.6	<i>The Choice Class</i>	480
16.4.7	<i>The Scrollbar Class</i>	483
16.4.8	<i>The Textfield Class</i>	488
16.4.9	<i>The TextArea Class</i>	492
16.5	Menu and Menubars	495
16.5.1	<i>Adding Shortcut Key to Menu</i>	501
16.6	Popup Menus	503
16.7	Dialogs	506
16.8	The FileDialog Class	511
16.9	Ponderable Points	515

17 WORKING WITH LAYOUT**518**

- 17.1 Layout and Layout Manager 518
- 17.2 FlowLayout 518
- 17.3 BorderLayout 520
- 17.4 GridLayout 522
- 17.5 GridBagLayout 524
- 17.6 CardLayout 524
- 17.7 Insets 531
- 17.8 Ponderable Points 532

18 THE COLLECTION FRAMEWORK**534**

- 18.1 Introduction 534
- 18.2 Collection Framework 534
 - 18.2.1 *Collection Interface* 535
 - 18.2.2 *The List Interface* 536
 - 18.2.3 *The Set Interface* 537
 - 18.2.4 *The SortedSet Interface* 538
- 18.3 The Collection Class 538
- 18.4 ArrayList and LinkedList Classes 539
 - 18.4.1 *Maintaining the Capacity* 542
 - 18.4.2 *The LinkedList Class* 542
- 18.5 Iterating Elements of Collection 545
 - 18.5.1 *The ListIterator Interface* 545
- 18.6 HashSet and TreeSet Classes 550
 - 18.6.1 *HashSet Class* 550
 - 18.6.2 *TreeSet Class* 551
- 18.7 Working with Maps 554
 - 18.7.1 *The Map Interface* 554
 - 18.7.2 *Map.Entry Interface* 556
 - 18.7.3 *The SortedMap Interface* 556
- 18.8 Working with Map Classes 557
- 18.9 The Comparator Interface 563
- 18.10 Historical Collection Classes 566
- 18.11 Algorithm Support 581
- 18.12 Ponderable Points 586

19 BASIC UTILITY CLASSES**590**

- 19.1 Introduction 590
- 19.2 The StringTokenizer Class 590

19.3	The Date Class	592
19.4	The Calendar Class	595
19.5	The Random Class	599
19.6	Observable Class and Observer Interface	601
19.7	The System Class	606
	<i>19.7.1 Environment Variables/Property</i>	608
19.8	The Object Class	610
	<i>19.8.1 The Clone Method</i>	610
19.9	The Class Class	614
19.10	The Runtime Class	619
19.11	The Process Class	621
19.12	Wrapper Class	624
	<i>19.12.1 The Number Class</i>	625
	<i>19.12.2 The Byte, Short, Integer and Long Class</i>	625
	<i>19.12.3 The Float and Double Class</i>	627
	<i>19.12.4 Methods of Float/Double Class</i>	628
	<i>19.12.5 The Character Class</i>	631
	<i>19.12.6 The Boolean Class</i>	635
19.13	Ponderable Points	637

20 NETWORKING IN JAVA

641

20.1	How Do Computers Talk to Each Other via Internet?	641
20.2	Java.Net Package	642
20.3	Internet Addressing with Java	642
20.4	Socket Fundamentals	645
	<i>20.4.1 Internet Domain Sockets</i>	645
20.5	Sockets in Java	645
	<i>20.5.1 Step-by-step Socket Application Development</i>	646
	<i>20.5.2 The Server</i>	646
	<i>20.5.3 The Client</i>	647
	<i>20.5.4 Thread Chat Server and Client</i>	672
	<i>20.5.5 Datagram</i>	675
20.6	URL	680
	<i>20.6.1 An Example of URL</i>	680
	<i>20.6.2 Creating Relative URLs</i>	681
	<i>20.6.3 The MalformedURLException</i>	682
20.7	Ponderable Points	682

21 MISCELLANEOUS TOPICS: JNI, SERIALIZATION AND RMI

684

21.1	Java Native Interface (JNI)	684
	<i>21.1.1 Role of the JNI</i>	684

21.1.2	<i>An Example of JNI</i>	685
21.1.3	<i>Limitations of Using JNI</i>	688
21.2	Serialization	689
21.2.1	<i>Class and Interfaces for Serialization</i>	689
21.3	RMI	695
21.3.1	<i>Writing RMI Service</i>	696
21.4	Ponderable Points	699

22 WORKING WITH IMAGES

702

22.1	Introduction	702
22.2	Drawing an Image	702
22.3	The ImageObserver Interface	705
22.3.1	<i>Why Override ImageUpdate?</i>	706
22.4	Double Buffering	709
22.5	The MediaTracker Class	713
22.6	Producing Image Data	716
22.7	Consuming Image Data	723
22.8	Ponderable Points	727

23 INTRODUCTION TO SWING

729

23.1	What is Swing?	729
23.2	The JApplet Class	730
23.3	The ImageIcon Class	730
23.4	The JLabel Class	730
23.5	The JButton Class	731
23.6	The JTextField Class	733
23.7	The JCheckBox Class	734
23.8	The JRadioButton Class	736
23.9	The JComboBox Class	738
23.10	The JTabbedPane Class	741
23.11	The JScrollPane Class	743
23.12	The JSplitPane Class	745
23.13	Dialogs	746
23.14	File Selection Dialog	754
23.15	The JColorChooser Class	757
23.16	The JTable Class	759
23.17	The JToolBar Class	760
23.18	The JProgressBar Class	763
23.19	The JSlider Class	767
23.20	The JTree Class	771

- 23.21 Examples of Menus 774
23.22 Ponderable Points 779

24 INTRODUCTION TO VIRTUAL MACHINE AND API PROGRAMMING

782

- 24.1 Introduction 782
24.2 Virtual Machine 783
24.3 Virtual Machine Errors 792
24.4 Instruction Set for Virtual Machine 793
24.5 Invoking Methods Using Virtual Machine 793
24.6 Class Instance and Virtual Machine 796
24.7 Array and Virtual Machine 800
24.8 Switch Statements and Virtual Machine 805
24.9 Throwing and Handling Exceptions, and Virtual Machine 810
24.10 Java Base API 813
 24.10.1 *Java Applet API* 813
24.11 Java Standard Extension API 813
 24.11.1 *Java Security API* 814
 24.11.2 *Java Media API* 814
 24.11.3 *Java Enterprise API* 815
 24.11.4 *Java Commerce API* 816
 24.11.5 *Java Server API* 816
 24.11.6 *Java Management API* 816
24.12 The Java API Package 816
24.13 Ponderable Points 818
Sample Project 821
A1 Java Keyword Reference 850
A2 Creating Java Executables 853
A3 The Jar Tool 859
Index 862

Preface

This book is a comprehensive, hands-on guide to Java programming. People who are already familiar with Java or any other programming language will, of course, have an easier time and can move through the initial chapters quickly.

This book will help readers and students to write sophisticated programs that take full advantages of Java's exciting and powerful object-oriented nature. A sincere effort has been made to cover the fundamentals of every technique that a professional needs to master.

This book lays stress on the new ways of thinking needed to master Java programming, so that even experts in more traditional programming languages can benefit from this book. This innovative approach has been followed because trying to force Java into the framework of older programming languages is ultimately self-defeating, and one cannot take advantage of its power if one continues to think within an older paradigm.

Organization of the Book

The subject matter of this book is divided into 24 chapters. Each chapter has been written and developed with a large number of programs which will clear the core concepts of Java. The book has been written specially for those students who are beginners in the field of programming. In this book, one will find numerous programs instead of just code snippets to illustrate even the basic concepts. The book assumes that the reader is new to the concepts of Java programming language, and hence it has been written in a lucid manner. This book also covers some advanced programming examples which might be useful for experienced programmers. The programming examples given in this book have been well tested. Each chapter contains a number of examples that explain the theoretical as well as the practical concepts. Every chapter is followed by questions to test the student performance and receptivity.

Chapter 1 provides an overview of basic introduction to OOPS. It opens with a discussion on structure programming, procedural programming, different programming methodologies, concepts and current trends in the field of object-oriented programming. It briefly introduces the diversity of OOP, and shows how the technology and architectural trends are driving a convergence in the field of OOP. The chapter develops a layered framework for understanding a wide variety of programming concepts and implementations. Viewing the convergence of the field in this framework, the last portion of the chapter lays out the fundamental concepts of object-oriented languages and object-based languages.

Chapter 2 provides an introduction to Java. It describes a set of motivating applications for software development and its applications that are used throughout the rest of the book. It shows what Java programs look like in the virtual machine environment and, hence, what primitives a system must support. It explains the concepts like why Java is a platform-independent language and how to compile and run the Java programs. At the end of this chapter, the concepts of bytecode and JVM are given.

Chapter 3 describes the basic techniques that good Java programmers use to get code-efficient programs to tune the performance out of the underlying architecture. It provides the understanding of operators and expressions used in Java programming. This chapter also covers the detailed description of precedence of operator, techniques used for type conversion, type casting and mathematical functions.

Chapter 4 takes up the challenge of developing the programming, based on the concepts of decision-making statements like if, if-else, nesting of if-else's and switch-case statement. This chapter also addresses the functionality of loops used in Java.

Chapters 5 and 6 provide a complete understanding of the arrays and functions-based program. This chapter covers the detailed description of 1-D, 2-D and 3-D arrays. Also the concept of function, recursion and overloading of function have been discussed.

Chapter 7 talks about classes and objects. The chapter not only includes the description of classes and objects, but also covers constructors, this reference, garbage collection; finalize method and final at the end of the chapter.

Chapter 8 discusses the concepts of inheritance which plays a major role in the field of programming language. This chapter explains the concepts of method overriding, dynamic method dispatch and the way to use super keyword. It also covers the details of object slicing and abstract class.

Chapter 9 introduces the basic concepts of packages (package types) and different methods used for importing packages. This chapter also includes the details of interfaces, mainly interface declaration, implementation, etc.

Chapters 10 and 11 give detailed descriptions of strings, string buffer and exception handling mechanism. It includes the constructor for string class and method of string class. It also includes the basic methodology for exception handling. Chapter 11 also covers different exception classes used in Java. It also gives guidelines to the programmer about exception handling mechanism without using try and catch block, and with the help of try and catch block. The method of creating own exception is given for the programmer/user at the end of Chapter 11.

Chapter 12 introduces threads in Java and describes the key elements of multithreading. A central question addressed in the chapter is how to create thread and implements. Thread priority, synchronization, communication, suspending and resuming of threads are covered in this chapter. This chapter also includes the concept of daemon thread.

Chapter 13 surveys a wide variety of concepts used with files in Java. This chapter deals with the concepts of different streams used in file handling. This chapter also handles operations like reading from console, writing to console, etc.

Chapter 14 introduces the set of core ideas that are used for Applet and Graphics programming. This chapter covers the life cycle and application areas of applets. This chapter also covers the motivational and widespread application areas of Graphic class.

Chapter 15 covers the detailed description of event handling methodology in Java. This chapter also includes the Delegation event model, Event Listeners, Event Sources. At the end of this chapter, Adapter classes, nested and inner classes have been covered.

Chapters 16 and 17 describe a representative sample for how to work with Abstract Windows Toolkit and the basic structure of AWT. Chapter 16 also covers the details of AWT windows hierarchy, container class and component class. Chapter 17 puts the main focus on the concepts of layouts like Flow Layout, Border Layout, Card Layout, etc. used in Java programming.

Chapters 18 and 19 consider the description of Collection Framework used in Java. Chapter 18 basically covers the description of Collection classes, Array list, Hash set, TreeSet, Maps and Maps classes. Chapter 19 discusses the utility classes like Date, Calendar, Random, System and Runtime used in Java.

Chapter 20 puts the focus on the basic networking supported by Java. This chapter is dedicated to Java.net package, Socket programming and Internet addressing using Java.

Chapter 21 gives a detailed description of the advanced topics like JNI, serialization and RMI. Basically, this chapter is devoted to the description of JNI and how to use the concepts of JNI. It also gives the description of classes and interfaces for serialization.

Chapters 22 and 23 describe the working of images and swing. Chapter 22 basically covers the concepts of ImageObserver interface, double buffering and producing and consuming image data. Chapter 23 includes the details of the following—JApplet class, ImageIcon class, JLabel class, JComboBox class, Dialogs and JTable class.

Chapter 24 explains the concepts of virtual machine and hierarchy of virtual machine. It also covers the detailed description of API programming, different packages and classes used for API programming.

Appendix 1 covers the description of the Java keywords.

Appendix 2 describes how to create Java executable file.

Appendix 3 covers the JAR tools with the details about the command and different options.

Sample Project is given for the students to understand the usability of Java.

My original goal was to make this book a ‘one-stop resource’ but, realistically, Java programming has gotten far too big and far too powerful to be covered in a single book. Nonetheless, if you finish this book, I truly believe that you will be in a position to begin writing commercial-quality Java programs.

All efforts have been made to make this book error free. However, there is possibility that some errors have crept in inadvertently. I would, therefore, be grateful if such oversights are pointed out by the readers and would also be happy to acknowledge suggestions for further improvement of this book.

HARI MOHAN PANDEY

This page is intentionally left blank.

Acknowledgements

First of all, I would like to thank ‘Baba Visvanath’ for his blessings in writing this book.

A large number of people directly or indirectly helped me in completing this book and I would be failing in my duty if I do not say a word of thanks to all those whose sincere advice and support helped to make this book educative, effective and pleasurable.

I would like to thank my father, Dr Vijay Nath Pandey; mother, Madhuri Pandey; sisters, Anjana and Ranjana; and brother, Man Mohan, for their patience and understanding. They never complained about the countless hours I spent seemingly glued to the computer keyboard while creating this text.

I have immense pleasure in expressing my wholehearted gratitude towards Dr T. R. Narayanan, Dean, MECIT, Muscat; Arun Sankarapaa, MECIT, Muscat; Dr Anurag Dixit, Senior Professor, VIT, Vellore; Dr R. R. Sedamkar, Thakur College of Engineering, Mumbai; Dr N. S. Choubey, NMIMS University, Mumbai; Vivek Sharma, MPSTME, NMIMS University, Mumbai; Dr Malvika Sharma, NMIMS University, Mumbai; and Professor B. Salendra, NMIMS University, Mumbai.

I am very much thankful to my students whose conceptual queries have always helped me in digging deep into the subject matter. I am also thankful to Bharat Thodji and Mahendra Joshi for their support in laboratory activities during the writing of this book.

I cannot forget to express my thanks to my close friends Shreedhar Desmukh and Jitendra Pandey for their support during writing of the book. I am also thankful to Anand Gawdekar for his help in providing me with the reference material on time whenever required during the writing process.

I want to extend special thanks to the editorial team of Pearson Education for their encouragement and support at each stage of the process and in publication of the book.

HARI MOHAN PANDEY

This page is intentionally left blank.

Introduction to OOPs

1

1.1 STRUCTURED INTRODUCTION

Structured programming (known as modular programming) is a subset of procedural programming that enforces a logical structure in the programming being written, to make it more efficient and easier to understand and modify. Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or a set of similar functions coded in separate modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure. Program flow follows a simple hierarchical model that employs looping constructs such as ‘for,’ ‘repeat’ and ‘while.’ Use of the ‘Go To’ statement is discouraged in structured programming.

Structured programming was first suggested by the mathematicians Corrado Bohm and Giuseppe Jacopini. They demonstrated that any computer program can be written with just three structures: decision, sequences and loops.

In structured programming coders break larger pieces of code into shorter subroutines (functions, procedures, methods, blocks or otherwise) that are small enough to be understood easily. In general, programs should use local variables and take arguments by either value or reference. These techniques help to make isolated small pieces of code easier to understand the whole program at once. PASCAL, Ada and C are some of the examples of structured programming languages.

1.1.1 Sequence Structure

A sequence structure consists of a single entry and single exit statement, i.e., it makes a sequential flow (Figure 1.1).

1.1.2 Loop or Iteration Structure

The loop consists of a sequence of statements, which are executed based on the logical condition. (Figure 1.2)

1.1.3 Decision Structure

Decision structure consists of a condition which may be true or false. Depending upon the condition, different branches are taken and executed (Figure 1.3).

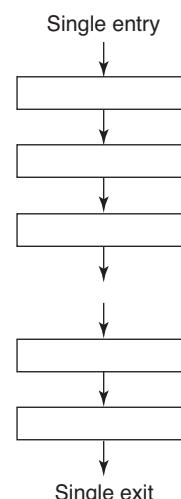


Figure 1.1 Implementation of sequence structure

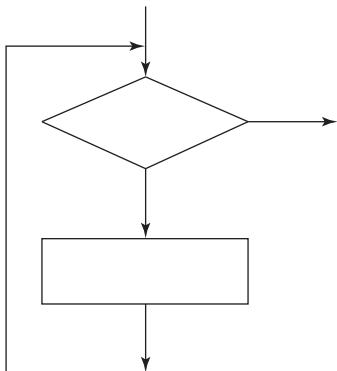


Figure 1.2 Implementation of loop or iteration structure

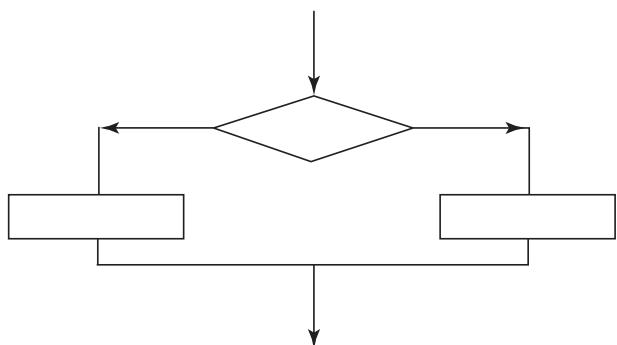


Figure 1.3 Implementation of decision structure

1.2 PROCEDURAL PROGRAMMING

Procedural programming is a programming paradigm based upon the concept of the procedural call. Procedures, also known as routines, subroutines, methods or functions, simply contain a series of computational steps to be carried out. Any given procedure can be called at any point during a program's execution including other procedures or itself. Especially in large, complicated programs, modularity is desirable. It can be achieved using procedures that have strictly defined channels for input and output and also clear rules about what types of input and output are allowed or expected. Inputs are usually specified syntactically in the form of arguments and the outputs delivered as return values.

To be considered a procedural, a programming language should support procedural programming by having an explicit concept of a procedure and syntax to define it. It should ideally support specification of argument types, local variables, recursive procedure calls and use of procedures in separately built program constructs. It may also support distinction of input and output arguments.

C, ALGOL are the classic examples of procedural programming language.

Characteristics of Procedure-oriented Programming:

1. Top-down approach is followed.
2. Data is given less importance than function.
3. Vulnerability of data is there, as functions share global data.
4. Functions manipulate global data, without letting other functions to know.
5. Big program is divided into small modules.
6. Algorithms are designed first without bothering about minute details.

1.3 PROGRAMMING METHODOLOGY

Bottom-up and Top-down Approaches

Any problem can be dealt with two ways, namely top-down or bottom-up. A simple example is given in Figure 1.4 to illustrate the concept.

Sorting an array of numbers involves the following:

1. Comparison
2. Exchange

In top-down approach, at the top level, an algorithm has to be formulated to carry out sorting using the above operations. Once the algorithm is confirmed, then the algorithms for comparison and exchange are formulated, before implementation of the entire algorithm. Therefore, in this approach one begins from the top level without bothering about the minute details for implementation, to start with.

The bottom-up approach is just the reverse. The lower level tasks are first carried out and are then integrated to provide the solution. In this method lower level structures are carried out. Here the algorithms for exchange and comparison are formulated before formulating the algorithms for the whole problem.

In any case, dividing the problem into small tasks and then solving each task provides the solution. Therefore, either the top-down or bottom-up methodology has to be adopted for dividing the problem into smaller modules and then solving it. In the top-down methodology, the overall structure is defined before getting into the details, but in the bottom-up approach, the details are worked out first before defining the overall structure.

Points About Bottom-up Approach

1. In bottom-up design, individual parts of the system are specified in detail. The parts are then linked together to form larger components, which are in turn linked until a complete system is formed.
2. Bottom-up design yield programs which are smaller and more agile. A shorter program does not have to be divided into many components, and fewer components means programs which are easier to read or modify.
3. Bottom-up design promotes code reusability. When you write two or more programs, many of the utilities you wrote for the first program will also be useful in the succeeding ones. That is why reusability of code is one of the main benefits of the bottom-up approach.
4. Bottom-up design makes programs easier to read.
5. Working bottom-up helps to clarify your ideas about the design of your program.
6. Bottom-up programming may allow you for unit testing, but until most of the system comes together none of the system can tested as a whole, often causing compilations near the end of the project.
7. Examples of programming which use this approach are C++ and Java.

Points About Top-down Approach

1. In the top-down model, an overview of the system is formulated, without going into the detail for any part of it. Each part of the system is then refined by designing it in more detail.
2. Each new part may then be refined again, defining it in yet more detail until the entire specification is detailed enough to validate the model.
3. Top-down approaches emphasize planning and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system.
4. Top-down programming is a programming style, the mainstay of traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces.
5. The technique for writing a program using top-down methods is to write a main procedure that have been coded the program is done.

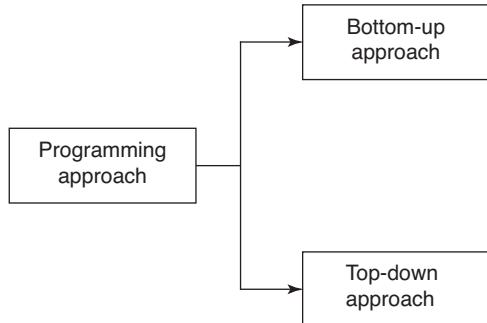


Figure 1.4 Different approaches of programming

6. Top-down programming may complicate testing, since nothing executable will even exist until near the end of the project.
7. Examples of programming which use this approach are C and Pascal.

1.4 OBJECT-ORIENTED PROGRAMMING

In computer science, object-oriented programming, OOP for short, is a computer programming paradigm.

The idea behind object-oriented programming is that a computer program may be seen as comprising a collection of individual units, or objects that act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions or simply as a list of instruction to the computer. Each object is capable of receiving messages, processing data and sending messages to other objects. Each object can be viewed as an independent little machine or actor with a distinct role or responsibility. In order to act as an object-oriented programming language, a language must support three object-oriented features:

1. Polymorphism
2. Inheritance
3. Encapsulation

Together, they are called the PIE principle (Figure 1.5).

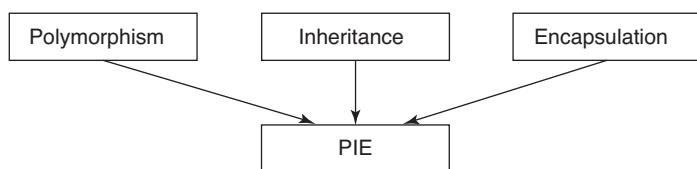


Figure 1.5 The PIE principle

1.5 BASIC CONCEPTS OF OOPS

The following are the basic concepts applied in object-oriented programming language.

- | | |
|---------------------|--------------------|
| 1. Class and object | 5. Polymorphism |
| 2. Encapsulation | 6. Inheritance |
| 3. Abstraction | 7. Dynamic binding |
| 4. Data hiding | 8. Message passing |

1. Class and object: A class is termed as a basic unit of encapsulation. It is a collection of function code and data, which forms the basis of object-oriented programming. A class is an abstract data type (ADT), i.e., the class definition only provides the logical abstraction. The data and function defined within the class, spring to life only when a variable of type class is created. The variable of type class is called an object which has a physical existence and also known as instance of class. From one class, several objects can be created. Each object has similar set of data defined in the class and it can use functions defined in the class for the manipulation of data.

For example: We can create a class car which may have properties such as company, model, year of manufacture, fuel type, etc., and which may have actions such as **acceleration ()**, **brake ()** etc.

Objects are the basic run time entity in a C++ program. All objects are instances of a class. Depending upon the type of class, an object may represent anything such as a person, mobile, chair, student, employee, book, lecturer, speaker, car, vehicle or anything which we see in our daily life. The state of an object is determined by the data values they are having at a particular instance. Objects occupy space in memory, and all objects share same set of data items which are defined when class is created. Two objects may communicate with each other through functions by passing messages.

In layman's terms:

- **Animal** can be stated as a class and **lion, tiger, elephant, wolf, cow**, etc., are its objects.

- **Bird** can be stated as class and **sparrow, eagle, hawk, pigeon**, etc., are its objects.
- **Musician** can be stated as a class and **Himesh Reshma, Anu Malik, Jatin-Lalit** are its objects.

2. Encapsulation: Encapsulation is the mechanism that binds together function and data in one compact form, known as class. The data and function may be private or public. Private data/function can be accessed only within the class. Public data/code can be accessed outside the class. The use of encapsulation hides complexity from the implementation. Linking of function code and data together gives rise to objects which are variables of type class.

3. Abstraction: Abstraction is a mechanism to represent only essential features which are of significance and hides the unimportant details. To make a good abstraction we require a sound knowledge of the problem domain, which we are going to implement using OOP principle. As an example of the abstraction consider a class **Vehicle**. When we create the **Vehicle** class, we can decide what function code and data to put in the class such as vehicle name, number of wheels, fuel type, vehicle type, etc., and functions such as changing the gear, accelerating/decelerating the vehicle. At this time we are not interested in vehicle works such as how acceleration, changing gear takes place. We are also not interested in making other parts of the vehicle to be part of the class such as model number, vehicle color, etc.

4. Data hiding: Data hiding hides the data from external access by the user. In OOP language, we have special keywords such as private, protected, etc., which hides the data.

5. Polymorphism: If we bifurcate the word polymorphism we get ‘poly’, which means many and ‘morphism’, which means form (Figure 1.6).

Thus, polymorphism means more than one form. Polymorphism provides a way for an entity to behave in several forms. In layman’s terms an excellent example of polymorphism is Lord Krishna in Hindu mythology. From programmers’ point of view polymorphism means one interface, many methods.’

It is an attribute that allows one interface to control access to a general class of actions. For example, we want to find out maximum of three numbers; no matter what type of input we pass, i.e., integer, float, etc. Because of polymorphism we can define three variable versions of the same function with the name max3. Each version of this function takes three parameters of the same time, i.e., one version of max3 takes three arguments of type integer, another takes three arguments of type double and so on. The compiler automatically selects the right version of the function depending upon the type of data passed to the function max3. This is also termed as function polymorphism or function overloading.

The two types of polymorphism are the following: (a) Compile time polymorphism, (b) Run time polymorphism

6. Inheritance: Inheritance is the mechanism of deriving a new class from the earlier existing class. The inheritance provides the basic idea of reusability in object-oriented programming. The new class inherits the features of the old class. The old class and new class is called (given as pair) base-derived, parent-child, super-sub.

The inheritance supports the idea of classification. In classification we can form hierarchies of different classes each of which having some special characteristics besides some common properties. Through classification, a class needs only definition of those qualities that make it unique within its class.

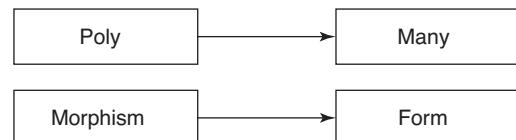


Figure 1.6 Bifurcation of polymorphism

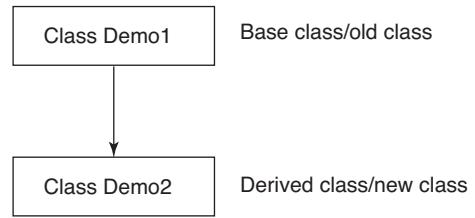


Figure 1.7 Demonstration of inheritance

Examples:

- In the example given below, we have a **vehicle class** at the top in hierarchy. All the common features of a vehicle can be put in this class. From this, we can derive a new class, i.e., **two wheeler**, which contains features specific to two-wheeler vehicles only.
- As another example we can take an engineering college as the top class and its various departments such as computer, electronics, electrical, etc., as the subclasses. The university to which the engineering college is affiliated may be its parent class.
- Dynamic binding:** Binding means linking. It involves linking of function definition to a function call.

- If linking of function call to function definition, i.e., a place where control has to be transferred is done at compile time, it is known as static binding.
- When linking is delayed till run time or done during the execution of the program then this type of linking is known as dynamic binding. Which function will be called in response to a function call is find out when program executes.

- Message passing:** In C++, objects communicate each other by passing messages to each other. A message contains the name of the member function and arguments to pass. The message passing is shown below:

```
object. method (parameters);
```

Message passing here means object calling the method and passing parameters. Message passing is nothing but calling the method of the class and sending parameters. The method in turn executes in response to a message.

1.6 CHARACTERISTICS OF OOPS

- Programs are divided into classes and functions.
- Data is hidden and cannot be accessed by external functions.
- Use of inheritance provides reusability of code.
- New functions and data items can be added easily.
- Data is given more importance than functions.
- Follows bottom-up approach.
- Data and function are tied together in a single unit known as class.
- Objects communicate each other by sending messages in the form of function.

1.7 ADVANTAGES OF OOPS

- Code reusability in terms of inheritance.
- Object-oriented system can be easily upgraded from one platform to another.
- Complex projects can be easily divided into small code functions.
- The principle of abstraction and encapsulation enables a programmer to build secure programs.
- Software complexity decreases.
- Principle of data hiding helps the programmer to design and develop safe programs.
- Rapid development of software can be done in short span of time.
- More than one instance of same class can exist together without any interference.

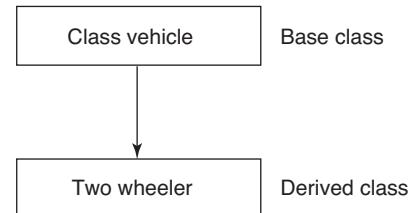


Figure 1.8 Working of inheritance

1.8 OBJECT-ORIENTED LANGUAGES

Some of the most popular object-oriented programming languages are:

- | | |
|--------------|------------|
| 1. C++ | 5. Ruby |
| 2. Java | 6. Delphi |
| 3. smalltalk | 7. Charm++ |
| 4. Eiffle | 8. Simula |

1.9 OBJECT-BASED LANGUAGES

The language which only concerns with classes and objects and do not have features such as inheritance, polymorphism, encapsulation (do not satisfy the PIE principle) is known as object-based languages. In these types of languages you can create classes and object and can work with them. They usually have a large number of built-in objects of various types. Some of the languages which are object based are Java script, Visual Basic, etc.

REVIEW QUESTIONS

1. What is the difference between the state and the behaviour of a class?
2. What is an abstract data type?
3. What is the difference between an abstract class and an abstract data type?
4. What is the difference between public, protected and private keywords?
5. What is an implicit argument?
6. What is the difference between composition and aggregation?
7. What is the difference between equality of objects and equality of the references that refer to them?
8. What is the difference between a constructor and a method?
9. What are the characteristics of procedure-oriented programming?
10. Explain the working of structured programming.
11. How is message passing done in object-oriented programming?
12. Distinguish between object-oriented and object-based programming language with suitable examples.

Multiple Choice Questions

1. The procedural programming follows
 - (a) top-down approach
 - (b) bottom-up approach
 - (c) both (a) and (b)
 - (d) none of the above
2. In procedural programming
 - (a) big program is divided into small modules
 - (b) big program is solved incrementally
 - (c) there is no need to divide the big program into small modules
 - (d) none of the above
3. Which programming language uses the top-down approach?

(a) Java	(c) C++
(b) Pascal	(d) None of the above
4. The languages that only concern with classes and objects and do not have features like inheritance, polymorphism and encapsulation are known as
 - (a) object-oriented languages
 - (b) structured languages
 - (c) object-based languages
 - (d) none of the above
5. Which of the following is an object-oriented language?

(a) Ruby	(c) Java Script
(b) Delphi	(d) Visula Basic
6. The programming language smalltalk is a
 - (a) object-based language
 - (b) structured programming language
 - (c) object-oriented language
 - (d) none of the above

7. In object-oriented programming, data and function bind together as
 - (a) modules
 - (c) encapsulation
 - (b) class
 - (d) abstraction
8. Objects communicate each other by sending messages in the form of
 - (a) class
 - (b) through inheritance
 - (c) functions
 - (d) none of the above
9. Which of the following is an example of procedural programming language?
 - (a) C++
 - (c) ALGOL
 - (b) Java
 - (d) Ruby
10. Data is given less importance in
 - (a) structured programming language
 - (b) object-oriented language
 - (c) procedural language
 - (d) object-based language

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. a 3. b 4. c 5. a 6. c 7. b 8. c 9. c 10. c

Marching Towards Java and Java Bytecodes

2

2.1 INTRODUCTION: WHAT IS JAVA?

Java is an easy to learn, object-oriented programming (OOP) language developed by Sun Microsystems. There are different types of programming languages, but Java is unique among them. For example, a Visual Basic program executes correctly on Microsoft Windows, but it does not run on Apple Mac OS. With Java, a developer can write an application and run that on different operating systems (including Windows, Mac OS, Unix and Mainframes) without any modification in the code.

Typically, when developing an application in a programming language, the source code is passed through a compiler (another application) that transforms the code into a set of native instructions (for target operating system). For example, when developing applications that run on Windows operating systems, they usually run on Intel processors (like Pentium chipset). The compiler converts the source code into instructions which the processor can understand and execute.

When writing Java applications, the developer does not need to directly call Windows (or other operating systems) library functions. The Java compiler does not write native instructions; instead it generates bytecodes for a virtual machine called Java Virtual Machine (JVM).

2.2 EVOLUTION OF JAVA

1. In 1990, James Gosling was given a task of creating programs to control consumer electronics. Gosling and his team at Sun Microsystems started designing their software using C++ because of its object-oriented nature. Gosling, however, quickly found that C++ was not suitable for the projects they had in mind. They faced problems due to program bugs like memory leak, dangling pointer and multiple inheritance.
2. Gosling soon decided that he would develop his own, simplified computer language to avoid all the problems faced in C++.
3. In designing a new language, Gosling kept the basic syntax and object-oriented features of the C++ language.
4. When Gosling completed his language design project, he had a new programming language that he named 'Oak'.
5. Oak was first used in the Green project, wherein the development team attempted to design a control system for use in the home. This control system would enable the user to manipulate a list of devices, including TVs, VCRs, lights and telephones, all from a hand-held computer called *7 (star seven).
6. By the time Sun discovered that the name 'Oak' was already claimed, they changed the name to Java.
7. In 1993, after the world wide web (www) had transformed the text-based internet into a graphics-rich environment, the Java team realized that the language they had developed would be perfect for web programming. The team came up with the concept of web applets, small programs that could

be included in web pages, and even went so far as to create a complete web browser (now called Hot Java) that demonstrated the language's power.

8. In the second quarter of 1995, **Sun Microsystems officially announced Java**. The 'new' language was quickly embraced as a powerful tool for developing internet applications. Support for Java was added in the Netscape Navigator (Web browser on UNIX) and in the Internet Explorer.

2.3 MAIN BENEFITS OF USING JAVA

1. Java programming language is very simple and object oriented. It is easy to learn and taught in many colleges and universities.
2. Java applications run inside JVM, and now all major operating systems are able to run Java including Windows, Mac and UNIX.
3. ***Write once, run anywhere***: a Java application runs on all Java platforms.
4. Java technologies have been improved by community involvement. It is suitable for most types of applications, especially complex systems that are used widely in network and distributed computing.
5. Java is very secure. Only Java applications that have permission can access the resources of the main computer. Thus, the main computer is protected from virus attackers and hackers.

2.4 KEY FEATURES OF JAVA

1. ***Java is simple***: Java was developed by taking the best points from other programming languages, particularly C and C++. Java, therefore, utilizes algorithms and methodologies that are already proven. Error prone tasks, such as pointers and memory management, have either been eliminated or are handled by the Java environment automatically rather than by the programmer. Since Java is primarily a derivative of C++, which most programmers are conversant with, it has a familiar feel which makes it easy to use.
2. ***Java is object oriented***: Even though Java has the look and feel of C++, it is a wholly independent language which has been designed to be object oriented from the ground up. In OOP, the data is treated as objects to which methods are applied. Java's basic execution unit is the class. Java is considered as a pure OOP language as no coding outside of the class definitions, including `main()`, is allowed. Java contains an extensive class library available in the core programming packages. The advantages of OOP include reusability of code, extensibility and dynamic applications.
3. ***Java is distributed***: Commonly used Internet protocols such as HTTP and FTP as well as calls for network access are built into Java. Internet programming can call on the functions through the supplied libraries and access files on the Internet, as easily as writing to a local file system. Applets can be created and distributed over the communication links, which can be run on any machine supporting JVM.
4. ***Java is compiled and interpreted***: When Java code is compiled, the compiler outputs the Java bytecode which is executable for the JVM. The JVM does not exist physically, but it is the specification for a hypothetical processor that can run Java code. The bytecode is then run through a Java interpreter on any given platform that has the interpreter ported to it. The interpreter converts the code to the target hardware and executes it. This provides portability to any machine for which a virtual machine has been written. The two steps of compilation and interpretation allow for extensive code checking and improved security.
5. ***Java is robust***: Java compels the programmer to be thorough. It carries out type checking at both compile and run time and makes sure that every data structure has been clearly defined and typed. Java manages memory automatically by using an automatic garbage collector. The garbage collector runs as a low priority thread in the background by keeping track of all objects and references to those objects in a Java program. When an object has no more references, the garbage collector tags it for removal and removes

the object when there is an immediate need for more memory. Exception handling built-in, strong type checking (i.e. all data must be declared as explicit type), initializing local variables are the key points which make Java robust.

6. *Java is secure:* The Java language has built-in capabilities to ensure that violations of security do not occur. First and foremost, at compile time, pointers and memory allocation are removed thereby eliminating the tools that a system breaker could use to gain access to system resources. Memory allocation is deferred until run time. Even though the Java compiler produces only correct Java code, there is still the possibility of the code being tampered with between compilation and run time. Java guards against this by using the bytecode verifier to check the bytecode for language compliance when the code first enters the interpreter, before it even gets the chance to run.

The bytecode verifier ensures that the code does not do any of the following:

- a. False pointers
- b. Violate access restrictions
- c. Incorrectly access classes
- d. Use illegal data conversions

At run time, the security manager determines what resources a class can access such as reading and writing to the local disk. Sun Microsystems are currently working on a public-key encryption system to allow Java applications to be stored and transmitted over the Internet in a secure encrypted form.

7. *Java is architecturally neutral:* The term ‘architectural neutral’ means that Java program does not depend upon the specific architecture of the machine on which they run. The Java compiler source code to a stage which is intermediate between source and native machine code. This intermediate stage is known as the bytecode or “.class” file which is neutral. The bytecode conforms to the specification of a hypothetical machine called JVM and can be efficiently converted into native code for a particular processor. Therefore, Java does not support the concept of executable files as they are machine dependent.
8. *Java is portable:* The Java language follows the principle of ‘write once, run anywhere’ which means Java application/applet once written can be run on many platforms. By porting an interpreter for the JVM to any computer/operating system, one is assured that all code compiled for it will run on that system. This forms the basis for Java’s portability. Another feature, which Java employs in order to guarantee portability, is by creating single standard for data sizes irrespective of processor or operating system platforms, i.e., all primitive types in Java are machine independent. For example, the size of the integer is 2 bytes under DOS and 4 bytes in Windows, but for Java the size of integer will remain fixed irrespective of OS, whether it is DOS, Windows, Linux or UNIX.
9. *Java is high performance:* The Java language supports many high-performance features such as multithreading, just-in-time compiling, and native code usage. Multithreading is the ability of an application to execute more than one task (i.e., thread) at the same time. For example, a word processor can carry out spell check in one document and print a second document at the same time. Java has employed multithreading to help overcome the performance problems suffered by interpreted code as compared to native code. Since an executing program hardly ever uses CPU cycles all the time, Java uses the idle time to perform the necessary garbage cleanup and general system maintenance that renders traditional interpreters slow in executing applications.

Since the bytecode produced by the Java compiler from the corresponding source code is very close to machine code, it can be interpreted very efficiently on any platform. In cases where even greater performance is necessary than the interpreter can provide, just-in-time compilation can be employed whereby the code is compiled at run time to native code before execution.

10. *Java is dynamic:* The linker of data and methods to where they are located is done at run time. New classes can be loaded while program is running. Linking is done *on the fly*. Even if libraries are recompiled, there is no need to recompile code that uses classes in those libraries. This differs from C++ which uses static binding. This can result in *fragile* classes for the cases where linked code is changed and memory pointers then point to the wrong addresses.

2.5 JAVA CHARACTER SET

A Java program is a collection of number of instructions written in a meaningful order. Further instructions are made up of keywords, variables, functions, objects, etc., which uses the Java character set defined by Java. It is a collection of various characters, digits and symbols which can be used in a Java program (Table 2.1). It comprises the followings:

S. No.	Elements of Java Character Set
1	Upper case letters: A to Z
2	Lower case letters: a to z
3	Digits: 0 to 9
4	Symbols (see the table below)

Symbols	Name	Symbols	Name
~	Tilde	>	Greater than
<	Less than	&	Ampersand
	Or/pipe	#	Hash
>=	Greater than equal	<=	Less than equal
==	Equal	=	Assignment
!=	Not equal	^	Caret
{	Left brace	}	Right brace
(Left parenthesis)	Right parenthesis
[Left square bracket]	Right square bracket
/	Forward slash	\	Backward slash
:	Colon	;	Semicolon
+	Plus	-	Minus
*	Multiply	/	Division
%	Mod	,	Comma
'	Single quote	"	Double quote
>>	Right shift	<<	Left shift
.	Period	_	Underscore
>>>	Right shift with zero fill	\$	Dollar

Table 2.1 Java character set

2.5.1 Java Tokens

The smallest individual unit in a Java program is called a Java token. Six types of tokens are defined in Java (Figure 2.1).

1. **Keywords:** Keywords are those words whose meaning has already been known to the compiler. The meaning of each keyword is fixed. One can simply use the keyword for its intended meaning. The meaning of keywords cannot be changed. Also one cannot use the keywords as names for variables, function, array, etc. All keywords are written in small case letters.

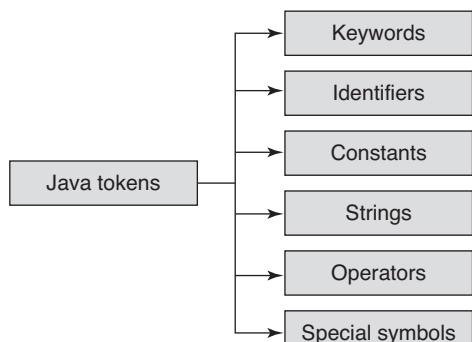


Figure 2.1 Six types of tokens defined in Java

There are 48 keywords in Java. They are mentioned in Table 2.2.

abstract	boolean	byte	break	case	catch	char
class	continue	default	do	double	else	extends
false	final	finally	float	for	if	implements
import	instanceof	int	interface	long	native	new
null	package	private	protected	public	return	short
static	super	switch	synchronized	this	throw	throws
transient	try	true	void	volatile	while	

Table 2.2 Keywords used in Java

2. *Identifier:* Identifiers are names given to various program elements like variables, array, functions, structures, etc.

Rules for writing identifiers

Rule 1: The first letter must be an alphabet or an underscore.

Rule 2: From the second character onwards, any combination of digits, alphabets or underscore is allowed.

Rule 3: Only digits, alphabets, underscore are allowed. No other symbol is allowed.

Rule 4: Keywords cannot be used as identifiers.

Examples of valid and invalid identifiers (on the basis of above given rules):

Valid identifiers

Order_no	name	_err	_123
xyz	radius	a23	int_rate

Invalid identifiers

Order-no	12name	err	int
x\$	s name	hari+45	123

3. *Constants or literal:* Constants in Java refer to fixed values that do not change during the execution of a program. They are also termed as literals. A name for a constant value is called a literal. A literal is what you have to type in a program to represent a value. ‘A’ and ‘*’ are literals of type char, representing the character value A and *.

There are various types of constants in Java as shown in Figure 2.2.

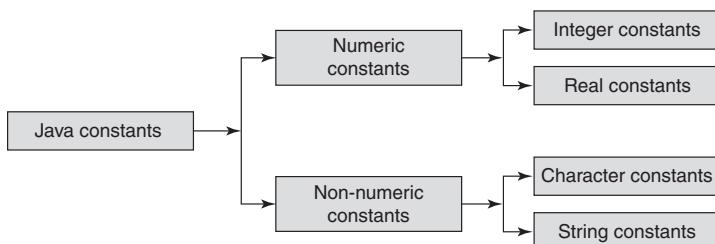


Figure 2.2 Java constants

They are classified into the following categories as given below:

- (a) Integer constants: Ordinary integers, such as 17456 and -32, are literals of type byte, short or int, depending on their size. One can make a literal of type long by adding ‘L’ as a suffix. For example: 17L or 728476874368L.

They are of three types:

- (i) Decimal constants: They are sequence of digits from 0 to 9 without fractional part. It may be negative, positive or zero. For example: 12, 455, -546, 0, etc.

- (ii) Octal constant: They have sequence of numbers from 0 to 7 and first digit must be 0. For example: 034, 0, 0564, 0123, etc.
- (iii) Hex constant: They have sequence of digits from 0 to 9 and A to F (represents 10 to 15). They start with 0x or 0X. For example: 0x34, 0xab3, 0X3E, etc.
- (b) Real constants: They are the number with fractional part. They are also known as floating point constants. Any numerical literal that contains a **decimal point** or **exponential** is a literal of type double. To make a literal of type float, one have to append an ‘F’ or ‘f’ to the end of the number. For example, ‘1.2F’ stands for 1.2 considered as a value of type float. You need to know this because the rules of Java say that a value of type double cannot be assigned to a variable of type float. So, an error message may appear if you try to do something like “float” x=1.2; .” You have to say “float x = 1.2 F”. / So, the advice is to stick to type double for **real numbers**.

Real constant can also be represented in exponential or scientific notation which consists of two parts. For example, the number 212.345 can be represented as 2.12345e+2 where e+2 mean 10 to the power 2. The portion before the e, i.e. 2.12345, is known as **mantissa** and +2 is the **exponent**. Exponent is always an **integer number** which can be written in either lower or upper case. There may be many more representation of the above given number.

For example: 34.56, 0.67, 1.23f, 2.45F.

- (c) Single character constants: They are enclosed in single quote. They consist of single character or digit. For example: ‘4’, ‘A’, ‘\n,’ etc.
- (d) Boolean constants: For the type boolean, there are precisely two literals: true and false. These literals are typed without quotes. But they represent values and not variables. Boolean values occur most often as the values of conditional expressions. For example, size > 30 is a Boolean-valued expression that evaluates to true if the value of the variable size is greater than 30, and false if the value of size is not greater than 30. Boolean-valued expressions are used extensively in control structures. Of course, Boolean values can also be assigned to variables of type Boolean.
- (e) String constants: They are sequence of characters, digits or any symbol enclosed in double quote. For example: “hello”, “23twenty three”, “&^ABC”. “2.456” etc.
- (f) Backslash constants: Java defines several backslash constants which are used for special purpose. They are called so because each backslash constant starts with a backslash (\). They are represented with two characters whose general form is \char but treated as a single character. They are also called escape **sequence**. Given below is the list of backslash (escape sequence) character constants (Table 2.3):

BCC	Meaning	ASCII
\b	Backspace	08
\f	Form feed	12
\n	New line	10
\r	Carriage return	13
\”	Double quote	34
\'	Single quote	39
\?	Question mark	63
\a	Alert	07
\t	Horizontal tab	09
\v	Vertical tab	11
\0	Null	00

Table 2.3 The backslash character constant

- (g) Special symbols: Special symbols or separators are having special meaning in the Java program. They are [], (), {}, comma (,), semicolon (;) and dot (.).
- The symbol [] is used as subscript operator and with an array.
 - The symbol () is known as function operator and used with function and enclosing expressions.
 - The symbol {} is known as braces, where '{' is known as an opening brace and '}' is a closing brace. They are used for defining the class, interface, function or block.
 - The comma is used as separators between identifiers. The separator semicolon is used as terminator or delimiter.
 - The dot is used for accessing members of class.

2.6 VARIABLES

A variable is a named location in memory that is used to hold a value that can be modified in the program by the instruction. A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if a user tries to violate this rule. Java is called a strongly typed language because it enforces this rule. All variables must be initialized prior to their use or they must be assigned values taken from standard input stream, i.e., keyboard. Java supports variables to be initialized dynamically at any place within the program.

The general form of variable declaration is:

```
data type variable [list];
```

Here the list denotes more than one variable separated by commas:

For example:

```
int a;
float b,c;
char p,q;
```

In the above example, a is a variable of type int, b and c are variables of type float, and p, q are variables of type char. int, float and char are data types used in Java. The rule for writing the variables are the same as for writing identifiers since a variable is nothing but an identifier.

Java program allows to declare variables anywhere in the program. It is not necessary to declare all the variables in the beginning of the program. It can be declared right on the place where one wants to use it. An advantage is that sometimes lots of variables are declared in advance in the beginning of the program and many of them are unreferenced. Declaring variables at the place where they are actually required is handy. It is not necessary to declare all the variables earlier prior to their use. On the other hand, it is burdensome to look for all the variables declared in the program as they will be scattered everywhere in the whole program.

Java also allows variables to be initialized dynamically at the place of their use. It can be written anywhere in the program. For example:

```
int x = 23;
float sal = 2345f;
char name[] = 'Hari';
```

This is known as ‘dynamic initialization’ of the variables. The assignment of values to types must be compatible as Java is a strongly typed language.

2.7 HOW TO PUT COMMENTS IN JAVA?

Java supports three types of comments: one for single line, another one for multiline and the last one for documentation purpose.

In Java, the multiline comments must start with `/*` on the first line and end with `*/` on the last line as shown below:

```
/*
This is multi
line comment */
```

You must have seen this type of comment if you are familiar with C. The second form of comment comes from C++. It is the single-line comment, which starts at a `//` and continues until the end of the line. This type of comment is convenient and commonly used.

```
// this is single-line comment.
```

The documentation type comment starts with `/**` on one line and ends with `*/`. Each line between the beginning and end of the documentation comment an `*` is placed. This is generally used for documenting class, interface and methods.

2.8 DATA TYPES IN JAVA

A data type is a class of data object which shares the same properties and methods for creating and manipulating them. Stating more simply, a data type is a set of values and a set of operations on those values.

There are eight so-called primitive data types built into Java. The primitive types are named `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`. The first four types hold integers (whole numbers such as 17, -2345 and 0). The four integer types are distinguished by the ranges of integers they can hold. The `float` and `double` types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type `char` holds a single character from the unicode character set. The unicode character code is of 16 bit code which can support almost all types of characters and symbols found in different types of languages like Russian, Korean, Japanese and Chinese. Since Java is designed to allow applets (small Java program meant for Internet) to be written for world wide use, it makes sense that it would use unicode to represent characters. A variable of type `boolean` holds one of the two logical values true or false.

The most important difference between the data types in other languages and in Java is that all primitive types in Java are machine independent, i.e., irrespective of the machine on which you are working, all data types will be having the fix size. Size of `char` is 2 bytes, size of `int` is 4 bytes and size of `float` is 4 bytes (Table 2.4). All these sizes will remain fixed on all the machines. This makes it easy for the allocation and de-allocation of memory independent of the implementation of data types on that machine. This also increases portability of the Java programs.

S. No.	Data Types	Size (in Bytes)	Range	
			From	To
1	short	2	-32768	32767
2	int	4	-2147483648	2147483647
3	long	8	-9223372036854775808	9223372036854775807
4	float	4	1.17549e - 038	3.40282e+038
5	char	2	-32768	32767
6	byte	1	-128	127
7	double	8	2.22507e - 308	1.79769e+308
8	boolean	1bit	True and false	

Table 2.4 Range of data types

One important thing to note about Java regarding the data type is that it is a strongly typed language. Each variable, expression, literal has a strictly defined type, and during assignment and function call, the type is strictly checked for compatibility. The Java compiler checks all expressions and parameters to ensure that the types are compatible.

A strongly typed language is one in which each name in a program in the language has a single type associated with it, and that type is known as compile time. Strong typing is a good thing because it improves security by preventing many form of accidental misuse of the data. Empirically, it has been observed that strong typing is a very effective aid to achieve program reliability.

The difference between `boolean` type in Java and other languages is that in other languages any non-zero value is considered as true and 0 is considered as false value. In Java with `boolean` types, one can use expressions which return either true or false values. Integers or other data types cannot be used to act as true or false value.

2.9 STRUCTURE OF A JAVA PROGRAM

The general structure of any Java program is as given below. It states that a standard Java program may look like according to various sections present in the structure (Figure 2.3).

1. The first section documentation is optional, and it is used to put comments for the program, usually the program heading as given.
2. In the second section, the package is defined if we want to include the file being written to be the part of the package. After the package statement, the package name is specified. A package is any collection of classes and interfaces. It is an optional section.
3. If the program needs to use standard Java classes, then we must use import statement for importing the packages containing the Java classes. This is also an optional section.
4. Next we can create various classes depending upon our requirements. This is also an optional section if you are not dealing with classes and objects and making elementary programs without using the class. In Java, all the functions, variables and constants must be defined inside the class.
5. The last section contains the class in which the `main()` function is defined. A class containing the main function must be present in every Java program. It contains all statements which are to be executed. Inside the main function, we may create objects of classes created earlier. All statements are put inside the braces and terminated with semicolon (;).

A full-fledged Java program is given below. If you do not understand extra stuff at this point, do not worry. This is just to give you an idea of the structure of a Java program.

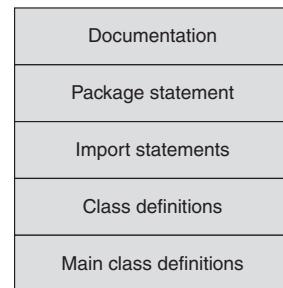


Figure 2.3 Structure of a simple Java program

```

/** Documentation
 * Demo program for java program structure,
 * does not serve any purpose except demonstration
 */

package demopackage;      //package name is demopackage
import java.io.*;         //importing package
import java.util.*;        //importing package
  
```

```

/*definition of class starts here */
class demo
{
    private int x, y;
    void input(int a, int b)
    {
        x=a;
        y=b;
    }
    void show()
    {
        System.out.println("X="+x);
        System.out.println("Y="+y);
    }
} //class definition of demo ends here
class Main
{
    public static void main(String[]args) /*main function
                                           definition*/
    {
        demo d1; //declaring objects
        d1=new demo(); //actually creating objects
        d1.input (10,20); //calling function input
        d1.show(); //calling function show
    } //main function ends here
} //Main class ends here

```

2.10 YOUR FIRST PROGRAM IN JAVA

```

/*PROG 2.1 DISPLAYING "Welcome in Java Programming" on to the
screen */

```

```

class Hello
{
    public static void main(String[] args)
    {
        System.out.println("WELCOME IN JAVA PROGRAMMING");
    }
}

```

Explanation: The class name is 'hello.' In Java, everything has to be in a class; even the main function must be defined inside the class. In order to define a program, a class must include a method called `main`, with a definition that takes the form (Figure 2.4)

```

public static void main(String[] args)
{
    //statements
}

```

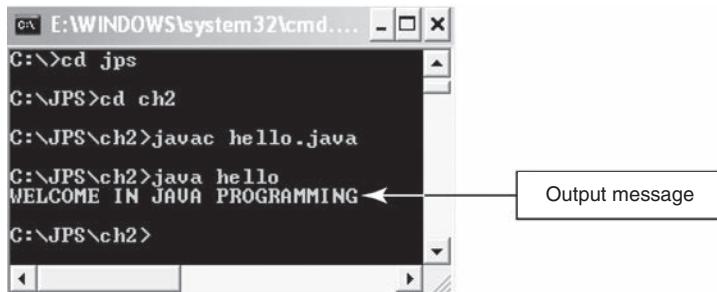


Figure 2.4 Output screen of Program 2.1

The word ‘public’ in the first line of `main()` means that this routine can be called from outside of the program. This is essential because the `main()` routine is called by the Java interpreter. The use of `static` for method is to allow you to call that method creating an object. The `void` means functions do not return any value. The arguments within `main` are called command line arguments. `String` is a built-in class in Java and all the arguments are passed to program and are collected in array `args`. The name of the `String` array may be anything.

When Java interpreter runs the program, the interpreter calls the `main()` subroutine and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The `main()` routine can call subroutines that are defined in the same class or even in other classes, but it is the `main()` routine that determines how and in what order the other subroutines are used.

To print anything onto the screen, we write as

```
System.out.println("WELCOME IN JAVA PROGRAMMING");
```

In the above example, ‘`System`’ is a built-in class defined in a package `java.lang`. (A package is a collection of classes.) The package is by default imported for all Java programs. ‘`.out`’ is the ‘`static`’ object of the class ‘`PrintStream`.’ An `static` object can be used with a class name using dot(.) operator. The `println` is a method defined for the `PrintStream` class which can be used without object using dot operator. The function prints whatever is written in quotes onto the screen and moves to the next line. Each statement in Java must end with a semicolon.

Assume the file name is `hello.java`. The program can be typed in any text editor or Java editor.

2.10.1 Compiling and Running the Program (DOS Based)

To compile and run the program and all the other programs in this book, you must first have a Java programming environment. You must install the Java Development Kit (JDK) onto your machine. The current version of the JDK is 1.6. The various components of the JDK are discussed in details later in the chapter.

Setting-up Java: The procedures to install the JDK version for Microsoft Windows Operating System are described below (Figure 2.5):

1. Run the executable file of the downloaded installation package. The name of this file under format is `JDK-version-windows-chipset_type.exe`.
2. Select all the features as below:

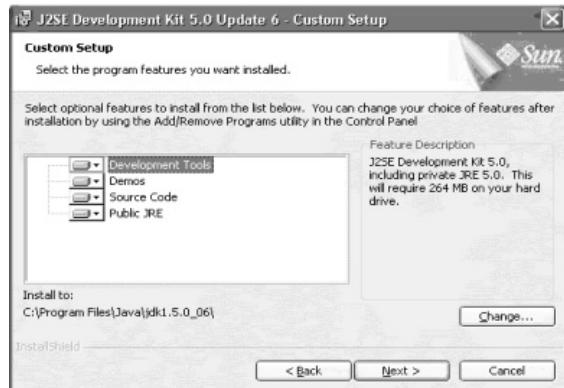
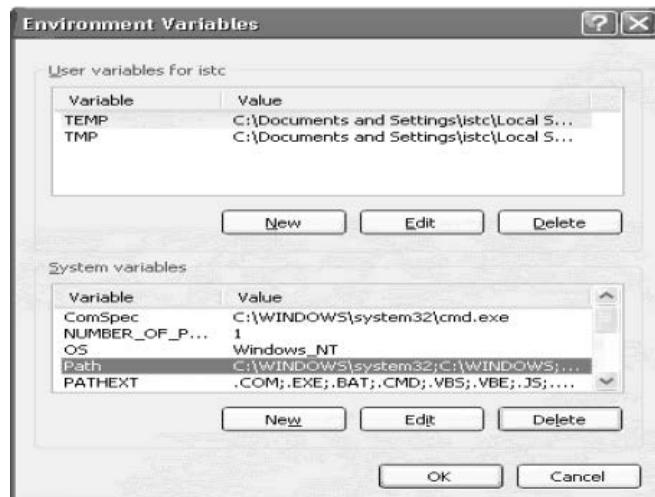


Figure 2.5 J2SE development kit

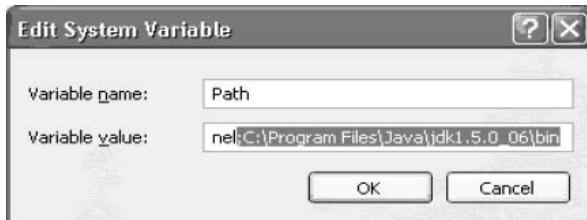
3. Set the PATH environment variable browsing to the BIN folder of the JDK installation folder.
- (a) Right click on the My Computer icon, select Properties and open the Advance tab as the following:



- (b) Click the Environment button to open the dialog Environment Variables.



- (c) Click the Edit button and append the path of the BIN folder to the end. The BIN folder is sub of the SDK installed folder, the example of its path such as: C:\Programfiles\Java\jdk1.5.0_06bin



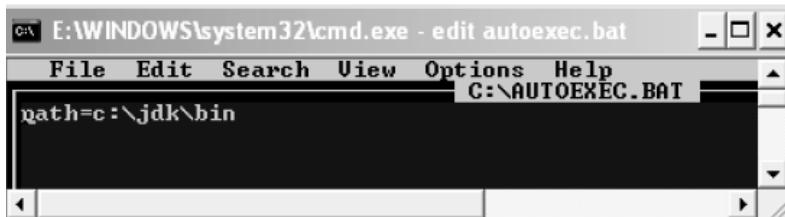
This will install Java on the computer.

Compiling Java Program: After installing and setting path if not, set up as:
Set a path for java and javac as follows:

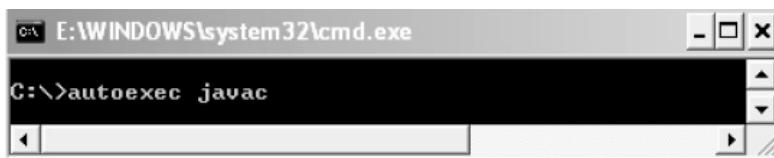
Step 1: Open the c:\edit autoexec.bat file in the Notepad Editor (or any kind of editor).



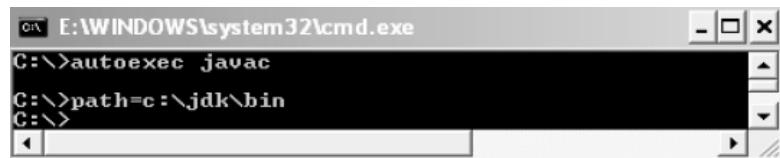
Step 2: Add this line - PATH=C:\jdk\bin



Step 3: After switching from editor to command prompt, type c:\autoexec javac and press Enter key.



After pressing Enter key, you will get your path PATH=C:\jdk\bin on the command prompt as shown below.



Step 4: Then write javac as c:\javac as shown below and press Enter key to execute all the class files.

If you do not want to set up the path, you may directly go to the **bin** directory and create your Java source file there; but setting the path is the preferred choice as you can work in any directory you like. We assume that the path has been set up.

Now develop your source code on any editor (here notepad is selected). We are in the directory JPS (Figure 2.6).

```
hello.java - Notepad
File Edit Format View Help
/*PROG 2.1 DISPLAYING "WELCOME IN JAVA PROGRAMMING" ONTO THE SCREEN*/
class hello
{
    public static void main(String[]args)
    {
        System.out.println("WELCOME IN JAVA PROGRAMMING");
    }
}
```

Figure 2.6 Showing the Java program on notepad

```
C:\JPS\ch2>javac hello.java
```

The **javac** is the java compiler which takes input as the name of java source file. If there is no error in the .java file and path was set up correctly, there will be no output and you will get your prompt back. It means that the file **hello.java** has been compiled successfully. The compiled file is known as bytecode file or .class file. At the prompt type **dir/p *.class** and you will see a file name **hello.class** as shown in Figure 2.7.

```
E:\WINDOWS\system32\cmd.exe
C:\JPS\ch2>dir/p *.class
Volume in drive C is Songs
Volume Serial Number is 3499-942F
Directory of C:\JPS\ch2
06/13/2008 09:59 AM           431 hello.class
   1 File(s)      431 bytes
   0 Dir(s) 20,411,961,344 bytes free
C:\JPS\ch2>
```

Figure 2.7 Showing the **hello.class** file

Running Java Program: To run the program, the following command is used:

```
C:\JPS\ch2>java hello
WELCOME IN JAVA PROGRAMMING
```

The **java** is the Java interpreter which takes .class file as argument (note: do not write the extension .class). This class file, **hello.class**, contains the Java bytecode that is executed by a Java interpreter. **Hello.java** is called the source code for the program. To execute the program, only the compiled class file is required not the source code. Note that during interpretation no file is produced. The interpreter reads the bytecode file line by line and displays output onto the screen. The interpreter is the part of JVM purely in software form.

In the program, the name of file was **hello.java** and name of the class was also **hello**. It was necessary as the Java interpreter interprets the .class file and the name of the file is same as name of the class. In case the file name and class name is different, we must compile the program by the file name

and execute the program using the class name. For example, in the above case, assume the file name is file1.java. You will compile the file as

```
C:\JPS\javac file1.java
```

The output file produced will be hello.java and not file1.java. Now the program runs as follows:

```
C:\JPS\ java hello
```

The figure below shows compilation and interpretation process for a java program that is stored in file MyProgram.java (Figure 2.8).

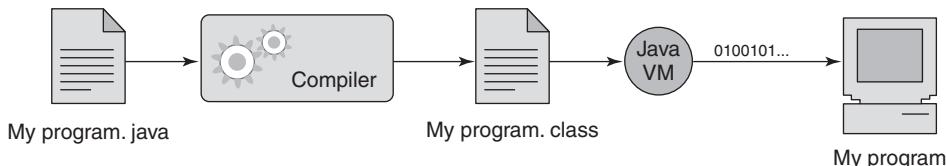


Figure 2.8 Compilation and interpretation for a Java program

2.11 JAVA IS FREE FORM

Java is a free form language. The Java program can be written in any manner, i.e., no botheration of line change, spaces, tabs, etc., as long as there is at least a white space character between any token. The white space characters in Java are space, tab and new line character. For example:

```
class freeform { public static void main(String[] args) {
    System.out.println("Java is Free Form")
}; }
```

The above program will compile smoothly and on interpretation, output will be produced ‘Java is Free Form.’ But it is usually advised not to write programs in the above manner as it makes it difficult to read and comprehend the program.

2.12 PROGRAMMING EXAMPLES

```
/*PROG 2.2 DISPLAYING OUTPUT ON SEPERATE LINES VER 1 */
```

```
class JPS1
{
    public static void main(String args[])
    {
        System.out.println("NMIMS UNIVERSITY");
        System.out.println("MPSTME SHIRPUR");
    }
}

OUTPUT:
C:\JPS\ch2>javac JPS1.java
C:\JPS\ch2>java JPS1
NMIMS UNIVERSITY
MPSTME SHIRPUR
C:\JPS\ch2>
```

Explanation: The program is simple. We have two statements that print NMIMS UNIVERSITY and MPSTME SHIRPUR on each line, respectively.

```
/*PROG 2.3 DISPLAYING OUTPUT ON SEPERATE LINES VER 2 */

class JPS2
{
    public static void main(String args[])
    {
        System.out.print("NMIMS UNIVERSITY MUMBAI \n");
        System.out.println("MPSTME SHIRPUR CAMPUS");
    }
}

OUTPUT:
NMIMS UNIVERSITY MUMBAI
MPSTME SHIRPUR CAMPUS
```

Explanation: The print method does not move to the next line after printing. So we have to use \n to make the text appear onto the next line.

```
/* PROG 2.4 DISPLAYING OUTPUT ON SEPERATE LINES USING TAB */

class JPS3
{
    public static void main(String args[])
    {
        System.out.print("\nCOMPUTER SCIENCE\t");
        System.out.println("INFORMATION TECHNOLOGY \n");
    }
}

OUTPUT:
COMPUTER SCIENCE INFORMATION TECHNOLOGY
```

Explanation: '\t' is the tab which leaves as tab and prints on the same line.

```
/*PROG 2.5 USING VARIABLES VER 1 */

class JPS4
{
    public static void main(String args[])
    {
        int num;
        System.out.println("num="+num);
    }
}
```

```

}

OUTPUT:
Compilation Error
Variable num might not have been initialized

```

Explanation: In Java, all variables must be declared and defined prior to their use. They do not contain any garbage value if not defined; instead compilation error is flashed as shown above.

```

/*PROG 2.6 USING VARIABLES VER 2*/

class JPS5
{
    public static void main(String args[])
    {
        String str;
        str = "WELCOME IN NMIMS UNIVERSITY FOR JAVA";
        System.out.println(str);
    }
}
OUTPUT:
WELCOME IN NMIMS UNIVERSITY FOR JAVA

```

Explanation: String is a class in Java defined in the file `java.lang` package. A package is a collection of classes and interfaces. The `java.lang` package is the default package imported in all Java programs. The statement `String str;` declares an object `str` of `String` class which is initialized to “WELCOME IN NMIMS UNIVERSITY FOR JAVA” and displayed using `println` method.

```

/*PROG 2.7 DEMO OF ALL TYPES OF VARIABLES */

class JPS6
{
    public static void main(String args[])
    {
        String str;
        str = "NMIMS UNIVERSITY";
        int num = 25;
        byte b = 20;
        float f = 35.46f;
        double d = 234.567;
        char ch = 'N';
        boolean bo = true;
        System.out.println("String str:=" + str);
        System.out.println("byte b :=" + b);
        System.out.println("float f :=" + f);
        System.out.println("Double d :=" + d);
        System.out.println("Boolean bo:=" + bo);
    }
}

```

```

}
OUTPUT:
String str      :=NMIMS UNIVERSITY
byte b          :=20
float f         :=35.46
Double d        :=234.567
Boolean bo      :=true

```

Explanation: To display any type of variables including objects of classes (only built-in and not user defined), we can concatenate it with string using + operator followed by the name of the variable to be printed. For example, "byte b: =" + b with b (i.e., value of b which is 20), so string becomes "byte b: = 20" and displayed. The same analogy applies to all other `println` methods used in the program.

```

/*PROG 2.8 READING STRING DATA */

import java.io.*;
class JPS7
{
    public static void main(String args[])
    {
        String sname;
        try
        {
            DataInputStream input;
            input = new DataInputStream(System.in);
            System.out.println("Enter your name");
            sname = input.readLine();
            System.out.println("Hello" + sname);
        }
        catch (Exception eobj)
        {
            System.out.println("Error");
        }
    }
}

OUTPUT:
Enter your name
Hari
Hello Hari

```

Explanation: In the first line of the program, import is the keyword, followed by `java.io` which is the package name and `.*` means all the classes from the package `java.io` will be imported to our program. The class `DataInputStream` is defined in the package `java.io`. This class has methods for reading the data from the console. The methods of this class might throw an exception (run time errors), therefore, we have placed the code for taking input into try block. A try block is a block created with `try` and `{ }`. In the try block, any Java code which might throw an exception during the execution of the program is placed. The exception thrown, if not caught, may terminate or crash the program abnormally. The catch block following the try block is responsible for catching and handling the exception thrown. In the try block, we create a reference (not object) `input` type `DataInputStream` type as:

```
DataInputStream input;
```

In the text line, we create an object of the DataInputStream class type and assign reference of it to input as:

```
input = new DataInputStream(System.in);
```

The ‘new’ is the operator for creating objects dynamically. In Java, all objects are created dynamically. Objects are instances of classes. Here, input is an instance of the class DataInputStream. The parameter System.in represents that we want to read from the standard input stream, i.e., from the keyboard. It is an object of the InputStream class type.

The readLine method of the class DataInputStream allows us to read a string from the keyboard delimited by ‘\n.’ Assume the entered string is ‘Hari.’ This entered string is read from the console using input.readLine() and assigned to sname. Next, we display the string by concatenating with ‘hari’ using + operator.

Assume that the readLine method is used for reading some data from a file and not from the keyboard. It is possible that some error might occur from the file, such as the file not found or end of file has been reached. In such cases, the method may throw an exception. The thrown exception object is caught by the catch block in object E. The object E is of the class Exception type, which can handle all types of exception. So, any run-time error will be handled by the catch block and your program will not terminate abnormally. Even try block must have a corresponding catch block. On compiling the above program, you will get message like:

```
C:\JPS\ch2>javac JPS7.java
Note: JPS7.java uses or overrides a deprecated API.
Note: Recompile with -Xlint: deprecation for details.
```

This simply means that the DataInputStream class has been replaced by some new classes in the new version of Java and that Application Programming Interface (API) is deprecated. The other new class for taking input where this message does not appear will be discussed later on. The above message does not affect execution of the program.

```
/*PROG 2.9 READING INTEGER DATA */

import java.io.*;
class JPS8
{
    public static void main(String args[])
    {
        String snum;
        int num;
        try
        {
            DataInputStream input;
            input = new DataInputStream(System.in);
            System.out.println("Enter a number");
            snum = input.readLine();
            num = Integer.parseInt(snum);
            System.out.println("You have entered:=" +
                               num);
        }
        catch (Exception eobj)
```

```

    {
        System.out.println("ERROR!!!!");
    }
}

OUTPUT:
Enter a number
30
You have entered: =30

```

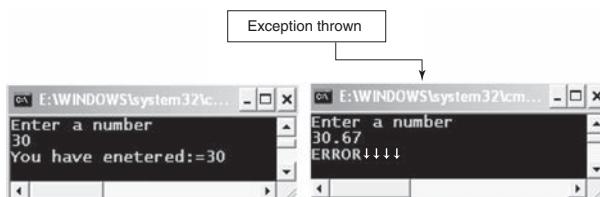


Figure 2.9 Output screen of Program 2.9

Explanation: The program is almost same as the previous one, but here we are reading into data from the console. Whatever you read from console using the readLine method is treated as string even if integer, float is input, etc. Assume that you entered 30, but 30 is a string not an integer. This 30 in string form is stored in the string snum. To convert the string into the integer, we have a static method parseInt of the Integer class which converts a string into an integer and returns the integer. The returned integer 30 is stored in variable num. If you do not enter an integer as input and enter something else, say string or float, then an exception is thrown as parseInt cannot convert the sent argument to the integer as shown in the above Figure 2.9.

The entered value can directly be converted to an integer without storing it into the string variable as

```
num =Integer.parseInt(input.readLine());
```

The similar method for taking a long value as input is:

```
Long.parseLong(String variable);
```

```

/*PROG 2.10 READING LONG DATA TYPES */

import java.io.*;
class JPS9
{
    public static void main(String args[])
    {
        long num1, num2;
        try
        {
            DataInputStream input;
            input = new DataInputStream(System.in);
            System.out.println("Enter the first long
                               value");
            num1 = Long.parseLong(input.readLine());
        }
    }
}

```

```

        System.out.println("Enter the second long
                           value");
        num2 = Long.parseLong(input.readLine());
        System.out.println("The two values you have
                           entered are");
        System.out.println("num1:=" + num1);
        System.out.println("num2:=" + num2);
    }
    catch (Exception eobj)
    {
        System.out.println("ERROR!!!!");
    }
}
}

OUTPUT:
Enter the first long value
213456789876
Enter the second long value
123456732453
The two values you have entered are
num1:=213456789876
num2:=123456732453

```

Explanation: The program is similar to the previous program, but here we have accepted a long value from the user instead of an integer value.

```

/*PROG 2.11 READING FLOAT DATA TYPE VER 1 */

import java.io.*;
class JPS10
{
    public static void main(String args[])
    {
        String snum;
        Float obj;
        float num;
        try
        {
            DataInputStream input;
            input = new DataInputStream(System.in);
            System.out.println("\nEnter a float
                               number");
            snum = input.readLine();
            obj = Float.valueOf(snum);
            num = obj.floatValue();
            System.out.println("You have eneterd"
                               + num);
            System.out.println("\n");
        }
        catch (Exception eobj)

```

```

        {
            System.out.println("ERROR!!!!");
        }
    }
}

OUTPUT:
Enter a float number
57.78
You have entered 57.78

```

Explanation: The float is a class and it is a primitive data type. Reading a float variable from console is a two-step process:

1. Convert the string object into the float object.
2. Get float value from the float object using `floatValue` method.

Assume that the number entered is 57.78 which is assigned to `snum` in string from. The `String` object is converted to an object of the `Float` class using the `static` method `valueOf` of `Float` class as:

```
obj = Float.valueOf(snum);
```

From this float object `obj` we can get the float value as,

```
num = obj.floatValue();
```

The three-line coding can be reduced to just one-line coding as shown in the next program.

```
/*PROG 2.12 READING FLOAT DATA TYPE VER 2 */
```

```

import java.io.*;
class JPS11
{
    public static void main(String args[])
    {
        float num;
        try
        {
            DataInputStream input;
            input = new DataInputStream(System.in);
            System.out.println("Enter a float number");
            num = Float.valueOf(input.readLine()).float
                  Value();
            System.out.println("You have entered"
                               + num);
        }
        catch (Exception eobj)
        {
            System.out.println("ERROR!!!!!");
        }
    }
}

```

OUTPUT:

```
Enter a float number
45.67894523
You have entered 45.678944
```

Explanation: The program is similar to the previous one, but here we have combined the three lines in just one line.

```
/*PROG 2.13 READING CHARACTER FROM CONSOLE */

import java.io.*;
class JPS12
{
    public static void main(String args[])
    {
        char ch;
        try
        {
            DataInputStream input;
            input = new DataInputStream(System.in);
            System.out.println("\n\n Enter a
                                character");
            ch = (char)(input.read());
            System.out.println("You have entered"
                               + ch);
            System.out.println("ASCII value"
                               + (int)ch);
        }
        catch (Exception eobj)
        {
            System.out.println("ERROR!!!!");
        }
    }
}

OUTPUT:
(FIRST RUN)
Enter a character
a
You have entered a
ASCII value 97
(SECOND RUN)
Enter a character
A
You have entered A
ASCII value 65
```

Explanation: To read a character, we have the read method of DataInputStream class. The method returns an integer, so we have to typecast it to a character. While displaying, we have also printed the ASCII value of the character taken by typecasting it with int.

2.13 READING USING SCANNER

One of the most remarkable inclusions in the JDK 1.5 is the scanner class defined in `java.util` package. This feature was not available in the previous version of Java. Therefore, in the earlier section, the older method of taking output from the user was used. But after reading this section, you will not look back and use only the scanner class for the input. Scanner is a simple text scanner which can parse primitive types and strings using regular expression.

A scanner breaks its input into token using a delimiter pattern, which by default matches white space. The resulting tokens may be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from `System.in`, i.e., from the keyboard.

```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt();
```

Though not necessary to use try and catch block while taking input using the scanner class, it is recommended to do so for the reasons they are in the language. Similar to `nextInt()`, there are methods for reading `long`, `byte`, `float` and other data types. They are given in the table 2.5.

Methods	Purpose
<code>nextInt()</code>	Reading an integer
<code>nextByte()</code>	Reading a byte
<code>nextLong()</code>	Reading a long
<code>nextFloat()</code>	Reading float
<code>nextBoolean()</code>	Reading a boolean
<code>nextShort()</code>	Reading short
<code>next()</code>	Reading a string, but included white space characters.

Table 2.5 Methods defined by the `Scanner` class

The programs are showing the usage:

```
/*PROG 2.14 READING AN INT AND DOUBLE USING SCANNER CLASS */
```

```
import java.io.*;
import java.util.*;
class JPS13
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("\nEnter an integer");
        int x = sc.nextInt();
        System.out.println("Integer is:=" + x);
        System.out.println("\nEnter a double");
        double d = sc.nextDouble();
        System.out.println("Double is:=" + d);
    }
}
```

```

}

OUTPUT:
Enter an integer
35
Integer is: = 35
Enter a double
345.678
Double is: =345.678

```

Explanation: The program is simple to understand.

```
/*PROG 2.15 READING STRING USING SCANNER CLASS */
```

```

import java.io.*;
import java.util.*;
class JPS14
{
    public static void main(String args[])
    {
        String str;
        Scanner sc=new Scanner(System.in);
        System.out.println("\nEnter you name here");
        str =sc.next();
        System.out.println("Welcome" + str);
    }
}
OUTPUT:
(First Run)
Enter you name here
Vijay
Welcome Vijay
(Second Run)
Enter you name here
Vijay Nath Pandey
Welcome Vijay

```

Explanation: In the program, we are reading a string from the console using the `next` method of the `Scanner` class. Note that as compared to `readLine` method of the `DataInputStream` class, the `next` method of `Scanner` class does not read the complete line of string. It breaks at the very first white space character like tab, space, etc. So if we enter the name as ‘Vijay Nath Pandey,’ only ‘Vijay’ will be returned by the `next` method as shown by second run.

2.14 COMMAND LINE ARGUMENTS

Command line arguments are the arguments that are supplied at command line when running your Java program. In Java, all command line arguments are objects of the `String` class. Few examples are given on following pages.

```
/*PROG 2.16 DEMO OF COMMAND LINE ARGUMENT */

class JPS15
{
    public static void main(String[] args)
    {
        System.out.println("\nCommand line arguments
                           are");
        for(int i=0; i<args.length; i++)
            System.out.println("Args" +(i+1)+"="
                               +args[i]);
    }
}

OUTPUT:

C:\JPS\ch2>javac command1.java
C:\JPS\ch2>java JPS15 one two three four five
Command line arguments are
Args 1=one
Args 2=two
Args 3=three
Args 4=four
Args 5=five
```

Explanation: Type the above program in a file called command1.java. After compilation, run the program as given in the figure above. Bytecode file is not a command line argument. The first argument one is stored in args [0], second command line argument two is stored in args [1] and so on. In the program using for loop, we have displayed all command line arguments.

```
/*PROG 2.17 FINDING MAXIMUM USING COMMAND LINE ARGUMENTS */

class JPS16
{
    public static void main(String[] args)
    {
        int max = 0, t;
        try
        {
            max = Integer.parseInt(args[0]);
            for (int i = 0; i < args.length; i++)
            {
                t = Integer.parseInt(args[i]);
                if (max < t)
                    max = t;
            }
        }
        catch (Exception e)
        {
            System.out.println("PARSING ERROR!!!!");
        }
    }
}
```

```

        System.out.println("Max is" + max);
    }
}

OUTPUT:
C:\JPS\ch2>javac command2.java
C:\JPS\ch2>java JPS16 30 56 23 45 67
Max is 67

```

Explanation: In the program above, we supply integers at command line. But all command line arguments are string objects, so we need to parse them into integers. The first argument is converted to an integer and assumed to be maximum and stored in max. The rest of the arguments are converted to integers and are compared to the max. If any of the argument is greater than maximum, then it becomes name command2.java.

2.15 USING CONSTANTS

In Java, constants can be created by using the final keyword. For example:

```

final int MAX= 105;
final float PI = 3.14;

```

Each of the above line declares a constant of type `int` and `float` type. You cannot change the value of a constant. You can only use it. The `final` keyword can also be used for creating a final method and a final class. This will be discussed later in the book.

2.16 BYTCODE AND JVM

2.16.1 What is the Java Virtual Machine? Why is it Here?

The JVM is an abstract computer that runs compiled Java programs. The JVM is ‘virtual’ because it is generally implemented in software on top of a ‘real’ hardware platform and operating system. All Java programs are compiled for the JVM. Therefore, the JVM must be implemented on a particular platform before compiled Java programs will run on that platform (Figure 2.10).

The JVM plays a central role in making Java portable. It provides a layer of abstraction between the compiled Java program and the underlying hardware platform and operating system. The JVM is central to Java’s portability because compiled Java programs run on the JVM, independent of whatever may be underneath a particular JVM implementation.

What makes the JVM lean and mean? The JVM is lean because it is small when implemented in software. It was designed to be small so that it can fit in as many places as possible like TV sets, cell phones and personal computers. The JVM is mean because of its ambition. ‘Ubiquity!’ is its battle cry. It wants to be everywhere, and its success is indicated by the extent to which programs written in Java will run everywhere.

2.16.2 What are Java Bytecodes?

Java programs are compiled into a form called Java bytecodes (Figure 2.11). The JVM executes Java bytecodes. So, Java bytecodes can be thought of as the machine language of the JVM.

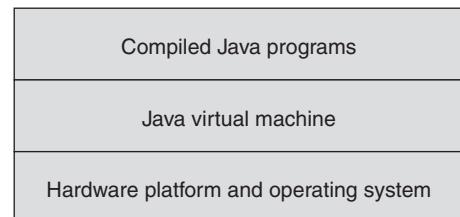


Figure 2.10 Location of JVM

The Java compiler reads Java language source (.java) files, translates the source into Java bytecodes and places the bytecodes into class (.class) files. The compiler generates one class file per class in the source.

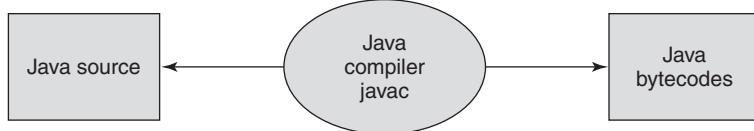


Figure 2.11 How Java compiler converts Java source into Java bytecodes

To the JVM, a stream of bytecodes is a sequence of instructions. Each instruction consists of a one-byte opcode and zero or more operands. The opcode tells the JVM what action to take. If the JVM requires more information to perform the action than just the opcode, the required information immediately follows the opcode as operands.

A mnemonic is defined for each bytecode instruction. The mnemonics can be thought of as an assembly language for the JVM. For example, there is an instruction that will cause the JVM to push a zero onto the stack. The mnemonic for this instruction is `iconst_0`, and its bytecode value is 60 hex. This instruction takes no operands. Another instruction causes program execution to unconditionally jump forwards or backwards in memory. This instruction requires one operand, a 16-bit signed offset from the current memory location. By adding the offset to the current memory location, the JVM can determine the memory location to jump to. The mnemonic for this instruction is `goto`, and its bytecode value is a7 hex (Figure 2.12).

60	<code>iconst_0</code> (opcode)	a7	<code>goto</code> (opcode)
	(no operands)	ff	(one two-byte operand, fff9, which defines an offset to jump to)
		f9	

Figure 2.12 Mnemonic instructions representation

2.16.3 Virtual Parts

The ‘virtual hardware’ of the JVM can be divided into four basic parts: the registers, the stack, the garbage-collected heap and the method area. These parts are abstract, just like the machine they compose, but they must exist in some form in every JVM implementation.

The minimum size of a word in the JVM is 32 bits. Each register in the JVM stores one word. The stack, the garbage-collected heap and the method area reside somewhere within the JVM’s addressable memory. The exact location of these memory areas is a decision of the implementor of each particular JVM.

A word in the JVM is 32 bits. The JVM has a small number of primitive data types: `byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits), `float` (32 bits), `double` (64 bits) and `char` (16 bits). With the exception of `char`, which is an unsigned unicode character, all the numeric types are signed. These types conveniently map to the types available to the Java programmer. One other primitive type is the object handle, which is a 32-bit address that refers to an object on the heap.

The method area, because it contains bytecodes, is aligned on byte boundaries. The stack and garbage-collected heap are aligned on word (32-bit) boundaries.

2.16.4 The Proud, the Few, the Registers

The JVM has a program counter and three registers that manage the stack. It has few registers because the bytecode instructions of the JVM operate primarily on the stack. This stack-oriented design helps keep the JVM’s instruction set and implementation small.

The JVM uses the program counter, or pc register, to keep track of where in memory it should be executing instructions. The other three registers—optop register, frame register and vars register—point to various parts of the stack frame of the currently executing method. The stack frame of an executing method holds the state (local variables, intermediate results of calculations, etc.) for a particular invocation of the method.

2.16.5 The Method Area and the Program Counter

The method area is where the bytecodes reside. The program counter always points to (contains the address of) some byte in the method area. The program counter is used to keep track of the thread of execution. After a bytecode instruction has been executed, the program counter will contain the address of the next instruction to execute. After execution of an instruction, the JVM sets the program counter to the address of the instruction that immediately follows the previous one, unless the previous one specifically demanded a jump.

2.16.6 The Java Stacks and Related Registers

The Java stack is used to store parameters for and results of bytecode instructions to pass parameters to and return values from methods and to keep the state of each method invocation. The state of a method invocation is called its stack frame. The vars, frame and optop registers point to different parts of the current stack frame.

There are three sections in a Java stack frame: the local variables, the execution environment and the operand stack. The local variables section contains all the local variables being used by the current method invocation. It is pointed to by the vars register. The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the frame register. The operand stack is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed and results of bytecode instructions are found. The top of the operand stack is pointed to by the optop register.

The execution environment is usually sandwiched between the local variables and the operand stack. The operand stack of the currently executing method is always the topmost stack section, and the optop register, therefore, always points to the top of the entire Java stack.

2.16.7 The Garbage-collected Heap

The heap is where the objects of a Java program live. Any time you allocate memory with the new operator, that memory comes from the heap. The Java language does not allow you to free allocated memory directly. Instead, the run time environment keeps track of the references to each object on the heap and automatically frees the memory occupied by objects that are no longer referenced. This process is called garbage collection.

2.16.8 What is in a Class File?

The Java class file contains everything a JVM needs to know about one Java class or interface. In their order of appearance in the class file, the major components are: magic, version, constant pool, access flags, this class, super class, interfaces, fields, methods and attributes.

Information stored in the class file often varies in length, that is, the actual length of the information cannot be predicted before loading the class file. For instance, the number of methods listed in the methods component can differ among the class files, because it depends on the number of methods defined in the source code. Such information is organized in the class file by prefacing the actual information by its size or length. This way, when the class is being loaded by the JVM, the size of variable-length information is read first. Once the JVM knows the size, it can correctly read in the actual information.

Information is generally written to the class file with no space or padding between consecutive pieces of information; everything is aligned on byte boundaries. This helps keep the class files petite so that they will be aerodynamic as they fly across networks.

The order of class file components is strictly defined so that JVMs can know what to expect, and where to expect it, when loading a class file. For example, every JVM knows that the first eight bytes of a class file contain the magic and version numbers, the constant pool starts on the ninth byte and the access flags follow the constant pool. But because the constant pool is variable length, it does not know the exact whereabouts of the access flags until it has finished reading in the constant pool. Once it has finished reading in the constant pool, it knows the next two bytes will be the access flags.

2.17 WHY UNDERSTAND BYTECODE?

Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs. The most knowledgeable C and C++ programmers know the assembler instruction set of the processor for which they are compiling. This knowledge is crucial when debugging and doing performance and memory usage tuning. Knowing the assembler instructions that are generated by the compiler for the source code you write helps you know how you might code differently to achieve memory or performance goals. In addition, when tracking down a problem, it is often useful to use a debugger to disassemble the source code and step through the assembler code that is executing.

An often overlooked aspect of Java is the bytecode that is generated by the javac compiler. Understanding the bytecode and what bytecode is likely to be generated by a Java compiler helps the Java programmer in the same way that knowledge of assembler helps the C or C++ programmer.

The bytecode is your program. Regardless of a JIT or Hotspot run time, the bytecode is an important part of the size and execution speed of your code. Consider that the more bytecode you have, the bigger the .class file is and the more code that has to be compiled by a JIT or Hotspot run time.

2.17.1 How to Get Bytecode?

For generating bytecode, the following steps need to be followed:

Step 1: Develop a Java program and save the program by using .java extension.

```

        catch (Exception eobj)
        {
            System.out.println("ERROR!!!!");
        }
    }
}

```

We have saved this program as JPS8.java.

Step 2: Now compile the above program by using javac compiler

```
javac JPS8.java
```

Step 3: After the successful compilation, apply the following statement: `javap -c JPS8 > JPS8.bc` (Figure 2.13).

Step 4: Show the bytecode (Figure 2.14).

```

E:\WINDOWS\system32\cmd.exe

C:\JPS\ch2>javac JPS8.java
Note: JPS8.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\JPS\ch2>java JPS8

```

Figure 2.13 Showing the successful compilation process

```

E:\WINDOWS\system32\cmd.exe

C:\JPS\ch2>javap -c JPS8 > JPS8.bc
C:\JPS\ch2>

```

Figure 2.14 Execution of javap for getting bytecode

```

Compiled from "JPS8.java"
class JPS8 extends java.lang.Object{
JPS8();
    Code:
    0:aload_0
    1:invokespecial    #1; //Method java/lang/Object."<init>":()V
    4:return
public static void main(java.lang.String[]);
    Code:
    0:new             #2; //class java/io/DataInputStream
    3:dup
    4:getstatic #3; //Field java/lang/System.in:Ljava/io/InputStream;
    7:invokespecial   #4; //Method
java/io/DataInputStream."<init>":(Ljava/io/InputStream;)V
    10:astore_3
    11:getstatic      #5; //Field java/lang/System.out:Ljava/io/PrintStream;

```

```

14:ldc          #6; //String \n
16:invokevirtual #7; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
19:getstatic     #5;    //Field    java/lang/System.out:Ljava/io/
PrintStream;
22:ldc          #8; //String Enter a number
24:invokevirtual #7; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
27:aload_3
28:invokevirtual #9; //Method
java/io/DataInputStream.readLine:()Ljava/lang/String;
31:astore_1
32:aload_1
33:invokestatic #10; //Method
java/lang/Integer.parseInt:(Ljava/lang/String;)I
36:istore_2
37:getstatic     #5;    //Field    java/lang/System.out:Ljava/io/
PrintStream;
40:new          #11; //class java/lang/StringBuilder
43:dup
44:invokespecial #12; //Method java/lang/StringBuilder."<init>":()V
47:ldc          #13; //String You have enetered: =
49:invokevirtual #14; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/
StringBuilder;
52:iload_2
53:invokevirtual #15; //Method
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
56:invokevirtual #16; //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
59:invokevirtual #7; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
62:goto         74
65:astore_3
66:getstatic     #5; //Field java/lang/System.out:Ljava/io/PrintStream;
69:ldc          #18; //String ERROR!!!!
71:invokevirtual #7; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
74:return
Exception table:
from to target type
 0 62 65 Class java/lang/Exception
}

```

2.17.2 Implementation of Bytecode Works

```

javac Employee.java
javap -c Employee > Employee.bc
Compiled from Employee.java
class Employee extends java.lang.Object {
public Employee(java.lang.String,int);

```

```

public java.lang.String employeeName();
public int employeeNumber();
}

Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object()>
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void
storeData(java.lang.String, int)>
20 return

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn

Method int employeeNumber()
0 aload_0
1 getfield #4 <Field int idNumber>
4 ireturn

Method void storeData(java.lang.String, int)
0 return

```

This class is very simple. It contains two instance variables, a constructor and three methods. The first five lines of the bytecode file list the file name that is used to generate this code, the class definition, its inheritance (by default, all classes inherit from `java.lang.Object`) and its constructors and methods. Next, the bytecode for each of the constructors is listed. Then, each method is listed in alphabetical order with its associated bytecode.

You might notice on closer inspection of the bytecode that certain opcodes are prefixed with an ‘a’ or an ‘i.’ For example, in the Employee class constructor, you see `aload_0` and `iload_2`. The prefix is representative of the type that the opcode is working with. The prefix ‘a’ means that the opcode is manipulating an object reference. The prefix ‘i’ means the opcode is manipulating an integer. Other opcodes use ‘b’ for byte, ‘c’ for char, ‘d’ for double, etc. This prefix gives you immediate knowledge about what type of data is being manipulated.

Details: To understand the details of the bytecode, we need to discuss how a JVM works regarding the execution of the bytecode. A JVM is a stack-based machine. Each thread has a JVM stack which stores *frames*. A frame is created each time a method is invoked, and it consists of an operand stack, an array of local variables and a reference to the run time constant pool of the class of the current method. Conceptually, it might look like this (Figure 2.15):

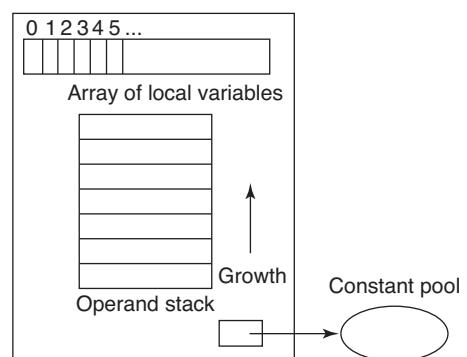


Figure 2.15 A simple design of frame

The array of local variables, also called the local variable table, contains the parameters of the method and also used to hold the values of the local variables. The parameters are stored first, beginning at index 0. If the frame is for a constructor or an instance method, the reference is stored at location 0. Then location 1 contains the first formal parameter, location 2 contains the second and so on. For a static method, the first formal method parameter is stored in location 0, the second in location 1 and so on.

The size of the array of local variables is determined at compile time and is dependent on the number and size of the local variables and formal method parameters. The operand stack is a LIFO stack used to push and pop values. Its size is also determined at compile time. Certain opcode instructions push values onto the operand stack; others take operands from the stack, manipulate them and push the result. The operand stack is also used to receive return values from methods.

```
public String employeeName()
{
    return name;
}

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn
```

The bytecode for this method consists of three opcode instructions. The first opcode, `aload_0`, pushes the value from index 0 of the local variable table onto the operand stack. Earlier, it was mentioned that the local variable table is used to pass parameters to methods. The `this` reference is always stored at location 0 of the local variable table for constructors and instance methods. The `this` reference must be pushed because the method is accessing the instance data, name of the class.

The next opcode instruction, `getfield`, is used to fetch a field from an object. When this opcode is executed, the top value from the stack, `this`, is popped. Then the #5 is used to build an index into the run time constant pool of the class where the reference to name is stored. When this reference is fetched, it is pushed onto the operand stack.

The last instruction, `areturn`, returns a reference from a method. More specifically, the execution of `areturn` causes the top value on the operand stack, the reference to name, to be popped and pushed onto the operand stack of the calling method.

The `employeeName` method is fairly simple. Before looking at a more complex example, we need to examine the values to the left of each opcode. In the `employeeName` method's bytecode, these values are 0, 1 and 4. Each method has a corresponding bytecode array. These values correspond to the index into the array where each opcode and its arguments are stored. You might wonder why the values are not sequential. Since bytecode got its name, each instruction occupies one byte, why are the indexes not 0, 1 and 2? The reason is some of the opcodes have parameters that take up space in the bytecode array. For example, the `aload_0` instruction has no parameters and naturally occupies one byte in the bytecode array. Therefore, the next opcode, `getfield`, is in location 1. However, `areturn` is in location 4. This is because the `getfield` opcode and its parameters occupy location 1, 2 and 3. Location 1 is used for the `getfield` opcode; location 2 and 3 are used to hold its parameters. These parameters are used to construct an index into the run time constant pool for the class to where the value is stored. The following diagram shows what the bytecode array looks like for the `employeeName` method (Figure 2.16):

0	1	2	3	4
aload_0	getfield	00	05	areturn

Figure 2.16 Bytecode array for `employeeName` method

Actually, the bytecode array contains bytes that represent the instructions. Looking at a .class file with a hex editor, the following values can be seen in the bytecode array:

0	1	2	3	4
2A	B4	00	05	B0

Figure 2.17 Values in the bytecode array

2A, B4 and B0 correspond to `aload_0`, `getfield` and `areturn`, respectively (Figure 2.17).

```

public Employee(String strName, int num)
{
    name = strName;
    idNumber = num;
    storeData(strName, num);
}

Method Employee(java.lang.String,int)
0  aload_0
1  invokespecial #3 <Method java.lang.Object()>,
4  aload_0
5  aload_1
6  putfield #5 <Field java.lang.String name>
9  aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14  aload_0
15  aload_1
16  iload_2
17  invokespecial #6 <Method void
storeData(java.lang.String, int)>
20  return

```

The first opcode instruction at location 0, `aload_0`, pushes the `this` reference onto the operand stack. (Remember, the first entry of the local variable table for instance methods and constructors is the `this` reference.)

The next opcode instruction at location 1, `invokespecial`, calls the constructor of this class's superclass. Because all classes that do not explicitly extend any other class implicitly inherit from `java.lang.Object`, the compiler provides the necessary bytecode to invoke this base class constructor. During this opcode, the top value from the operand stack, `this`, is popped.

The next two opcodes, at locations 4 and 5, push the first two entries from the local variable table onto the operand stack. The first value to be pushed is the `this` reference. The second value is the first formal parameter to the constructor, `strName`. These values are pushed in preparation for the `putfield` opcode instruction at location 6.

The `putfield` opcode pops the two top values off the stack and stores a reference to `strName` into the instance data name of the object referenced by `this`.

The next three opcode instructions at locations 9, 10 and 11 perform the same operation with the second formal parameter to the constructor, `num`, and the instance variable, `idNumber`.

The next three opcode instructions at locations 14, 15 and 16 prepare the stack for the `storeData` method call. These instructions push the `this` reference, `strName` and `num`, respectively. The `this` reference must be pushed because an instance method is being called. If the method was declared static, the `this` reference would not need to be pushed. The `strName` and `num` values are pushed since they

are the parameters to the storeData method. When the storeData method executes, the **this** reference, strName and num will occupy indexes 0, 1 and 2, respectively, of the local variable table contained in the frame for that method.

2.18 THE JAVA PLATFORM

A *platform* is the hardware or software environment in which a program runs. We have already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it is a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components: the JVM and the API.

The JVM has been earlier mentioned; it is the base for the Java platform and is ported onto various hardware-based platforms.

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages* (Figure 2.18).

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in the compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

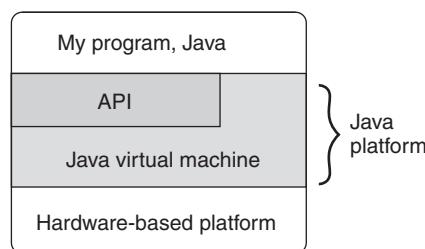


Figure 2.18 The API and JVM insulate the program from the underlying hardware

2.19 JAVA, INTERNET AND WWW

Computers can be connected together on networks. A computer on a network can communicate with other computers on the same network by exchanging the data and files or by sending and receiving the messages. Computers on a network can even work together on a large computation. Today, millions of computer throughout the world are connected to a single huge network called Internet. Internet is the network of networks. New computers are being connected to the Internet every day. In fact, a computer can join the Internet temporarily by using a modem to establish a connection through telephone lines leased lines, through VSAT, etc.

The www is based on pages which can contain information of many different kinds as well as links to other pages. These pages are viewed with a web browser program such as Netscape or Internet Explorer. Many people seem to think that the WWW is the Internet, but it is really just a graphical user interface to the Internet. The pages that you view with a web browser are just files that are stored on computers connected to the Internet. When you tell your web browser to load a page, it contacts the computer on which the page is stored and transfers it to your computer using a protocol known as Hyper Text Transfer Protocol (HTTP). Any computer on the Internet can publish pages on the www. When you use a web browser, you have access to a huge sea of interlinked information that can be navigated with no special computer expertise. The web is the most exciting part of the Internet and is driving the Internet to a truly phenomenal rate of growth. Nowadays, the web has become a universal and fundamental part of every day life.

Java is intimately associated with the Internet and the www. Special Java programs called applets are meant to be transmitted over the Internet and displayed on web pages. A web server transmits a Java applet just as it would transmit any other type of information. A web browser understands Java, that is,

that includes an interpreter for the JVM and can then run the applet right on the web page. Since applets are programs, they can do almost anything, including complex interaction with the user. With Java, a web page becomes more than just a passive display of information. It becomes anything that programmers can imagine and implement. For the Internet and networking Java has special support.

2.20 JDK TOOLS

There are a number of tools shipped with the JDK which are discussed below:

1. *javac*: The *javac* stands for Java compiler. It is provided as an executable file in the bin directory of JDK. The *javac* tools read and interface definitions, written in the Java programming language, and compiles them into the bytecode class files. The most common usage is to compile single Java source file (Figure 2.19) as:

```
javac filename.java
```

Figure 2.19 Compilation of *JPS8.java* without using *nowarn* option; two warning messages are displayed

This, on compilation, creates the *filename.class* file provided the program is error free. You can compile multiple files together by separating the file names using a space or tab as:

```
javac file1.java file2.java file3.java
```

- If you want to put *.class* file in another directory, after the compilation you can set the *-d* option. The syntax is given as:

```
javac -d dirname file1.java
```

On compilation, the *file1.class* file will be placed in the directory *dirname*. This comes handy when you want to compile a number of files together and put the files in a directory.

- Another option is you can use is *-classpath*. Later we will discuss this option in depth.
- The *nowarn* option hides warning message (Figure 2.20). One such example of warning message is when you use *readLine* method with an object of the *DataInputStream* class. To suppress the message, you can compile the program as (assume that the file name is *file1.java* which used the *readLine* method):

```
javac -nowarn file1.java
```

Figure 2.20 *JPS8.java* with *nowarn* option; no warning message is displayed

- (d) The `-deprecation` option tells which methods or classes have been deprecated. This can be used as (Figure 2.21):

```
javac -deprecation file1.java
```

```
E:WINDOWS\system32\cmd.exe
C:\JPS\ch2>javac -deprecation JPS8.java
JPS8.java:15: warning: [deprecation] readLine() in java.io.DataInputStream has been deprecated
    snum = input.readLine();
                                         ^
1 warning
C:\JPS\ch2>
```

Figure 2.21 JPS8.java with deprecation option

The run-time view is shown in the image given below which uses nowarn and deprecation option for a file JPS8.java

2. *java*: The *java* is the Java interpreter. Similar to the *javac*, it is provided as executable file within the *JDK\bin* directory. The Java tool is used to run the Java application. It does this by starting a Java run-time environment, loading a specified class and invoking that class's main method. No specific option will be covered in this book.
3. *jdb*: The Java Debugger, *jdb*, is a simple command line debugger for the Java classes. The *jdb* helps you find and fix bugs in Java language programs.
4. *javadoc*: The *javadoc* tool parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing (by default) the public and protected classes, nested classes, interfaces, constructors, methods and fields. You can use it to generate the API documentation or the implementation documentation for a set of source files.
5. *javap*: The *javap* command disassembles a class file.
6. *javah*: The *javah* produces C header files and C source files from a Java class. These files provide the connective links that allows your Java and C code to interact.

2.21 PONDERABLE POINTS

1. Java was developed by James Gosling at Sun Microsystems around 1991. Its original name was 'Oak.' The official name 'Java' was announced in 1995.
2. The first complete application developed in Java was a web browser named 'Hot Java.'
3. Java was mainly written for the Internet, but it can be used for writing console applications.
4. Java is both compiled and interpreted. All Java programs have the extension `.java`. The Java source file is compiled by the Java compiler which produces bytecode file with `.class` extension. This file is platform-independent and can be run by a Java interpreter anywhere on any machine.
5. Java supports unicode character set which is of 16 bits. This character code supports almost all different types of symbols present in all types of language of the world.
6. All primitive data types in Java are platform-independent, i.e., their size remains same on every operating system.
7. Java is a pure object-oriented programming language, i.e., even a single line of code cannot go outside from the class.
8. Bytecode is a machine independent code which is produced for the JVM.
9. The JVM is a virtual machine purely implemented in the form of a software program and not in hardware.

10. JDK stands for Java Development Kit which can be freely downloaded from the Internet. For running Java programs, JDK must be installed.
11. javac is the Java compiler and java is the Java interpreter.
12. In Java, no .exe file is generated; this is to support platform-independent and make it architectural neutral.

REVIEW QUESTIONS

1. What are the key features of Java?
2. Explain the concept of bytecode in Java. Also explain why Java does not support the concept of '.exe' file.
3. What is JVM? Why Java has both compiler and interpreter? Justify.
4. What is JDK? Explain the various tools of JsDK in short.
5. 'Java is free form,' justify the statement.
6. Translate this ADT into a Java interface:

```
ADT: Point
Amplitude(): Real
distanceTo(Point): Real
equals(Point): Boolean
magnitude(): Real
toString(): String
xCoordinate(): Real
yCoordinate(): Real
```

7. Explain the purpose of nextBoolean() method.
8. What is command line argument? Explain the benefits of using command line arguments.
9. Explain the working of Java stacks. How it is useful for the bytecodes?
10. How the JVM works? Also explain the working of Hypervisor used in X86 architecture.
11. Why we use Java Application Programming Interface (API)? Explain with the help of an example.
12. Explain the execution of the bytecode in Java.
13. How garbage collection heap works? How it is useful?
14. What is virtualization? How JVM performed virtualization for Java code?
15. What are comments? Describe the various comments used in Java.
16. What are the tags supported by javadoc?
17. What is the use of following JDK tools?

(a) javap	(c) jdb
(b) javadoc	(d) javah
18. Explain the major uses of jar tools available in Java.
19. Write a program to print '*'Happy New Year XYZ.'* The *XYZ* is a year; it should be given interactively through the keyboard.
20. Write a program to print
'Failure is the pillar of success.'
21. What is token? State the Java character set.
22. Write a short note on keyword.
23. What is an identifier? What are the rules to code it?
24. Which of the following are valid variable name?

(a) id2	(e) \$HARI
(b) 2cons	(f) MPSTME NMIMS
(c) _5har	(g) 567
(d) #var1	(h) switch
25. What is a constant? How it is classified?
26. What is the difference between character constant and string constant?
27. Distinguish between the bytecode and Unicode character set.
28. Which of the following will compile without warning or error?

(a) float f = 2.7	(b) char c = 'x'
(c) byte B = 257	(d) boolean bo = null;
(e) int count = 0	(f) boolean flag = true
29. What will be the result for this code if it is run with the following command line argument?

```
class test1
{
    public static void main(String args[])
    {
        System.out.println(arg[3]);
    }
}
```

Multiple Choice Questions

1. Java variables cannot start with
 - (a) a number
 - (c) a character
 - (b) an alphabet
 - (d) none of the above
2. Identifiers are
 - (a) user defined names
 - (b) reserved key words
 - (c) Java statements
 - (d) none of the above
3. The variable name can be started with
 - (a) underscore symbol (_)
 - (b) dollar symbol (\$)
 - (c) ampersand symbol (and)
 - (d) none of the above
4. How many variables can be initialized at a time?
 - (a) One
 - (c) Five
 - (b) Two
 - (d) Any number of variables
5. In Java every variable has
 - (a) a type
 - (d) a size
 - (b) a name
 - (e) all of the above
 - (c) a value
6. Which escape sequence does not have any specific meaning?
 - (a) '\t'
 - (c) '\b'
 - (b) '\a'
 - (d) '\c'
7. Which of these is an invalid identifier?
 - (a) wd-count
 - (c) wdcountbce
 - (b) wd_count
 - (d) w4count
8. Who developed Java language?
 - (a) Ken Thomson
 - (b) Bjarne Stroustrup
 - (c) Dennis Ritchie
 - (d) James Gosling
9. Java is a pure object-oriented programming language because
 - (a) Java is platform-independent
10. Java is both compiled and interpreted
 - (b) Java is both compiled and interpreted
 - (c) a single line of code cannot go outside from the class
 - (d) none of the above
11. The next () method is used for
 - (a) reading an integer
 - (b) reading a byte
 - (c) reading a long
 - (d) reading a string, but included white space characters
12. Java supports Unicode character set which is
 - (a) 20 bits
 - (c) 64 bits
 - (b) 32 bits
 - (d) 16 bits
13. The Scanner class is defined in
 - (a) java.util package
 - (b) java.applet package
 - (c) java.awt package
 - (d) none of the above
14. Java is an object-oriented language which is
 - (a) platform independent language
 - (b) a pure object-oriented language.
 - (c) free form language
 - (d) all of the above
15. Which is not correct?
 - (a) jdb is a simple command line debugger for the Java classes.
 - (b) javac-nowarn option hides warning message.
 - (c) java is used to compile and run Java program.
 - (d) javap command disassembles a class file.
16. _____ command disassembles a class file.
 - (a) javac
 - (c) javah
 - (b) javap
 - (d) jdb

KEY FOR MULTIPLE CHOICE QUESTIONS

- | | | | | | | | |
|------|-------|-------|-------|-------|-------|-------|------|
| 1. a | 2. a | 3. a | 4. d | 5. e | 6. d | 7. a | 8. d |
| 9. c | 10. d | 11. d | 12. a | 13. d | 14. c | 15. b | |

Introduction to Operators and Expression in Java

3

3.1 INTRODUCTION

For performing different kinds of operations, various types of operators are required. An operator denotes an operation to be performed on some data that generates some value. For example, a plus operator (+) on 3 and 4 generates 7 ($3 + 4 = 7$). Here, 3 and 4 are called operands.

Java is rich in built-in operators. The four main classes of operators are arithmetic, relational, logical and bitwise.

Apart from these, there are many other operators. A list of operators provided by Java is given in Table 3.1.

Operator	Symbolic Representation
Arithmetic	+, -, /, *, %
Logical	&&, , !
Relational	>, <, <=, >=, ==, !=
Assignment	=
Increment	++
Decrement	--
Comma	,
Conditional	? :
Bitwise	&, , ^, !, >>, <<, >>>
Instanceof	

Table 3.1 Operators in Java

3.1.1 Binary Operators

Operators which require two operands to operate on, are known as binary operators. For example, as shown in Table 3.1, the arithmetic, relational, logical (except ‘!’ (NOT) operator), comma, assignment, bitwise (except ~ operator), etc., are binary operators.

For example: $2 + 4$, $34 > 45$, $x = 23$, 2 and 5

3.1.2 Unary Operators

Operators which require only one operand to operate on are known as unary operators. For example, as shown in Table 3.1, increment/decrement! (NOT), ~ (1's complement operator), sizeof, etc., are unary operators. C also provides unary plus and unary minus, i.e., $+20$ and -34 . Here, $+$ and $-$ are known as unary plus operator and unary minus operator, respectively. The unary plus operator does not change the meaning of the operands, i.e., writing $+x$ has no effect on the value of x whereas, unary minus changes the sign of the operand, i.e., if $x = 10$, then $-x$ changes the value of x from 10 to -10.

For example: $+4$, 3, $+x$, x .

NOTE: ? operator is known as ternary operator as it requires three operands to operate on: One before '?', second after '?' and third after ':'

3.1.3 Expressions

An operator together with operands constitutes an expression or a valid combination of constants, variables and operators forms an expression which is usually recognized by the type of operator used within it. Depending on whether it is integer, floating point or relational expression, etc., one may have different types of operators in an expression—called a mixed mode expression. See Table 3.2 for examples.

Expression	Type of Expression
$2 + 3 * 4 / 6 - 7$	Integer Arithmetic
$2.3 * 4.7 / 6.0$	Real Arithmetic
$a > b! = c$	Relational
$X \&& 10 \parallel y$	Logical
$2 > 3 + x \&& y$	Mixed

Table 3.2 Types of expressions

3.2 ARITHMETIC OPERATORS

As given in Table 3.2, there are mainly five arithmetic operators in Java: $+$, $-$, $*$, $/$ and $\%$, which are used for addition, subtraction, multiplication, division and remainder, respectively. Table 3.3 presents arithmetic operators.

The operators have their inherent meanings in mathematics. The main point to note when a division is performed with an integer operand is that the result will be an integer. If any of the operand is a floating point the answer will also be in floating point. The percentage symbol ' $\%$ ' is used for modulus of a number. a/b produces quotient and $a\%b$, where $\%$ is called **remainder operator**, produces remainder when a is divided by b . The remainder operator $\%$ can be used with integer as well as with floating points. The mathematical formula behind remainder operator is:

$$a \% b = a - (a/b) * b \quad (\text{where } a/b \text{ is integer division})$$

Operator	Meaning / Used for
$+$	Addition
$-$	Subtraction
$/$	Division
$*$	Multiplication
$\%$	Remainder

Table 3.3 Arithmetic operators

For example, $a = 13$ and $b = 5$

$$\begin{aligned} 13 \% 5 &= 13 - (13/5) * 5 \\ &= 13 - (2) * 5 \\ &= 13 - 10 \\ &= 3 \end{aligned}$$

As another example with floating point data consider $a = 5.3f$ and $b = 2.4f$

$$\begin{aligned} 5.3 \% 2.4 &= 5.3 - (5.3/2.4) * 2.4 \\ &= 5.3 - (2) * 2.4 \\ &= 5.3 - 4.8 \\ &= 0.5 \end{aligned}$$

Few programs can be presented based on these operators.

```
/*PROG 3.1 DEMO OF ARITHMETIC OPERATORS */

class JPS1
{
    public static void main(String args[])
    {
        float num1=13.5f, num2=7.0f;
        float a,b,c,d,e;
        a=num1+num2;
        b=num1-num2;
        c=num1*num2;
        d=num1/num2;
        e=num1%num2;
        System.out.println("Sum of num1 & num2 :="+a);
        System.out.println("Sub of num1 & num2 :="+b);
        System.out.println("Mul of num1 & num2 :="+c);
        System.out.println("Div of num1 & num2 :="+d);
        System.out.println("Mod of num1 & num2 :="+e);
    }
}

OUTPUT:
Sum of num1 & num2 :=20.5
Sub of num1 & num2 :=6.5
Mul of num1 & num2 :=94.5
Div of num1 & num2 :=1.9285715
Mod of num1 & num2 :=6.5
```

Explanation: The program is self-explanatory.

```
/*PROG 3.2 SUM OF TWO NUMBERS, INPUT FROM USER */

import java.io.*;
import java.util.*;
class JPS2
{
    public static void main(String args[])
    {
        int x, y, z;
        Scanner sc = new Scanner(System.in);
        System.out.print("\n Enter first number :=");
        x = sc.nextInt();
        System.out.print("\n Enter second
                        number:=");
        y = sc.nextInt();
        z = x + y;
        System.out.println("\n x:= " + x + "ty:= " + y);
        System.out.println("\n      Sum:= " + z);
    }
}
```

OUTPUT:

```
Enter first number:=15
Enter second number:=20
x:= 15 y:= 20
Sum:= 35
```

Explanation: Two numbers are taken from input. Both the numbers are taken one by one. For taking input, scanner class has been used. This method of taking input has been shown earlier.

```
/* PROG 3.3 DEMO OF MOD OPERATOR, FINDING QUOTIENT AND
REMAINDER */

import java.io.*;
import java.util.*;
class JPS3
{
    public static void main(String args[])
    {
        int num1, num2, rem, quo;
        Scanner sc = new Scanner(System.in);
        System.out.println("\nEnter two integer
                           numbers");
        num1 = sc.nextInt();
        num2 = sc.nextInt();
        quo = num1 / num2;
        rem = num1 % num2;
        System.out.println("\nnum1:=" + num1 + "\tnum2:="
                           + num2);
        System.out.println("\nQuotient:=" + quo);
        System.out.println("\nRemainder:=" + rem);
    }
}
OUTPUT:
Enter two integer numbers
13 5
num1:=13      num2:=5
Quotient: =2
Remainder: =3
```

Explanation: The program is simple to understand. We take two integer numbers as inputs and find division and modulus of two numbers. The numbers and result are then shown.

```
/* PROG 3.4 BEHAVIOUR OF MOD OPERATOR */

import java.io.*;
import java.util.*;
class JPS4
{
```

```

public static void main(String args[])
{
    System.out.println("\n BEHAVIOUR OF MOD OPERATOR\n");
    System.out.println("17%5      := " + (17 % 5));
    System.out.println("-17%5     := " + (-17 % 5));
    System.out.println("17%5      := " + (17 % 5));
    System.out.println("-17%-5    := " + (17 % -5));
}
}

OUTPUT:
BEHAVIOUR OF MOD OPERATOR

17%5      :=2
-17%5     :=-2
17%5      :=2
-17%-5    :=2

```

Explanation: Using % operator if the numerator is negative, the answer will also be negative. This can be verified by the formula given in the program for **-17** and **5**.

$$\begin{aligned}
 -17 \% 5 &= -17 - (-17/5)*5 \\
 &= -17 - (-3) * 5 \\
 &= -17 + 15 \\
 &= -2
 \end{aligned}$$

3.3 RELATION AND TERNARY OPERATOR

The two symbols ‘?’ and ‘:’ together are called ternary or conditional operator. Before ‘?’, condition is specified with the help of relational operators which may be any of the operators as given in the Table 3.4. If the condition specified before ‘?’ is true any expression or statement after ‘?’ is executed, and if the condition is false the statement or expression after ‘:’ is evaluated. The general syntax is:

(Condition)? True part: false part;

All relational operators yield Boolean values, i.e., true or false.

Operator	Meaning/Used for
>	Greater than
<	Less than
> =	Greater than equal to
< =	Less than equal to
= =	Equal to
! =	Not equal to

Table 3.4 Relational operators

An example of the use of ternary operator is given below.

```

String str;
str= 5>0?"Welcome":Bye Bye.....";

```

As the condition `5>0` is true, "Welcome" will be assigned to `str`.

```
/* PROG 3.5 DEMO OF CONDITIONAL OPERATOR */

import java.io.*;
import java.util.*;
class JPS5
{
    public static void main(String args[])
    {
        int num1 = 5, num2 = 7;
        boolean res;
        String str;
        res = (num1 == num2);
        str = res ? "Both are equal\n" : "Both are not equal";
        System.out.println(str);
    }
}

OUTPUT:
Both are not equal
```

Explanation: The expression `a==b` gives **false** in `res`. If it were true, **true** would be in `res`. Before '?' the value is false due to **false** in `res` and so is the output.

```
/* PROG 3.6 FINDING MAXIMUM OF TWO NUMBERS */

import java.io.*;
import java.util.*;
class JPS6
{
    public static void main(String args[])
    {
        String snum;
        int a,b,c,max1, max;
        Scanner sc= new Scanner(System.in);
        System.out.print("\n\nEnter first number :=");
        a=sc.nextInt();
        System.out.print("Enter second number:=");
        b=sc.nextInt();
        System.out.print("Enter third number :=");
        c=sc.nextInt();
        max1=a>b?a:b;
        max=max1>c?max1:c;
        System.out.println("a="+a);
        System.out.println("b="+b);
        System.out.println("c="+c);
        System.out.println("Maximum="+max);
    }
}
```

OUTPUT:

```
Enter first number:=15
Enter second number:=45
Enter third number :=10
    a=15
    b=45
    c=10
Maximun=45
```

Explanation: Initially, maximum of two numbers is found out and stored in the max1. Then, maximum of max1 and c is found out, which gives maximum of three numbers. The maximum can also be found out by nesting of ternary operators as shown below.

max = (a>b? (a>c? a: c) : (b>c? b: c));	A1	A2
---	----	----

In this example, assume two names for the expressions A1 and A2. In the nested condition when $a > b$ is true, **A1** will execute and **A2** is skipped. This means that $a > b$ so next it can be checked whether $a > c$ and this is what **A1** represents. If $a > c$ then the maximum is **a**, else maximum is **c**, which is assigned to **max**.

If $a > b$ fails, it means $b > a$, then **A1** is skipped and **A2** is executed. Falsity of $a > b$ means $b \leq a$ so next it can be checked whether $b > c$ and this is what **A2** represents. In **A2**, if $b > c$ then maximum is **b**, else maximum is **c**, which is assigned to **max**.

```
/*PROG 3.7 DEMO OF RELATIONAL OPERATOR */
```

```
class JPS7
{
    public static void main(String args[])
    {
        int a = 5, b = 6;
        boolean p, q, r, s, t, u;
        p = a > b;
        q = a < b;
        r = (4 != 3);
        s = (8 >= 15);
        t = (130 <= 130);
        u = (0 == 0);
        System.out.println("\n\nvalue of p :=" + p);
        System.out.println("value of q :=" + q);
        System.out.println("value of r :=" + r);
        System.out.println("value of s :=" + s);
        System.out.println("value of t :=" + t);
        System.out.println("value of u :=" + u);
    }
}
```

OUTPUT:

```
value of p :=false
value of q :=true
value of r :=true
value of s :=false
value of t :=true
value of u :=true
```

Explanation: All relational operators return either a **true** or a **false** value. The **true** and **false** value can only be stored in the **Boolean** variables and not in any other types of variables. In the program, depending on the expression **true** or **false** value is assigned to variables **p, q, r, s, t** and **u**.

3.4 LOGICAL OPERATOR

Logical operators are used to check logical relation between two expressions. Depending on the truth or falsehood of the expression they are assigned **true** or false value. The expressions may be variables, constants, functions, etc. These operators always work with the **true** and **false** value. Table 3.5 presents logical operators.

The ‘**&&**’ and ‘**||**’ are binary operators. For ‘**&&**’ to return true value, both of its operands must yield true value. For ‘**||**’ to yield true value, at least one of the operands should yield true value. The **NOT** (!) operator is unary operator. It negates its operand, that is, if the operand is true it converts it into false and vice versa.

In case of logical operators ‘**&&**’ and ‘**||**’, if the left operand yields false value, the right operand is not evaluated by a compiler in a logical expression using ‘**&&**’. If the left operand yields true value, the right operand is not evaluated by the compiler in a logical expression with the operator ‘**||**’. The operators ‘**&&**’ and ‘**||**’ have left to right associativity, hence the left operand is evaluated first and based on the output, the right operand may or may not be evaluated. For example, consider the following expression:

```
int x =10;
boolean b = (10>=15) &&(x++!=4);
System.out.println ("x="+x);
```

The left operand of ‘**&&**’ (**10 > = 15**) is false so the right operand (**x++!=4**) is not evaluated. If it were ‘**||**’ (OR) in the above expression in place of ‘**&&**’, the second operand would have been checked and **x = 11** would be the output.

Due to the aforesaid feature of ‘**&&**’ and ‘**||**’ operators, they are sometimes known as **short circuit operators** as they short-circuit the next operand to be checked.

3.4.1 Logical AND (&&)

The operator works with two operands which may be any expression, variable, constant or function. It checks if both of its operands returns true value or not. If they return true value, and if either of its operands is false, a false value is returned (See Table 3.6).

Symbol	Meaning
&&	AND
	OR
!	NOT

Table 3.5 Logical operators

Operand 1	Operand 2	Returned Value
False	False	False
False	True	False
True	False	False
True	True	True

Table 3.6 Truth table of logical AND

```
/*PROG 3.8 DEMO OF LOGICAL AND (&&) OPERATOR */

class JPS8
{
    public static void main(String args[])
    {
        int num1 = 5, num2 = 7;
        boolean a, b;
        a = (num1 > num2) && (num2 > 5);
        b = (num1 > 2) && (num2 > 3);
```

```

        System.out.println("\nVALUE OF a := " + a);
        System.out.println("VALUE OF b := " + b);
    }
}

OUTPUT:
VALUE OF a :=false
VALUE OF b :=true

```

Explanation: Logical AND returns true when both the conditions and its operand are **true** and also when the first operand returns a **false** value and the second operand is not evaluated. In the program, if the expression `a = (num1 > num2) && (num2 > 5);` returns **false**, `num1 > num2` is **false** and the second condition is not checked. Thus, ‘**a**’ stores **false** as its value. In the expression `b = (num1 > 2) && (num2 > 3);` the first condition `num1 > 2` is true, so the second condition is checked, which also turns out **true**. Thus, ‘**b**’ stores **true** as its value.

3.4.2 Logical OR

The operator works with two operands which may be any expression, variable, constant or function. It checks whether any of its operands returns true value or not. If any one of them is true, it returns true value and if both of its operands are false, false value is returned.

Table 3.7 presents the truth table of logical **OR**.

Operand 1	Operand 2	Returned Value
False	False	False
False	True	True
True	False	True
True	True	True

Table 3.7 Truth table of OR

```

/* PROG 3.9 DEMO OF LOGICAL OR OPERATOR */

class JPS9
{
    public static void main(String args[])
    {
        int a=5,b=6;
        boolean d,c,e,f;
        c=(a>b) || (b>15);
        d=(a>12) || (b>3);
        e=(a>2) || (b>13);
        f=(a>2) || (b>3);
        System.out.println("\nValue of c:=" + c);
        System.out.println("\nValue of d:=" + d);
        System.out.println("\nValue of e:=" + e);
        System.out.println("\nValue of f:=" + f);
    }
}

OUTPUT:
Value of c: =false
Value of d: =true
Value of e: =true
Value of f: =true

```

Explanation: In logical OR, if any of the conditions is **true**, the whole logical OR expression is considered **true**. If the first condition is **true** then the second condition is not evaluated at all. In the above program, of the four expressions, three are **true** and one is **false**.

3.4.3 Logical NOT (!)

The operator converts a true value into false and vice versa (Table 3.8). Again, the operand may be any expression, constant, variable or function.

Operand	Returned Value
False	True
True	False

Table 3.8 Truth table of NOT

```
/*PROG 3.10 DEMO OF LOGICAL NOT OPERATOR */

class JPS10
{
    public static void main(String args[])
    {
        int num1 = 15, num2 = 17;
        boolean check1, check2;
        check1 = !(num1 > 10);
        check2 = !(num1 > 20);
        System.out.println("\n Value of check1:=" + check1);
        System.out.println("\n Value of check2:=" + check2);
    }
}
OUTPUT:
Value of check1:=false
Value of check2:=true
```

Explanation: The NOT operator ‘!’ negates the value; that is, **true** value is converted into **false** and **false** into **true**.

3.5 ASSIGNMENT OPERATOR

The ‘=’ is called assignment operator. Several instances of this operator have been presented in earlier programs. In addition, one use of this operator is the shortening of following types of expressions:

$$x = x + 1, \quad y = y * (x - 5), \quad a = a/10, \quad t = t\%10;$$

In all the above expressions, the variable on both sides of ‘=’ operator is the same. So the above expression can be changed to a shorter form as follows:

$x = x + 1$	$=>$	$x + = 1$
$y = y * (x - 5)$	$=>$	$y * = (x - 5)$
$a = a/10$	$=>$	$x / = 10$
$t = t\%10$	$=>$	$t \% = 10$

This form **op=**, where operator may be any operator is called **compound operator** or **shorthand assignment operator**.

Java also supports **chained assignment**, i.e., assigning a value to one of the variables in the chain and then from this to the next one in the chain and so on. The rightmost value in the chain must be assigned the value as *assignment operator works from right to left*. For example:

```
int x, y, z;
x = y = z = 15;
```

In this example, first 15 will be assigned to z, and then the value of z will be assigned to y and at the end, the value of y will be assigned to x. In terms of chain assignment, it can be thought that after assigning the value 15 to z, it returns a value which is given to next variable in the chain.

3.6 INCREMENT (++) AND DECREMENT (--) OPERATOR

The ‘++’ is known as increment operator and the ‘–’ is known as decrement operator. Both are unary operators. The ‘++’ increments the value of its operand by 1 and ‘–’ decrements the value of its operand by 1. For example: $+x$ becomes 11 (assume $x = 10$, prior to this operation $+x$ and $-x$) and $-x$ becomes 9. The following syntax is given, for example:

```
int x=10;
x++;           //makes x 11;
System.out.println("x =" + x); //prints 11
x--;           //makes x 10;
System.out.println("x=" + x); //prints 10
```

If the operator ‘++’ or ‘–’ is written before the operand, like $-x$ or $+x$, it is known as pre-decrement/pre-increment. In case, it is written after the operand, like $x-$ or $x++$, it is known as post-decrement/post-increment. The difference between the two forms of these operators is visible when single expression is assigned involving these operators to some variable. For example:

```
int x=10;
y = x++;
System.out.println("x =" + x);      //prints 11
System.out.println("y =" + y);      //prints 10
```

In this code, post-increment operator has been used with x, and y is assigned the value. Due to post-increment, first x will be assigned to y, and x will be incremented by 1, i.e., $y = x++$ is equivalent to the following two statements:

```
y = x;
x = x + 1;
```

Now consider the following code:

```
int x=10;
y = ++x;
System.out.println("x =" + x);      //prints 11
System.out.println("y =" + y);      //prints 11
```

In this code, pre-increment operator has been used with x and y is assigned the value. Due to pre-increment, first x will be incremented by 1 and the same incremented value will be assigned to y. $y = +x$, which is equivalent to the following statement:

```
x = x + 1;
y = x;
```

See the program given below for more explanation.

```
/*PROG 3.11 DEMO OF ++ & -- OPERATOR */

class JPS11
{
    public static void main(String args[])
    {
        int x,y;
        x = 10;
        y = ++x;
        --x;
        y--;
        x = y++;
        y = --x;
        x = y++;
        System.out.println("x="+x);
        System.out.println("y="+y);
    }
}

OUTPUT:
x=9
y=10
```

Explanation: For explanation see Table 3.9.

Statement	Value of x	Value of y
$Y = ++x$	11	11
$--x$	10	11
$y--$	10	10
$X = y++$	10	11
$Y = --x$	09	09
$X = y++$	09	10

Table 3.9 Basic implementation of Program 3.11

$y = ++ x;$ is equivalent to $x = x + 1;$ $y = x;$ the value of x is incremented first then it is assigned to $y.$

```
/* PROG 3.12 IMPLEMENTATION OF METHODOLOGY OF ++ AND --
OPERATOR */

class JPS12
{
    public static void main(String args[])
    {
        int x=10;
        System.out.println("\nBEHAVIOUR OF ++ AND
                           -- OPERATOR");
        System.out.println("\n "+(--x)+" "+(++x)+"")
```

```

        " + (x) + " " + (x++) + " " + (x--) );
    }
}

OUTPUT:
BEHAVIOUR OF ++ AND -- OPERATOR
9 10 10 10 11

```

Explanation: The evaluation takes place from left to right. This is shown in the following steps:

1. In $--x$, first x is decremented and then printed, so the output is 9 and the same for the next expression.
2. In $++x$, first x is incremented and then printed, so the output is 10 and the same for the next expression.
3. In x , value of x is printed, so the output is 10 and the same for the next expression.
4. In $x++$, first x is printed and then incremented, so the output is 10 and 11 for the next expression.
5. In $x--$, first x is printed and then decremented, so the output is 10.

3.7 BITWISE OPERATORS

Bitwise operators are called so because they operate on bits. They can be used for the manipulation of bits. Java provides a total of seven types of bitwise operators as shown in Table 3.10.

Operator	Meaning/Used for
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	One's Complement
>>	Right Shift
<<	Left Shift
>>>	Right Shift with zero fill

Table 3.10 Bitwise operators

For all the following programs, the byte data type is only considered. The byte data type is of 8 bits, so the range of possible numbers is from -128 to 127 and numbers are represented using 8 bits.

3.7.1 Bitwise AND (&)

Bitwise AND (**&**) takes two bits as operand and returns the value 1 if both are 1. If either of them is 0, the result is 0 (Table 3.11).

First Bit	Second Bit	Result
0	0	0
0	1	0
1	0	0
1	1	1

For example, consider the following program:

Table 3.11 Truth table of bitwise AND

```
/*PROG 3.13 DEMO OF BITWISE AND (&) OPERATOR */
```

```

class JPS13
{
    public static void main(String args[])

```

```

    {
        byte b1 = 12, b2 = 7;
        byte b3 = (byte) (b1 & b2);
        System.out.println("\nBehavior of Bitwise
                           AND(&) operator");
        System.out.println("\nThe value of b3:=" + b3);
    }
}

OUTPUT:
Behavior of Bitwise AND (&) operator
The value of b3:=4

```

Explanation: For explanation, see the following table, if both the input and output are 1.

b1(12)	0	0	0	0	1	1	0	0
b2(7)	0	0	0	0	0	1	1	1
b1 & b2(4)	0	0	0	0	0	1	0	0

3.7.2 Bitwise OR (|)

Bitwise OR (|) takes two bits as operand and returns the value 1 if at least one operand is 1. If both are 0, only then the result will be 0, else it is 1 (Table 3.12).

For example, consider the following program:

First Bit	Second Bit	Result
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.12 Truth table of bitwise OR

```

/*PROG 3.14 DEMO OF BITWISE OR (|) OPERATOR*/

class JPS14
{
    public static void main(String args[])
    {
        byte b1 = 65, b2 = 42;
        byte b3 = (byte) (b1 | b2);
        System.out.println("\n\nBehavior of Bitwise OR");
        System.out.println("\nValue of b3 := " + b3);
    }
}

OUTPUT:
Behavior of Bitwise OR
Value of b3:=107

```

Explanation: For explanation, see the following table. If any of the input is 1 then the output will be 1.

b1(65)	0	1	0	0	0	0	0	1
b2(42)	0	0	1	0	1	0	1	0
b1 b2(107)	0	1	1	0	1	0	1	1

3.7.3 Bitwise XOR (^)

Bitwise **XOR (^)** takes at least two bits (may be more than two). If the number of 1's are odd then the result is 1, else the result is 0 (Table 3.13).

For example, consider the following program:

First Bit	Second Bit	Result
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.13 Truth table of bitwise XOR (^)

```
/*PROG 3.15 DEMO OF BITWISE XOR (^) OPERATOR */

class JPS15
{
    public static void main(String args[])
    {
        byte b1 = 125, b2 = 107;
        byte b3 = (byte)(b1 ^ b2);
        System.out.println("\n\nBehaviour of Bitwise
                           XOR");
        System.out.println("\nThe value of b3:=" + b3);
    }
}
OUTPUT:

Behaviour of Bitwise XOR
The value of b3:=22
```

Explanation: For **bitwise XOR (^)** operator, if the number of 1's are odd then the result is 1, else the result will be 0. For better explanation, see the following table:

b1 (125)	0	1	1	1	1	1	0	1
b2 (107)	0	1	1	0	1	0	1	1
b1 b2 (22)	0	0	0	1	0	1	1	0

3.7.4 1's Complement (~)

The symbol (~) denotes 1's complement. It is a unary operator and complements the bits in its operand, i.e., 1 is converted to 0 and 0 is converted to 1.

```
/*PROG 3.16 DEMO OF 1'S COMPLEMENT (~)*/

class JPS16
{
    public static void main(String args[])
    {
        byte b1 = 107;
        byte b2 = (byte)~b1;
        System.out.println("\n\nBehaviour of 1's
                           complement");
```

```

        System.out.println("\nThe value of b2 := " + b2);
    }
}
OUTPUT:
Behaviour of 1's complement
The value of b2 := -108

```

Explanation: Representing b1 in binary 8 bits and doing bitwise 1's complement:

b1(107)	0	1	1	0	1	0	1	1	Value is 107
b2(148)	1	0	0	1	0	1	0	0	After 1's complement value is 148 on paper

On running the program, as given above, the answer will be 148, otherwise the answer will be -108 . Both the answers are correct but as the left most bit (bit number 7) is 1 in b2, the number will be negative and the computer has given the answer in 2's complement form. In order to cross check whether the answer is correct or not, find out the binary value of 108, which is as follows:

108	0	1	1	0	1	1	0	0
-----	---	---	---	---	---	---	---	---

Taking 1's complement of the above value:

1	0	0	1	0	0	1	1
---	---	---	---	---	---	---	---

Now adding 1 to it, the result is:

Binary 108	0	1	1	0	1	1	0	0
1's complement of 108	1	0	0	1	0	0	1	1
Adding binary 1	0	0	0	0	0	0	0	1
2's complement of 108	1	0	0	1	0	1	0	0

Now, the binary value of 148 and -108 is the same. So the answer is correct.

3.7.5 Left Shift Operator ($<<$)

Left shift operator is used to shift the bits of its operand towards left. It is written as $x << num$, which means shifting the bits of x towards left by num number of times. A new zero is entered in the least significant bit (LSB) position. For example, consider the following code:

```

/*PROG 3.17 DEMO OF LEFT SHIFT OPERATOR (<<) */

class JPS17
{
    public static void main(String args[])
    {
        byte b1 = 42;
        byte b2 = (byte)(b1 << 1);
        System.out.println("\n\nDemo of Left Shift
                           Operator");
        System.out.println("\nThe value of b2:=" + b2);
    }
}

```

```

    }
}

OUTPUT:

Demo of Left Shift Operator
The value of b2:=84

```

Explanation: The binary representation of 42 in 8 bits.

bits	b7	b6	b5	b4	b3	b2	b1	b0
b1(42)	0	0	1	0	1	0	1	0

In the left, shifting bits are shifted towards left as: **bit b0 moves to b1, b1 to b2 and so on**. A zero is inserted at the b0 position. After shifting bits of b1 once, the resultant bits patterns will be as follows:

bits	b7	b6	b5	b4	b3	b2	b1	b0
b1(84)	0	1	0	1	0	1	0	0

NOTE: Shifting the bits left once multiplies the number by 2, while shifting the bits left by n number of times multiplies the number by 2^n .

3.7.6 Right Shift Operator (>>)

Right shift operator is used to shift the bits of its operand towards right. It is written as `x>>num`, which means shifting the bits of x towards right by num, number of times. A new zero (not necessarily, depends on sign bit) is entered in the most significant bit (MSB) position. See the program given below.

```

/*PROG 3.18 DEMO OF RIGHT SHIFT OPERATOR (>>) */

class JPS18
{
    public static void main(String args[])
    {
        byte b1 = 42;
        byte b2 = (byte)(b1 >> 1);
        System.out.println("\n\nDemo of Right Shift
                           Operator");
        System.out.println("The value of b2:=" + b2);
    }
}

OUTPUT:

Demo of Right Shift Operator
The value of b2:=21

```

Explanation: Look at the binary representation of 42 in 8 bit format as given below.

bits	b7	b6	b5	b4	b3	b2	b1	b0
b1(42)	0	0	1	0	1	0	1	0

In the right, shifting bits are shifted towards right as: **bit b7 moves to b6, b6 to b5 and so on**. A zero is inserted at the **b7** position. It is to be noted that **b7** represents sign bit. For positive numbers, the sign bit is 0 and for negative numbers sign bit. In the right shifting the sign bit remains at its position and propagates too in the right direction. After shifting the bits of b1 once, the resultant bits pattern will be as follows:

bits	b7	b6	b5	b4	b3	b2	b1	b0
b1(21)	0	0	0	1	0	1	0	1

NOTE: Shifting the bits right once divides the number by 2. Shifting the bits right by n number of times divides the number by 2^n .

```
/*PROG 3.19 DEMO OF BITWISE OPERATOR ALL IN A SINGLE PROGRAM */
```

```
class JPS19
{
    public static void main(String args[])
    {
        int x = 2,y = 3, and, or, xor, comp, lshift, rshift;
        and = x & y;
        or = x | y;
        xor = x ^ y;
        comp = ~x;
        lshift = x << 2;
        rshift = (x * 2) >> 1;
        System.out.println("x:=" + x);
        System.out.println("y:=" + y);
        System.out.println("and:=" + and);
        System.out.println("xor:=" + xor);
        System.out.println("comp:=" + comp);
        System.out.println("lshift:=" + lshift);
        System.out.println("rshift:=" + rshift);
    }
}
OUTPUT:
x:=2
y:=3
and:=2
xor:=1
comp:=-3
lshift:=-8
rshift:=2
```

Explanation: Read the theory given above for explanation.

3.7.7 Right Shift with Zero Fill (>>>)

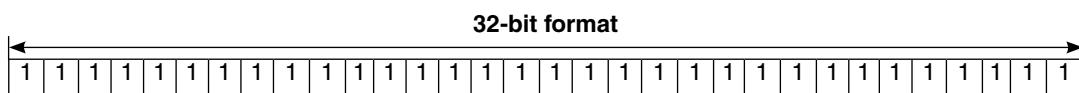
The operator ‘>>’ shifts number towards right and depending on sign bit fills 0 or 1. The ‘>>>’ operator is the unsigned right shift operator. Regardless of sign of number it always fills zero at the left side. The effect of ‘>>>’ operator can only be seen in terms of integer and long data types. In order to understand this concept, look at the program given below.

```
/* PROG 3.20 DEMO OF RIGHT SHIFT WITH ZERO FILL (>>>) OPERATOR */

class JPS20
{
    public static void main(String args[])
    {
        int x = -1;
        int y1 = x >>> 16;
        int y2 = x >> 16;
        System.out.print("\n\nApplying >> on -1,16 times");
        System.out.println(" give " + y2);
        System.out.print("\nApplying >>> on -1,16times");
        System.out.println(" give " + y1);
    }
}

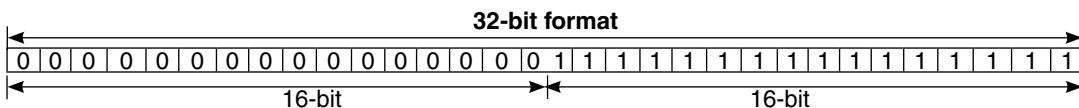
OUTPUT:
Applying >> on -1, 16 times give -1
Applying >>> on -1, 16 times give 65535
```

Explanation: Internally, -1 is represented as all ones. In 32 bit format, it can be shown as follows:



Applying ' $>>$ ' operator of the above binary string sixteen times, shifts the number sixteen times towards right and fills sixteen 1s as the number is $-ve$ and as the left most bit represents the sign so the bit string remains the same. Hence, the output will be -1 .

Applying ' $>>>$ ' operator to -1 sixteen times has the same effect as ' $>>>$ ' but instead of 1s, 0s are inserted towards the left. The binary string therefore becomes as follows:



There are sixteen 1s from the right and all other bits are 0s; so in decimal, the value of the binary string will be $2^{16}-1$, i.e., 65535.

3.8 THE INSTANCEOF OPERATOR

The **instanceOf** operator is used to check which object is of which class. By knowing this, one can call the appropriate method of the class. For example, if one creates an object `d` of class `demo` (say) as `demo d = new demo();` then the expression will be **Boolean b = d instance of demo** stores **true** in **b**. This operator will be used in a number of programs presented in subsequent chapters. In the present context a short program is given below.

```

/* PROG 3.21 DEMO OF INSTANCEOF OPERATOR */

class demo
{}
class JPS21
{
    public static void main(String args[])
    {
        Object top;
        Float objf = 0.0f;
        Integer obji = 15;
        Long objl = 25L;
        top = objf;
        if (top instanceof Float)
            System.out.println("\n of class Float");
        top = obji;
        if (top instanceof Integer)
            System.out.println("\n of class Integer");
        top = objl;
        if (top instanceof Long)
            System.out.println("\n of class Long");
        top = new demo();
        if (top instanceof demo)
            System.out.println("\n of class demo");
    }
}
OUTPUT:
of class Float
of class Integer
of class Long
of class demo

```

Explanation: The class object is the base class for all the classes in the Java class hierarchy, even for the user-defined classes. In the program, there is a reference `obj` of Object class type. Three objects of the class—**Integer**, **Long** and **Float**—are created and initialized as follows.

```

top = objf;
if (top instanceof Float)
System.out.println("\n of class Float");

```

Object **objf** of float class is assigned to a reference of Object class. This is perfectly appropriate as the base class can hold a reference of any derived class and Object is the base class for all classes in Java. The **if** condition checks whether the reference **top** referring to an object of **Float** class which is true so output produced is of class **Float**.

The same discussion applies to **Integer** and **Long** class. At the end, **top** contains reference of user-defined class **demo** and if condition is **true** which prints of **class demo**.

3.9 THE COMMA OPERATOR

The comma is not an operator in Java; it is simply used as a separator for the variable declarations. For example, the following expression used in C and C++ will not work in Java.

```
int x = (2, 5, 6);
System.out.println ("x =" +x);
```

This will generate compilation errors.

Also, in loops, the following code will generate compilation errors.

```
for(i=0, j=10;i<10,j>=10;i++,j--)
```

```
{}
```

In the condition part (*i<10, j>=10*) one cannot use comma as operator.

3.10 THE `sizeof()` OPERATOR

In C and C++, the `sizeof()` operator satisfies a specific need: it tells the number of bytes allocated for the data items. The most compelling need for `sizeof()` in C and C++ is portability. Different data types might be of different sizes on different machines, so the programmer must find out how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store it as 16 bits. Programs could store larger values in integers on the first machine. Probably, portability is a major issue for C and C++ programmers.

Java does not need a `sizeof()` operator for this purpose, because all the data types are of the same size on all machines. There is no need to think about portability on this level—it is designed into the language.

3.11 PRECEDENCE OF OPERATOR

Precedence tells in an expression which operations should be performed first depending on priority of operators. Associativity means when two or more operators have same priority then from which side (left or right) to operate (see Table 3.14); for example, in the expression $a*b/c$ '*' and '/' have got same priority whether one performs $a*b$ first or b/c first or in other way 'b' should be considered as part of sub-expression b/c or $a*b$. Associativity of '*' and '/' is from left to right, so the sub-expression $a*b$ will be performed first and then the result of $a*b$ will be divided by c. While executing some expressions first, regardless of the priority of the operators, one can console the expressions within parenthesis. For example:

```
float a=12f, b=3f, c=4f, d;
d = (a-d)/c;
System.out.println(d); //prints 2.25
```

Operators	Associativity	Priority
(), [], .	Right to left	1
+, -, ++, !, &, ~, <code>sizeof</code>	Right to left	2
*, /, %	Left to right	3
+, -	Left to right	4
<<, >>	Left to right	5
<, <=, >, >=	Left to right	6
<code>==</code> , <code>!=</code>	Left to right	7
<code>&</code>	Left to right	8
<code>^</code>	Left to right	9

(Continued)

Operators	Associativity	Priority
	Left to right	10
&&	Left to right	11
	Left to right	12
:?	Right to left	13
=	Right to left	14

Table 3.14 Operator precedence table

In order to evaluate the difference of a and b divided by c, it can be written $(a - b)/c$. As precedence of ‘/’ is more than ‘-’, if it is written as $a - b/c$, then initially b/c will be evaluated, which is not required. So, whenever an expression needs to be evaluated irrespective of the priority of the operator, it is written within parenthesis.

3.12 TYPE CONVERSION AND TYPECASTING

Sometimes, in an expression, it is required to convert the data type of a variable or a constant, for example, from float to int or int to float, etc. In this case, two types of conversions are used in practice.

1. *Implicit type of conversion:* In this type of conversion, the compiler internally converts the type depending on the expression without letting the user to know. The following example can be presented:

```
double num =34;
double d = 45.56f;
```

In this example, value 34 is integer constant and an integer is assigned to double. As double data type has higher width than integer data type, 34 will be converted internally into double and will be assigned to num. Similarly, in the next case, floating point constant is assigned to double data type. Note that a reverse will not be true, i.e., one cannot assign a float to an integer, an integer to byte or a double to float, etc. Java is a strong type language. For example, in C++, assigning a float to an integer is allowed. The float value is internally converted to integer and the fraction part is lost. In Java, this will be an error. In Java, only the lower data type can be assigned to higher data type without casting. This is known as widening or promotion.

Look at the following code snippet as an example of implicit type conversion.

```
int a =10;
float b = 2.5f, c;
c = a+b;
```

In the expression $c = a + b$, type of ‘a’ and ‘b’ is not the same. According to size, **float** is greater than **int** so variable ‘a’ is internally converted into **float** and then addition is performed so that the result obtained will also be in **float**.

2. *Explicit type conversion:* In this type of conversion (also known as typecasting), the data type of the variable or constant is explicitly converted from one to another. For example:

```
float x= 2.5f;
int a=x;
```

If the above code is compiled in Java, it will produce a compilation error as one cannot assign a higher data type to a lower data type without typecasting. Assigning a higher data type to a lower data type can only be allowed when typecasting is done. For example:

```
int x = 23.45f;           //error cannot assign float to
                           int;
```

```

int x = (int)23.45f;           //valid 23 will be assigned to x;
byte b = 260;                 //error 260 is integer
byte b =(byte)260;            //260%256 (range of byte data type)
                             =4 is assigned to b;
float f = 14.4878129878     //error
float f =(float)14.4878129878; or float f= 14.4878129878f;

```

NOTE: Loss of precession will occur as **f** will contain 14.487813, as float allows only six digits of precision whereas double allows 14 digits of precision.

In addition, all expressions involving either byte or short or both are internally converted to integer before the operation is performed. For example:

```

byte b1 = 20, b2 = 5;
byte b3 = b1 * b2;

```

The above line will give error as $b1*b2$ is promoted to integer in the expression that becomes 100 and treated as integer. Assigning an integer to byte data type without typecasting will give compilation error, which has been seen in bitwise operators. The correct way will be as follows:

```
byte b3 = (byte) (b1*b2);
```

3.13 MATHEMATICAL FUNCTIONS

For using mathematical functions in the program, Math class as defined in `java.lang` package can be used. The class defines a number of mathematical functions which can be used in a number of programming situations. The class `Math` contains methods for programming basic numeric functions. All methods are static and return double values.

Some of the methods return value other than double. It also provides a field PI which returns values of $22/7$ or the ratio of the circumference of a circle to its diameter.

The most commonly used functions are listed in Table 3.15.

Function Signature	Purpose
<code>static double abs (double a)</code>	Returns the absolute value of a double value.
<code>static double ceil (double a)</code>	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument.
<code>static double cos (double a)</code>	Returns the trigonometric cosine of an angle.
<code>static double exp (double a)</code>	Returns Euler's number e raised to the power of a double value.
<code>static double floor (double a)</code>	Returns the largest (closest to positive infinity) double value that is less than or equal to the argument.
<code>static double hypot (double x, double y)</code>	Returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
<code>static double log (double a)</code>	Returns the natural logarithm (base e) of a double value.
<code>static double max (double a, double b)</code>	Returns the greater of two double values.
<code>static double min (double a, double b)</code>	Returns the smaller of two double values.
<code>static double pow (double a, double b)</code>	Returns the value of the first argument raised to the power of the second argument.

(Continued)

Function Signature	Purpose
static double random()	Returns a double value with a positive sign greater than or equal to 0.0 and less than 1.0.
static long round (double a)	Returns the closest long to the argument.
static double sin (double a)	Returns the trigonometric sine of an angle.
static double sqrt (double a)	Returns the correctly rounded positive square root of a double value.
static double tan (double a)	Returns the trigonometric tangent of an angle.

Table 3.15 Some mathematical functions

Some of the functions are discussed below.

1. **Math.abs (argument)**

The method returns the absolute value of the argument. The method is overloaded for int, long, float and double data type.

For example:

```
double d = Math.abs (223);           // returns 23
```

2. **Math.cbrt (double d)**

The method finds cube root of argument d of double type.

For example:

```
double d1 = Math.cbrt(27.0);        //prints 3.0
```

3. **Math.ceil (double a)**

The method returns the smallest double value that is greater than or equal to the argument and is equal to a mathematical integer.

For example:

```
double d = Math.ceil (2.3);         //returns 3.0
```

4. **Math.floor (double d)**

This function returns the largest double value that is less than or equal to the argument and is equal to a mathematical integer.

For example:

```
double d = Math.floor (2.3);        //return 2.0
```

5. **int Math.round (double d)**

The method returns the closest int to the argument.

For example:

```
int x = Math.round (2.3);           //returns 2
int y = Math.round (2.6);           // returns 3
```

6. **Math.pow (double a, double b)**

The method returns the value of the first argument raised to the power of the second argument.

For example:

```
double d = Math.pow(2, 3);          //returns 8
```

7. **Math.sqrt (double d)**

The method returns square root of the argument passed.

For example:

```
double d= Math.sqrt (25);           //returns 5.0
```

3.14 SCOPE AND LIFETIME

When a variable is declared within a function or a block (A block can be created by enclosing statements within {}) that variable is accessible only to that block or function. Other blocks/functions cannot access that variable as they are local to that block/function, and where the variable can be accessed is called the scope of the variable. In other words, it is called the portion of program where the variable may be visible or available.

Nesting of scope can be done, i.e., one block can be created inside another block. All the variables of the outer block can be accessed in the inner block but reverse is not true. Here, the scope is discussed within block and method only. The other type of scope is created by the class which will be discussed in the study about classes.

```
/*PROG 3.22 DEMO OF BLCOK AND SCOPE VER 1 */
```

```
class JPS22
{
    public static void main(String args[])
    {
        int p = 20;
        System.out.println("\nIn main");
        System.out.println("p=" + p);
        {
            System.out.println("\nIn block");
            int y = 30;
            System.out.println("y=" + y);
            System.out.println("p=" + p);
        }
    }
}
OUTPUT:
In main
p=20
In block
y=30
p=20
```

Explanation: In the program, the ‘p’ is declared within main. Inside main, a block is created and in it there is an integer variable y initialized by 30. In the block, one can access the variable p of the main but y cannot be accessed outside the block. Thus, the scope of p is the entire main method whereas the scope of y is the block only. If one tries to use y outside the block, he will get a compilation error.

One important point to note regarding variables in the blocks or methods is that whenever control reaches into the method, variables spring to life and as soon as control goes out of method, they expire, i.e., exist no longer. Also, one cannot use the variable before its declaration. All variables must be used only after they are declared and initialized within the methods or blocks.

The lifetime of a variable is the time from its birth to its death, i.e., the time elapsed the moment it sprang to life to the time it lost. In the above case, the scope and lifetime for variables within block and method is the same.

Finally, it should be noted about blocks and scope that if an outer block has a variable by some name than the inner block cannot have a variable by the same name. Look at the program given below.

```
/* PROG 3.23 DEMO OF BLOCK AND SCOPE VER 2 */

class JPS23
{
    public static void main(String args[])
    {
        int x = 20;
        {
            int x = 40;
        }
    }
}

OUTPUT:
C:\JPS\ch3>javac JPS23.java
JPS23.java:8: x is already defined in main(java.lang.String[])
        int x = 40;
               ^
1 error
```

Explanation: The program generates compilation error as x has already been defined within the main. One cannot redefine it in the inner block.

3.15 PONDERABLE POINTS

1. In Java, ‘>>>’ operator is called right shift with zero fill. It is a right shift operator which fills 0 instead of the sign of the number.
2. Java is a strong type language, i.e., one cannot assign a lower data type to a higher without typecasting.
3. By default, a floating point number is considered as double. To make it float, one needs to suffix f/F to the number.
4. A valid combination of constants, variables and operators continues an expression.
5. When several operators appear in one expression, evaluation takes place according to certain predefined rules, which specify the order of evaluation and are called precedence rules.
6. The expression ‘x++’ executes faster than ‘x+1’ as the former requires a single machine instruction, such as INR (increment) to carry out the increment operation whereas, ‘x+1’ requires more instructions to carry out this operation.
7. If an operand in an expression or a statement is converted to a higher rank data type causing upward type conversion, it is known as integral promotion. For example, int data type is converted to float.
8. If an expression or a statement is converted to a lower rank data type, it is known as downward type conversion. It may occur when the left-hand side of an assignment has a lower rank data type compared to the values in the right-hand side. For example, assigning a float value to an int type variable. This is not possible in Java without explicit typecasting.
9. Typecasting is also known as type coercion.
10. For using mathematical functions in Java one can use Math class.
11. “The” are where the variable can be accessed is called the scope of the variable. In other words, we can say portion of the program where the variable may be visible or available.
12. Lifetime of a variable is time from its birth to its death, i.e., the time elapsed the moment it sprang to life to the time it lost.

REVIEW QUESTIONS

1. What are the different types of operators available in Java?
2. What is the difference between '`>>`' and '`>>>`' operator?
3. Explain how Boolean operator works in Java. How it is different from `int` type?
4. Explain the functionality of `instanceof` operator. How it is used?
5. Explain with suitable example why Java is called a strong type language.
6. Explain demotion and promotion in typecasting.
7. Discuss few mathematical functions in Java.
8. Write a program to compute division of 2 without using any arithmetical operator.
9. Find the sum of the digits if the given number is 12345 and the total is $1 + 2 + 3 + 4 + 5 = 15 = 1 + 5 = 6$ in single digit. Do not use any looping structure.
10. Write a program that performs the following: If the user gives input as 1, the output is 2; if the input is 2 then the output becomes 1.
11. Give the precedence and associativity of Java operators.
12. What are Java separators? Give an example.
13. Suppose you declare a float variable `f` and initialize the value as 1.275 `float test = 1.275;` which cannot be compiled. How do you avoid the error?
14. If `a = -1` and `b = -2`, then the statement `result = (a>b)? a: b;` What is the value in the result?
15. What is the result of the following operation:
`System.out.println(3|4);`
16. What will be the result of the expressions:
 - (a) `13&9`
 - (b) `25|9`
 - (c) `-21%5`
 - (d) `2519`
17. What is the octal equivalent of 1234?
 - (a) 01234
 - (b) 0X1234
 - (c) 0x1234
 - (d) 1234
18. What will be the output of the following program when it is executed with command line argument?
`java contest how are you`
`class contest`
`{`
 `public static void main`
 `(String args[])`
 `{`
 `System.out.println`
 `(args[1]);`
 `}`
`}`
19. What will happen if you compile/run this code in your program:
`int j = 089;`
20. What will happen when you compile and run the program:
`public class JPS`
`{`
 `public static void main`
 `(String args)`
 `{`
 `System.out.println`
 `("Welcome in Java");`
 `}`
`}`

Multiple Choice Questions

1. The number of binary arithmetic operators in Java is:
 - (a) 5
 - (b) 4
 - (c) 7
 - (d) 6
2. The operator `%` can be applied only to
 - (a) float values
 - (b) double values
 - (c) (a) and (b)
 - (d) integral values
3. Which of the following is not a valid expression?
 - (a) `+0x3456`
 - (b) `-0345`
 - (c) `23-`
 - (d) `+a`
4. Which of the following executes quickly?
 - (a) `p++`
 - (b) `++p`
 - (c) (a) and (b)
 - (d) `p+1`
5. Identify the valid expression(s).
 - (a) `a = 0`
 - (b) `a = b = 0`
 - (c) `a % = (x % 10)`
 - (d) all of the above
6. Integer division results in
 - (a) rounding of the fractional part of the quotient
 - (b) floating value
 - (c) truncating the fractional part of the quotient
 - (d) syntax error
7. Which of the following is not a valid expression?
 - (a) `++(a+ b)`
 - (b) `y--`
 - (c) `--x --`
 - (d) `+ + p + q`

8. If the value of $a = 10$ and $b = -1$, the value of x after executing the following expression is:
 $x = (a! = 10) \&& (b = 1)$
(a) 0 (c) -1
(b) 1 (d) None

9. Which of the following is a better approach to do the operation $I = I * 16$?
(a) Multiply I by 16 and keep it
(b) Shift left by 4 bits
(c) Add I , 16 times
(d) None of the above

10. The second operator of the operator $\%$ must always be a
(a) negative value (c) zero
(b) non zero (d) positive value

11. Typecasting is also known as
(a) type coercion (c) neither (a) nor (b)
(b) type conversion (d) none of the above

12. In Java, $>>>$ operator is called
(a) left shift
(b) right shift

(c) left shift with zero fill
(d) right shift with zero fill

13. By default, a floating point number is considered as double. To make it float, you need to
(a) suffix f/F to the number
(b) prefix f/F to the number
(c) prefix d/D to the number
(d) suffix d/D to the number

14. int a = 15;
float b = 3.5 f, c;
c = a + b;

The code snippet shows the example of
(a) explicit type conversion
(b) implicit type conversion
(c) both (a) and (b) are correct
(d) none of the above

15. To check which object is of which class, which of the following is used
(a) sizeof (c) comma
(b) instanceof (d) none of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a

2. d

3. c

4. c

5. d

6. c

7.c

8. a

9. b

10. b

11. a

12. d

13. a

14. b

15. b

Working with Decision-making Statement in Java

4

4.1 INTRODUCTION

Decision-making statements are needed to alter the sequence of statements in a program depending on certain circumstances. In the absence of decision-making statements, a program executes in a serial fashion on statement by statement basis, which have been presented in programs given in earlier chapters. This chapter deals with statements that control the flow of execution on the basis of some decision. All the control flow constructs have their conditions which return Boolean value. Depending on truth or falsity, the decision is taken. Decision can be made on the basis of success or failure of some logical condition. They allow one to control the flow of program. The statements inside source files are generally executed from top to bottom, in the order they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping and branching, enabling the program to *conditionally* execute particular blocks of code. This chapter describes the decision-making statements (*if-then*, *if-then-else*, *switch*), the looping statements (*for*, *while*, *do-while*), and the branching statements (*break*, *continue*, *return*) supported by the Java programming language. Java language supports the following decision-making/control statements:

1. The if statement
2. The if-else statement.
3. The if-else-if statement.
4. The switch-case statement.

All these decision-making statements check the given condition and then execute its sub-block if the condition happens to be true. On falsity of condition, the block is skipped. (A block is a set of statements enclosed within opening brace and closing brace {and}). All control statements use a combination of relational and logical operators to form conditions as per the requirement of the programmer.

4.2 THE **if** STATEMENT

The **if** statement is the most basic of all the control flow statements. It tells the program to execute a certain section of code only if a particular test evaluates to **true**. The general syntax of if statement is given as follows:

```
if (condition)
{
    statements;
    statements;
    ....;
    ....;
}
```

The `if` statement is used to execute/skip a block of statements on the basis of truth or falsity of a condition. The condition to be checked is put inside the parenthesis which is preceded by the keyword `if`.

For example, consider the following code:

```
int x = 16;
if(x > 0)
    System.out.println("x is greater than 0");
```

In this code, the condition is `x > 0` which is checked using `if`. As the value of `x` is greater than `0`, the condition `x > 0` is true. If no braces are present after `if`, the first statement after `if` depends on `if`. So it is executed and the output becomes '**x is greater than 0**'.

In case condition is false, the statement after `if` will be skipped and there will be no output. An example is given below.

```
x = 30;
if (x < 15)
    System.out.println("x is less than 15");
    System.out.println("x is greater than or equal to fifteen");
```

Here as the condition is false and there are no braces after `if`, only the first statement after `if` will be skipped and the next statement will be executed. So the output will be '**x is greater than or equal to fifteen**'.

In case no condition is specified in `if` statement, i.e., the expression is given as `if (x)`, it is considered equivalent to the condition `if (x != false)`. The condition `if (!x)` is considered equivalent to the condition `if (x == false)`. In both the cases, `x` must be a Boolean variable.

```
/*PROG 4.1 DEMO OF IF VER 1 */

import java.io.*;
import java.util.*;
class JPS1
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner (System.in);
        System.out.print("\n\nEnter an integer number:=");
        int x = sc.nextInt();
        if (x>=100)
        {
            System.out.println("\n x is greater or equal to 100\n");
            System.out.println("\n You think better\n");
        }
        System.out.println("\n x is less than 100\n");
    }
}
OUTPUT:
First Run)
Enter an integer number:=15
x is less than 100
(Second Run)
Enter an integer number: = 105
x is greater or equal to 100
You think better
x is less than 100
```

Explanation: Here the statements after `if` expression are put in the braces. When the `if` condition is true, all the statements within the braces get executed; in case the condition is false, the statements within the braces are skipped. Regardless of true/falsity of `if` condition, the remaining statements are executed and so is the output. To execute only `if`-dependent statements, one will have to use `if-else` construct or terminate the program after the execution of if block. The second alternative is shown in the next program.

```
/*PROG 4.2 DEMO OF IF VER 3 */

class JPS3
{
    public static void main(String args[])
    {
        boolean x = false;
        if (x)
        {
            System.out.println("\n X is greater or equal
                               to 100");
            System.out.println("\n You think better");
            System.exit(0);
        }
        System.out.println("\n X is less than 100");
    }
}
OUTPUT:
X is less than 100
```

Explanation: In Java, `if(x)` is equivalent to `if(!x == false)` and `x` can only be Boolean value. Similarly, `if(!x)` is equivalent to `if(x == false)`.

4.3 THE `if-else` STATEMENT

The `if-else` statement provides a secondary path of execution when an ‘if’ clause evaluates to false. In all the above programs, the other side of `if` condition was not written, i.e., no action was taken when the condition fails. The `if-else` construct allows one to do this as shown below:

```
if(condition)
{
    statements;
    statements;
    .......
}
else
{
    statements;
    statements;
    .......
}
```

If the condition within `if` command is true, all the statements within the block following `if` are executed else they are skipped and statements within the `else` block get executed. In any case either `if` block or `else` block is executed. For example, consider the following code:

```
if (x>=0)
    System.out.println("x greater than equal to 0");
else
    System.out.println("x less than 0");
```

Check out few programs given below:

```
/*PROG 4.3 PROGRAM TO CHECK NUMBER IS EVEN OR ODD */

import java.io.*;
import java.util.*;
class JPS4
{
    public static void main(String args[])
    {
        int x;
        Scanner sc = new Scanner(System.in);
        PrintStream pr = System.out;
        pr.println("\n Enter an Integer number");
        x = sc.nextInt();
        if (x % 2 == 0)
            pr.println("Number " + x + " is even");
        else
            pr.println("Number " + x + " is odd");
    }
}

OUTPUT:
(First Run)
Enter an Integer number
14
Number 14 is even
(Second Run)
Enter an Integer number
13
Number 13 is odd
```

Explanation: As discussed earlier, `System.out` is an object of class `PrintStream` so it is assigned to a reference `pr` of `PrintStream` type. Now instead of writing `System.out.println`, you can write `pr.println`. If the `if` condition `x%2 == 0` is true, the number is even, else the number is not even. The `else` part executes only when `if` part is false and vice-versa. In the program both `if` and `else` parts contain just one statement to be dependent on them so braces are not needed; however, if you put the braces there will not be any harm.

```

/*PROG 4.4 MAXIMUM OF TWO NUMBERS */

import java.io.*;
import java.util.*;
class JPS5
{
    public static void main(String[] args)
    {
        int x, y;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter first number :=");
        x = sc.nextInt();
        System.out.print("\nEnter second number:=");
        y = sc.nextInt();
        if (x == y)
        {
            System.out.println("\nBoth are equal");
            System.exit(0);
        }
        if (x > y)
            System.out.println("\nMaximum is " + x);
        else
            System.out.println("\nMaximum is " + y);
    }
}

OUTPUT:
(FIRST RUN)
Enter first number      :=40
Enter second number     :=40
Both are equal
(Second Run)
Enter first number      :=40
Enter second number     :=60
Maximum is 60

```

Explanation: Program is self-explanatory.

4.4 NESTING OF if-else STATEMENT

Nesting of if-else means one if-else or simple if as the statement part of another if-else or simple if statement. There may be various syntaxes of nesting if-else. The most commonly used are given in Figure 4.1 as shown below:

1. In the first form of nesting of if-else, if the first if condition is true then the inner if-else block is executed. In case the condition is false, the entire inner if-else block is skipped and no action is taken.
2. The second form is the variation of the first form, where if the outer if condition is false, outer else block gets executed. In the outer else block there is no inner if or if-else part.
3. In the third form, we have inner if-else blocks for execution in both outer if and outer else blocks.

SYNTAX-1	SYNTAX-2
<pre>if(condition) { If(condition) { Statements; Statements; Statements; } else { Statements; Statements; Statements; } }</pre> <p>In the above case there is no else block of the first if command.</p>	<pre>if(condition) { if(condition) { Statements; Statements; Statements; } else { Statements; Statements; Statements; } } else { Statements; Statements; Statements; }</pre>
SYNTAX-3	
<pre>if(condition) { if(condition) { Statements; Statements; Statements; } else { Statements; Statements; Statements; } }</pre>	<pre>else { if(condition) { Statements; Statements; Statements; } else { Statements; Statements; Statements; } }</pre>

Figure 4.1 Different types of nested if-else statement

```
/*PROG 4.5 MAXIMUM OF THREE NUMBERS USING NESTING OF IF-ELSE STATEMENT */
```

```
import java.io.*;
import java.util.*;
class JPS6
{
    public static void main(String args[])
    {
        int a, b, c, d = 0;
        Scanner sc = new Scanner(System.in);
        System.out.println("\nEnter three number");
        a = sc.nextInt();
        b = sc.nextInt();
        c = sc.nextInt();
        if ((a == b) && (a == c))

```

```

        System.out.println("All three are equal\n");
    else
    {
        if (a > b)
        {
            if (a > c)
                d = a;
            else
                d = c;
        }
        else
        {
            if (b > c)
                d = b;
            else
                d = c;
        }
    }
    System.out.println("Maximum is " + d);
}
}

OUTPUT:
Enter three number
23 56 78
Maximum is 78

```

Explanation: If the numbers are equal the first `if` condition is true else `else` block is executed. In the else part, there is one more if-else. If `a > b` is true then inside this if block, using one more if-else it is checked whether `a > c`, if this is so then `a` is greater else `c` is greater.

If the first `if` condition is false in the `else` block then `else` part of the inner if executes which means `b > a`. Inside this inner else one more if-else checks whether `b > c`, if it is so then `b` is greater else `c` is greater.

4.5 else-if LADDER

The general syntax of else-if ladder is shown below:

```

if (condition)
    statements;
else if (condition)
    statements;
else if (condition)
    statements;
....;
....;
....;

else
    statements;

```

If the first `if` condition is satisfied then all its related statements are executed and all other `else-if`'s are skipped. The control reaches to first `else-if` only if the first `if` fails. This is same for second, third and other `else-if`'s depending on what the program required. That is, out of this `else-if` ladder only one `if` condition will be satisfied.

For example, consider the following code:

```
if(marks>=90)
    System.out.println("Grade is A");
else if(marks>=80 && marks<90)
    System.out.println("Grade is B");
else if(marks>=70 && marks<80)
    System.out.println("Grade is C");
else if(marks>=60 && marks<70)
    System.out.println("Grade is D");
else if(marks>=50 && marks<60)
    System.out.println("Grade is E");
else
    System.out.println("Fail");
```

If $\text{marks} \geq 90$, first `if` will be true, 'Grade is A' will be displayed and the remaining `else-ifs` will be skipped. In case first `if` is false, first `else-if` will be checked. If it is true then 'Grade is B' will be displayed and the remaining `else-ifs` will be skipped. The same analogy applies for other `else-ifs`.

A few programs are given below:

```
/*PROG 4.6 FINDING ROOTS OF THE QUADRATIC EQUATION */

import java.io.*;
import java.util.*;
class JPS8
{
    public static void main(String args[])
    {
        double a, b, c, r1, r2, dis;
        Scanner sc = new Scanner(System.in);
        System.out.print("\n\nEnter value of A :=");
        a = sc.nextDouble();
        System.out.print("Enter value of B :=");
        b = sc.nextDouble();
        System.out.print("Enter value of C :=");
        c = sc.nextDouble();
        dis = b * b - 4 * a * c;
        if (dis < 0)
            System.out.print("\nRoots are imaginary\n");
        else if (dis == 0)
        {
            System.out.print("\nRoots are equal\n");
            r1 = -b / (2 * a);
            r2 = b / (2 * a);
            System.out.print("\nRoot1 or r1 := "
                +r1+"\n");
        }
    }
}
```

```

        System.out.print("\nRoot2 or r2 := "
                        +r2+"\n");
    }
else
{
    System.out.println("\n Roots are
                        unequal\n");
    r1 = (-b + Math.sqrt(dis)) / (2.0 * a);
    r2 = (-b + Math.sqrt(dis)) / (2.0 * a);
    System.out.print("\nRoot1 or r1:= "
                        +r1+"\n");
    System.out.print("\nRoot2 or r2:= "
                        +r2+"\n");
}
}

OUTPUT:
Enter value of A :=2
Enter value of B :=6
Enter value of C :=2
Roots are unequal
Root1 or r1:= -0.3819660112501051
Root2 or r2:= -0.3819660112501051

```

Explanation: The **quadratic equation** in mathematics is given as follows:

$$AX^2 + BX + C = 0,$$

where, A, B and C are constants. The solution of the equation comprises two roots as power of X is 2.

$$R1 = (-B + \sqrt{B * B - 4 * A * C}) / 2 * A$$

$$R2 = (-B - \sqrt{B * B - 4 * A * C}) / 2 * A$$

The expression **B * B – 4 * A * C** is known as **discriminant (say dis)**, and on the basis of its value the roots are determined as equal (**dis == 0**) **imaginary (dis < 0)** or **unequal (dis > 0)** as shown in the above program.

In the program, the value of three constants are taken as output using A, B and C and the value of **dis** is calculated.

```
/*PROG 4.7 TO DETERMINE THE STATUS OF ENTERED CHARACTER */
```

```

import java.io.*;
class JPS9
{
    public static void main(String args[])
    {
        char c;
        try
        {
            DataInputStream input;
            input=new DataInputStream(System.in);

```

```

        System.out.println("Enter a character");
        c=(char)(input.read());
        if(c>=65 && c <=90)
            System.out.println("You have entered
                                uppercase letter\n");
        else if(c>=97&&c<=122)
            System.out.println("You have entered
                                lowercase letter\n");
        else if(c>=48 && c<=57)
            System.out.println("You have entered a digit\n");
        else
            System.out.println("You have entered a
                                special symbol\n");
    }
    catch(Exception eobj)
    {
        System.out.println("ERROR!!!!");
    }
}
OUTPUT:

(First Run)
Enter a character
A
You have entered uppercase letter
(Second Run)
Enter a character
a
You have entered lowercase letter
(Third Run)
Enter a character
2
You have entered a digit
(Fourth Run)
Enter a character
$
You have entered a special symbol

```

Explanation: The ASCII/unicode values is from 97 to 122 (inclusive both) for lowercase alphabets and 65 to 90 (inclusive both) for uppercase alphabets. It is checked whether the entered character is within these two ranges using else-if ladder. Similarly ASCII/Unicode values for digits are from 48 to 57 (inclusive both), so character entered is also checked with this range. If all the three conditions fail then the character entered must be a special symbol.

4.6 SWITCH-CASE STATEMENT

Switch-case can be used to replace else-if ladder construct. The `switch` statement allows for any number of possible execution paths. A `switch` works with the `byte`, `short`, `char` and `int` primitive data types. It also works with *enumerated types* and a few special classes that ‘wrap’ certain primitive types: `Character`, `Byte`, `Short` and `Integer` (discussed in simple data objects). Its general syntax is as follows:

```

switch (expression)
{
    case choice1:
        statements;
        break;
    case choice2:
        statements;
        break;
    case choice3:
        statements;
        break;
    .....
    .....
    .....
    case choice N:
        statements;
        break;
    default:
        statements;
}

```

The expression may be any integer or **char** type which yields only **char** or **integer** as result. **choice1**, **choice2** and **choice N** are the possible values which are going to be tested with the expression. In case none of the values from **choice 1 to choice N** matches with the value of expression, the **default** case is executed. Writing **default** is optional. For example:

```

switch(num)
{
    case0:
        System.out.println("You have entered ZERO\n");
        break;
    case1:
        System.out.println("You have entered ONE\n");
        break;
    case2:
        System.out.println("You have entered TWO\n");
        break;
    default:
        System.out.println("Other than 0,1,2\n");
}

```

The condition to be checked is placed inside the **switch** enclosed in parenthesis. Here num is checked against different case expressions. The different values against which condition is checked are put using **case** statement. In the first case, value of num is checked against 0. This is similar to writing **if (num==")**. The colon ‘:’ after **case** is necessary. If the value of num is zero then the first case matches and all the statements under that get executed. Similarly, if the value of num is 1, the second case gets executed and the same applies to the rest of the **case** statements. A **break** statement is needed to ignore the rest of the **case** statements and come out from **switch** block in case a match is found. If none of the **case** matches then

default gets executed. It is to be noted that no break is used after the default as the switch block ends after the default: case. Writing default is optional and not necessarily be put at the end of switch-case. It can be put anywhere within the switch-case; in that case, one will have to write break after the default.

A few programs are given below:

```
/*PROG 4.8 DEMO OF SWITCH-CASE VER 1 */

import java.io.*;
import java.util.*;
class JPS10
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.print("\n Enter 0, 1, or 2 :=");
        int num = sc.nextInt();
        switch (num)
        {
            case 0:
                System.out.println("\n U entered zero\n");
                break;
            case 1:
                System.out.println("\n U entered one\n");
                break;
            case 2:
                System.out.println("\n U enterd two\n");
                break;
            default:
                System.out.println("\n U entered other
                                    than 0,1,or 2\n");
        }
    }
}

OUTPUT:
(First Run)
Enter 0, 1, or 2 :=0
U entered zero
(Second Run)
Enter 0, 1, or 2 :=1
U entered one
(Third run)
Enter 0, 1, or 2 :=2
U enterd two
(Fourth Run)
Enter 0, 1, or 2 :=3
U entered other than 0,1,or 2
```

Explanation: There is no break statement in case 1, so both case 1 and case 2 are assumed to be true and so is the output. In fact, due to the absence of break statement in the second case, rest of the statements are considered part of the second case till a break is not found. Break in case 2 causes control to come out from switch. If break were not in case 2 also then default would have got executed too.

4.7 INTRODUCTION OF LOOPS

Looping is a process in which a set of statements are executed repeatedly for a finite or infinite number of times. Java provides three ways to perform looping by providing three different types of loop. Looping can be synonymously called iteration or repetition. Loops are the most important part of almost all the programming languages such as C, C++, Java, VB, C#, and Delphi.

In many practical examples some repetitive tasks have to be performed like finding average marks of students of a class, finding maximum salary of a group of employees, counting numbers, etc.

A loop is a block of statements which are executed again and again till a specific condition is satisfied. Java provides three loops to perform repetitive actions: `while`, `for` and `do-while`.

To work with any type of loop, three things have to be performed: loop control variable and its initialization; condition for controlling loop; and increment/decrement of control variable.

4.7.1 The `while` Loop

The `while` statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as follows:

```
while (expression/condition)
{
    statement(s);
}
```

The `while` statement evaluates expression, which must return a boolean value. If the expression evaluates to `true`, the `while` statement executes the statement(s) in the `while` block. The `while` statement continues testing the expression and executing its block until the expression evaluates to `false`.

The statement inside {} is called the body of the `while` loop. If no braces are there then only the first statement after `while` is considered as the body of the `while` loop. All the statements within the body are repeated till the condition specified in the parenthesis in `while` is satisfied. As soon as the condition becomes false, the body is skipped and control is transferred to the next statement outside the loop. There should be no semicolon after the `while`. The `while` loop is also called top-testing or entry-controlled loop as the control enters into the body of the loop only when initial condition turns out to be true. If in the beginning of the `while` loop, condition is false then control never enters into the body of the `while` loop.

```
int t=1;           //initial value of loop control variable t
while(t<=10)      //condition in the while
{
    System.out.print(t+" ");
    t++;
}
```

In the `while` loop, the condition stated is `t <= 10` where `t` is called **loop control variable**. Initially, the value of `t` is 1. This value is compared with 10, which is true so control reaches into the `while` loop body, and `System.out.println` statement within the loop executes and prints the value of `t`. Note that the statement within braces are known as the body of the `while` loop. If no braces are present, the first statement after the `while` is considered the body of the `while` loop. Then `t` is incremented by 1, i.e., becomes 2. Control reaches

back to the condition of the while loop, which is true again. This process continues and when the value of t becomes 11, the condition in the while loop becomes false and control comes out of the loop. The output of the above code will be as follows:

```
1 2 3 4 5 6 7 8 9 10
```

Now consider the example of printing number backwards by doing a small change in the above program:

```
int t=10;           //initial value of loop control variable t
while(t>=1)        //condition in the while
System.out.print(t--" ");
```

In this example, the loop control variable t is decremented after printing its value within the expression t itself. The point to be noted is braces were not used; however, there is no problem even if it is used. In the above code if one forgets to decrement the value of t then infinite loop follows.

Similarly, the loop counter may be incremented or decremented by any value depending on the requirement. But while incrementing the counter, test condition must use $<=$ or $<$ operator and in decrementing test condition must use $>=$ or $>$ operator. This is very common programming mistake done by most of the beginners. This mistake is better avoided.

As stated in the beginning of the section, there should be no semicolon after the while loop. But what if there is a semicolon after the while loop? Consider the following code:

```
int t=5;
while(t>=1);
System.out.println(t--" ");
```

The semicolon (;) after the while is called empty statement which is also the body of the while loop since no braces are there. Empty statement means doing nothing, so while loop does nothing and keeps checking the value of t against 1; as t is 5 initially, the condition remains true forever and the infinite loop follows.

The programs given below will help understand the working of the while loop better:

```
/*PROG 4.9 GENERATING TABLE OF A GIVEN NUMBER */

import java.io.*;
import java.util.*;
class JPS12
{
    public static void main(String args[])
    {
        int num, t = 1, value;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter any +ve number:= ");
        num = sc.nextInt();
        System.out.println("\nThe table of "+num+" is
                           given below\n");
        while (t <= 10)
        {
            value = num * t;
            System.out.println(num + " x " + t + " = " +
                               value);
            t++;
        }
    }
}
```

```

        }
    }

OUTPUT:

Enter any +ve number: = 5
The table of 5 is given below

5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50

```

Explanation: Any +ve number is taken as input. The number n is multiplied by 1 through 10 inside the loop and the loop counter; the number and value are displayed by formatting as to produce the desired output.

```

/*PROG 4.10 MAXIMUM OF N ELEMENTS */

import java.io.*;
import java.util.*;
class JPS13
{
    public static void main(String args[])
    {
        int n, max, t=1, m;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter how many numbers");
        n = sc.nextInt();
        System.out.println("Enter the number");
        m = sc.nextInt();
        max = m;
        while (t<=n-1)
        {
            System.out.println("Enter the number");
            m = sc.nextInt();
            if(max<m) ;
            max=m;
            t++;
        }
        System.out.println("Maximum element is " + max);
    }
}

```

OUTPUT:

```
Enter how many numbers:=5
Enter the number:=12
Enter the number:=15
Enter the number:=17
Enter the number:=18
Enter the number:=145
Maximum element is:= 145
```

Explanation: Initially the number of elements taken is **n**. The first number is taken outside the loop and assumed to be maximum; this number is stored in **max**. The remaining numbers are taken inside the loop. On each iteration, the number is compared with the **max**. If the **max** is less than the number taken, the number will be the maximum one. This is checked through **if** statement. In the end when control comes out from **while** loop, **max** is displayed.

```
/*PROG 4.11 FINDING REVERSE OF A NUMBER */

import java.io.*;
import java.util.*;
class JPS14
{
    public static void main(String args[])
    {
        int orig, rev = 0, r;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter any +ve number:= ");
        orig = sc.nextInt();
        while (orig != 0)
        {
            r = orig % 10;
            rev = rev * 10 + r;
            orig = orig / 10;
        }
        System.out.print("\n\nReverse Number:= " + rev);
    }
}
OUTPUT:
Enter any +ve number:= 3456
Reverse Number:= 6543
```

Explanation: The logic to reverse a number is quite simple. It can be understood step by step inside the loop:

Suppose **orig** = > original number entered by programmer.

```
rev => reverse of the orig.
r => remainder
While (orig! =0)
```

```

{
    r = orig%10;
    rev = rev *10 +r;
    orig= orig /10;
}

```

The step-by-step procedure inside the loop is as follows.

Initially set $r = 0$

Suppose $orig = 3456$

Then while ($3456 \neq 0$) (condition given inside while loop is true)

S1	orig = 3456	$r = 3456 \% 10 = 6$	$rev = 0*10 + 6 = 6$	orig = $3456/10 = 345$
S2	orig = 345	$r = 345 \% 10 = 5$	$rev = 6*10 + 5 = 65$	orig = $345/10 = 34$
S3	orig = 34	$r = 34 \% 10 = 4$	$rev = 65*10 + 4 = 654$	orig = $34/10 = 3$
S4	orig = 3	$r = 3 \% 10 = 3$	$rev = 654*10 + 3 = 6543$	orig = $3/10 = 0$
S5	orig = 0			

As $orig = 0$, so condition inside the `while` loop is false and control comes out of loop. The reverse number is variable `rev`, which is printed.

```

/*PROG 4.12 TO CHECK NUMBER IS PRIME OR NOT */

import java.io.*;
import java.util.*;
class JPS15
{
    public static void main(String args[])
    {
        int num, c = 2;
        Scanner sc = new Scanner(System.in);
        System.out.print("\n\nEnter any +ve number:= ");
        num = sc.nextInt();
        while (c <= num / 2)
        {
            if (num % c == 0)
            {
                System.out.println("\nNumber is not prime");
                System.exit(0);
            }
            c++;
        }
        System.out.println("\nNumber is prime");
    }
}

OUTPUT:
(First Run)
Enter any +ve number: = 12
Number is not prime
(Second Run)
Enter any +ve number: = 23
Number is prime

```

Explanation: A number is prime if it is completely divisible by 1 and itself, e.g., 1, 3, 5, 7, 11, 13, 17, 19, 23, etc. To check whether a number is prime or not, start from a counter **c = 2** (every number divides by 1) **and** continue till $c \leq num/2$ since no number is completely divisible by a number which is more than half of that number. For example, 12 is not divisible by 7, 8, 9, 10, 11 which are greater than 6. So it should be checked whether the number divisible by any number $\leq num/2$ is true; simply print ‘Number is not prime’ and exit from the program, using `System.exit(0)`. If `num%c == 0` is never true during the loop then at the end when loop exits maturely print ‘Number is prime’.

```
/*PROG 4.13 CHECK NUMBER FOR PALINDROME */

import java.io.*;
import java.util.*;
class JPS16
{
    public static void main(String args[])
    {
        int orig, rev = 0, r, save;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter any +ve number:=");
        orig = sc.nextInt();
        save = orig;
        while (orig != 0)
        {
            r = orig % 10;
            rev = rev * 10 + r;
            orig = orig / 10;
        }
        if (rev == save)
            System.out.println("Number is palindrome");
        else
            System.out.println("Number is not palindrome");
    }
}

OUTPUT:
First Run)
Enter any +ve number: =4545
Number is not palindrome

Second Run)
Enter any +ve number: =454
Number is palindrome
```

Explanation: A number is called palindrome if on reversing it is equal to the original number, e.g., 121, 454 and 3443, etc. To check whether an entered number is palindrome or not, simply reverse a number; the original number becomes zero when controls come out from the loop. So, the original number is saved in a variable before processing; in the above program, it is in the `save` variable.

```
/*PROG 4.14 CHECK NUMBER FOR ARMSTRONG */

import java.io.*;
import java.util.*;
class JPS17
{
    public static void main(String args[])
    {
        int num, r, save, newnum = 0, count = 0;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter any +ve number:= ");
        num = sc.nextInt();
        save = num;
        while (num != 0)
        {
            num = num / 10;
            count++;
        }
        num = save;
        while (num != 0)
        {
            r = num % 10;
            newnum = newnum + (int) Math.pow(r, count);
            num = num / 10;
        }
        if (newnum == save)
            System.out.println("\nNumber " + save + " is
                               Armstrong");
        else
            System.out.println("\nNumber " + save + " is
                               not Armstrong");
    }
}

OUTPUT:
(First Run)
Enter any +ve number:= 153
Number 153 is Armstrong
(Second Run)

Enter any +ve number: = 2345
Number 2345 is not Armstrong
```

Explanation: A number is called Armstrong if sum of count number of power of each digit is equal to the original number. For example, to check whether **153** is **Armstrong number** or not, see that number of digits are **3** then

$$1^3 + 5^3 + 3^3 = > 1 + 125 + 27 = > 153$$

This is equal to the original number so number 153 is Armstrong.

Consider a four digit number **1653**.

$$1^4 + 6^4 + 5^4 + 3^4 = 1 + 1296 + 81 + 256 = 1653$$

So program proceeds as follows:

First while loop: First find out number of digits; before doing this, **save** the number in the **save** variable. Now number of digits is stored in the **count** variable. At this stage, num is **0** so copy the value from **save** to **num**. **POW** is a library function whose prototype is given in header file **math.h**. It returns the number to the power where first argument is number and second is power, e.g., **pow(2, 3)** gives $2^3 = 8$, **pow(3, 2) = $3^2 = 9$** .

Second while loop: Steps of second while loop are as follows:

Initially	num = 153	newnum = 0	count = 3
S1	r = 153%10 = 3	newnum = 0 + pow (3, 3) = 0 + 33 = 27	num = 153/10 = 15
S2	r = 15 %10 = 5	newnum = 27 + pow (5, 3) = 27 + 53 = 152	num = 15/10 = 1
S3	r = 1%10 = 1	newnum = 152 + pow (1, 3) = 152 + 1 = 153	num = 1/10 = 0

As **newnum** contains 153, it is compared with the original number stored in **save**. The output becomes **153** as Armstrong number.

4.7.2 The for Loop

The **for** statement provides a compact way to iterate over a range of values. Programmers often refer to it as the '**for loop**' because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the **for** statement can be expressed as follows:

```
for (initialization; termination; increment)
{
    Statements;
    Statements;
    Statements;
}
```

When using this version of the **for** statement, keep in mind the following:

1. The *initialization* expression initializes the loop; it is executed once, as the loop begins.
2. When the *termination* expression evaluates to **false**, the loop terminates.
3. The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to **increment** or **decrement** a value.

For example: Consider the following code:

```
for (t=1; t<=10; t++)
    System.out.print(" " +t);
```

The first part of **for** loop **t = 1** is the initialization, i.e., providing an initial value of 1 to loop counter **t**. Next statement **t <= 10** is the testing condition for which this loop runs. If the condition is **true**, the last part is incremented with loop counter **t**. The **for** loop works by first taking **t = 1** and then checking the condition whether **t <= 10**; if so, it executes the next statement following **System.out.println**. Recall as there are no braces after the **for** loop, only the first statement is considered the body of the **for**

loop. It then goes on to increment the value of t, checks the test condition and again executes `System.out.println`. This continues till the condition `t <= 10` is true. As soon as the condition becomes false (in this case t becomes 11), control comes out from the loop and the program terminates. The output of the loop will be as follows:

```
1 2 3 4 5 6 7 8 9 10
```

The initialization statement executes only once, whereas condition check and increment/decrement part of the for loop execute till the condition remains true.

Note: There must be two semicolons in the for loop.

Note the difference between while loop and for loop. In the while loop, only condition is specified; initialization of the loop control variable is done before the while loop and increment/decrement is done within the body of the loop. But in the for loop, all three are used simultaneously within the loop itself.

Some more examples of for loop are given below:

- (a)

```
for(int t=10;t>=1;t--)
        System.out.println(" "+t);
Output:- 10 9 8 7 6 5 4 3 2 1
```
- (b)

```
for(t=30;t>=1;t=t-3)
        System.out.print(" "+t);
Output:-30 27 24 21 18 15 12 9 6 3
```
- (c)

```
for(t=100;t>=10;t=t-10)
        System.out.print(" "+t);
Output: - 100 90 80 70 60 50 40 30 20 10
```

4.8 DIFFERENT SYNTAXES OF for LOOP

There are different syntaxes of for loop. The first syntax has been presented as above. Other syntaxes are as follows:

1. `initialization;`
`for (; condition; increment/decrement)`
`body of the loop;`

In this syntax, though we have left the initialization part and have put this before the for loop, semicolon (;) is must in the for loop. For example, see the code given below:

```
int t=10;
for ( ;t<=100;t+=10)
System.out.print (" "+t);
```

2. `for(initialization; condition ;)`
`{`
 `Statements;`
 `Increment/decrement;`
`}`

In this syntax, increment/decrement part has been written within braces. Note that braces are must because there are two statements required to be made as body of the for loop. For example, see the code given below:

```

for(t=1;t<=10;)
{
System.out.print(" "+t);
t++;
}
3. initialization;
for(;condition;)
{
Statements;
Increment/decrement;
}

```

In this syntax, though we have left both initialization and increment part, semicolon is necessary on both sides.

```

t=1;
for ( ; t<=10 ;)
System.out.println(" "+t);

```

4.9 PROGRAMMING EXAMPLES

```

/*PROG 4.15 SUM OF THE SERIES 1-2+3-4+.....*/
import java.io.*;
import java.util.*;
class JPS18
{
    public static void main(String args[])
    {
        int n, sum = 0, t, k = 1;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of terms\n");
        n = sc.nextInt();
        for (t = 1; t <= n; t++)
        {
            if (t % 2 != 0)
                System.out.print(t + "-");
            else
                System.out.print(t + "+");
            sum = sum + (t * k);
            k = k * (-1);
        }
        System.out.println("\nSum of series is " + sum);
    }
}

OUTPUT:
Enter the number of terms
7
1-2+3-4+5-6+7-
Sum of series is 4

```

Explanation: In the series to be generated, odd numbers are +ve and even numbers are -ve. Take one variable **k = 1**. **k** is multiplied by **-1** inside the loop in each iteration.

Initially **sum = 0 + (1 * 1)** gives **sum = 1** then **k** becomes **k = 1 * -1 = -1**, which is used in the second iteration of the loop. In the second iteration, **sum** becomes **sum = 1 + (2 * -1) => sum = 1 - 2 = -1** and value of **k** changes to **1** again (**-1 * -1 = 1**) and this continues for **n** times.

```
/*PROG 4.16 TO PRINT AND FIND SUM OF SERIES 1^2+2^2+3^2+4^2+.....(^ STANDS FOR POWER) */
```

```
import java.io.*;
import java.util.*;
class JPS19
{
    public static void main(String args[])
    {
        double sum = 0;
        int n, t;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of terms\n");
        n = sc.nextInt();
        for (t = 1; t <= n; t++)
        {
            System.out.print(t + " ^2+");
            sum = sum + Math.pow(t, 2);
        }
        System.out.println("\n Sum of series is " + sum);
    }
}
```

OUTPUT:

```
Enter the number of terms
7
1^2+2^2+3^2+4^2+5^2+6^2+7^2+
Sum of series is 140.0
```

Explanation: **pow** is standard library function which returns power of first argument to second number, i.e., **pow(2, 3)** returns **2** to the power **3**, i.e., **8**. A number of terms are taken in **n** and loop is run from **1** to **n**. On each iteration, **2** was found to the power **t** and added to sum. The pattern is displayed within the **for** loop and sum of series is displayed outside the loop.

```
/*PROG 4.17 TO CHECK A NUMBER IS PERFECT OR NOT */
```

```
import java.io.*;
import java.util.*;
class JPS20
{
    public static void main(String args[])
    {
        int num, t, sum = 1;
```

```

Scanner sc = new Scanner(System.in);
System.out.println("Enter an integer");
num = sc.nextInt();
for (t = 2; t <= num / 2; t++)
{
    if (num % t == 0)
        sum = sum + t;
}
if (sum == num)
    System.out.println("The number is perfect");
else
    System.out.println("The number is not perfect");
}

OUTPUT:
Enter an integer
15
The number is not perfect

```

Explanation: A number is called perfect if sum of its factor is equal to the number itself, e.g., 6; its factors are 1, 2 and 3 and their sum is 6, which is equal to the number itself, so it is a perfect number. Similarly 28 is a perfect number. In the loop, the sum is initialized to 1 and the loop is run for num/2 as for any number, e.g., t which is more than num/2, num%t will not be zero (excluding num itself).

4.9.1 Nesting of for Loop

Nesting of for loop is used most frequently in many programming situations and one of the most important usage is in displaying various patterns which can be seen in the following programs. The general syntax is given below:

```

for(initialization;condition;increment/decrement)
{
    for(initialization;condition;increment/decrement)
    {
        statements;
    }
    statements;
}

```

For each iteration of first for loop (outer for loop), inner for loop runs as it is part of the body of outer for loop. The inner for loop has its own set of statements which executes till the condition for inner for loop is true.

For example, consider the following code:

```

for(a=1;a<=3;a++)
{
    for(b=1;b<=4;b++)
        System.out.print(a*b+"\t");
    System.out.println();
}

```

The body of the first `for` loop (outer) contains three statements enclosed within braces. The inner `for` loop's body has got only one statement as there are no braces after the inner `for` loop. Initially `a` is 1 and condition `a <= 3` in the outer loop is true. The first statement inside outer `for` loop is inner `for` loop which initializes `b = 1`. Now this `for` loop runs from 1 to 4 for the value of `a = 1`. When this loop terminates on reaching a value of `b = 5`, control is transferred to third statement `System.out.println();` which leaves a line on the output screen. Now control is transferred to outer loop which increments the value of `a` by 1, which becomes 2. The process repeats with value of `b` from 1 to 4, for value of `a = 2`, till `a <= 3` remains true.

We present few programs on nesting of `for` loop.

```
/*PROG 4.18 PRINT THE FOLLOWING PATTERN, INPUT IS NUMBER
OF LINES
1
2 3
4 5 6
7 8 9 10
*/
import java.io.*;
import java.util.*;
class JPS21
{
    public static void main(String args[])
    {
        int line, row, col, value = 0;
        Scanner sc = new Scanner(System.in);
        System.out.print("\n\nEnter the number of
lines:= ");
        line = sc.nextInt();
        System.out.println("\nThe patter is\n");
        for (row = 1; row <= line; row++)
        {
            for (col = 1; col <= row; col++)
            {
                value++;
                System.out.print(" " + value);
            }
            System.out.println();
        }
    }
}

OUTPUT:
Enter the number of lines: = 4
The patter is
1
2 3
4 5 6
7 8 9 10
```

Explanation: Two `for` loops have been used in the program. One is to control the number of rows and second is to control the number of columns. Initially assume `line = 5` so outer `for` loop runs five times. In the first run, `row = 1` and inner loop runs only once. The `value` of value variable is incremented and printed. Control moves to the new line. For `row = 2`, inner loop runs twice and second row of output is printed. This continues till `row <= line`.

```

/*PROG 4.19 TO DISPLAY THE PATTERN AS
A
B B
C C C
D D D D
E E E E E
.....
....*/

```

```

import java.io.*;
import java.util.*;
class JPS22
{
    public static void main(String args[])
    {
        int line, row, col;
        char ch = 'A';
        Scanner sc = new Scanner(System.in);
        System.out.print("\n\nEnter number of lines:=");
        line = sc.nextInt();
        System.out.println("\n\nThe pattern is\n");
        for (row = 1; row <= line; row++)
        {
            for (col = 1; col <= row; col++)
                System.out.print(" " + ch);
            System.out.println();
            ch++;
        }
    }
}

OUTPUT:
Enter number of lines:=7
The pattern is

A
B B
C C C
D D D D
E E E E E
F F F F F F
G G G G G G G

```

Explanation: For the first run of outer **for** loop, **ch** = 'A'. When first iteration of inner **for** loop finishes, value of **ch** is incremented by 1 and becomes B. As one A on first row, two B on second row and so on have to be printed, **ch** at the end of every iteration of outer **for** loop is incremented.

```
/*PROG 4.20 TO DISPLAY THE PATTERN AS
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
.....
..... */

import java.io.*;
import java.util.*;
class JPS23
{
    public static void main(String args[])
    {
        int line, row, col, temp;
        Scanner sc = new Scanner(System.in);
        System.out.print("\n\nEnter the number of lines:=");
        line = sc.nextInt();
        System.out.println("The pattern is \n");
        for (row = 1; row <= line; row++)
        {
            for (col = 1; col <= row; col++)
                if (row % 2 == 0)
                {
                    if (col % 2 == 0)
                        System.out.print(" 1");
                    else
                        System.out.print(" 0");
                }
                else
                    System.out.print(" " + col % 2);
            System.out.println();
        }
    }
}

OUTPUT:
Enter the number of lines: =7
The pattern is
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
0 1 0 1 0 1
1 0 1 0 1 0 1
```

Explanation: Observe the pattern carefully. There are different patterns on both even and odd number of lines. On even number of lines, i.e., for row = 2, 4, 6..., if the complement of col%2 is printed, the desired pattern is obtained. On row = 1, 3, 5..., col%2 is simply checked. If it is equal to 0, 1 is printed else 0 is printed.

```
/*PROG 4.21 TO PRINT THE FOLLOWING PATTERN, INPUT IS NUMBER OF LINES
      1
      2 3 2
      3 4 5 4 3
      4 5 6 7 6 5 4
      5 6 7 8 9 8 7 6 5
*/
```

```
import java.io.*;
import java.util.*;
class JPS24
{
    public static void main(String args[])
    {
        int line, row, col, val, b;
        Scanner sc = new Scanner(System.in);
        System.out.print("\n\nEnter the number of lines");
        line = sc.nextInt();
        System.out.println("\nThe patter is\n");
        /*Logic to print first half of pattern demarcted
         through bold figure*/
        for (row = 1; row <= line; row++)
        {
            val = row;
            for (b = 1; b <= line - row; b++)
                System.out.print(" ");
            for (col = 1; col < 2 * row + 1; col += 2)
                System.out.print(" " + val++);
        /*Logic to print second half of pattern demarcated
         through bold figures*/
            val = val - 2;
            for (b = 1; b <= row - 1; b++)
                System.out.print(" " + val--);
            System.out.println();
        }
    }
}
```

OUTPUT:

```
Enter the number of lines 5
The patter is
```

```
      1
      2 3 2
      3 4 5 4 3
      4 5 6 7 6 5 4
      5 6 7 8 9 8 7 6 5
```

Explanation: It is assumed the pattern itself consists of two sub-patterns. The first pattern assumed is up to the numbers shown in the bold. Rest of the numbers is part of the second pattern. For each iteration of outer loop, row number is stored into **val** variable and printed till **col < 2*row + 1** remains true. But before this, spaces are left so that numbers appear in the centres and space I is obtained on the left before printing each number. This condition has been chosen as for **row = 1, 2, 3, 4, 2*row + 1** is obtained as **3, 5, 7, 9** which are shown in bold letters in the pyramid.

For second half of pattern, note that in the second line, just one number (**2**) has to be printed; in the third line, two numbers (**4, 3**) and so on which is controlled by **row - 1**. To print the number, **2** is subtracted from **val** and each iteration is decremented.

4.9.2 The do-while Statement

The last loop is do-while loop. Its syntax is as follows:

```
do
{
    statements;
    statements;
    statements;
    .......;
    .......;
}while(condition);
```

It is similar to while loop with the difference that condition is checked at the end, instead of being checked at the beginning. Just because of this the do-while loop is called **bottom-testing loop** or exit-controlled loop. The loop is called so, as condition is checked at the bottom of the loop and exit of the loop is false; the loop executes at least once.

Consider the code given below:

```
int t=1;
do
{
    System.out.print(" "+t);
    t++;
}while(t<=10);
```

The do-while loop is similar to the while loop with the difference that the condition is checked at the end. The loop starts with a do following the block, and at the end of block the condition is written using while. The while is part of the do-while loop and it must be terminated by semicolon; initially the value of t is 1 which is printed through System.out.print, then t becomes 2. At the end of block, as the condition is true, control transfer back to do block and the process continues.

Some important points regarding do-while loop are as follows:

1. It is also called bottom testing loop or exit controlled loop as condition is checked at the bottom of the loop.
2. Regardless of the condition given at the end of block the loop runs at least once.
3. The do-while loop is mostly used for writing menu-driven programs.

Now consider the code

```
int t=0;
do
```

```
{
    System.out.print(" "+t);
    t++;
}while(t<0);
```

The condition at the end in while is false at the very first iteration of loop but loop runs at least once as the condition is checked at the end.

```
/*PROG 4.22 SUM OF DIGITS UP TO SINGLE DIGIT USING DO-WHILE */
```

```
import java.io.*;
import java.util.*;
class JPS25
{
    public static void main(String[] args)
    {
        int num;
        int sum=0,save,r;
        Scanner sc=new Scanner(System.in);
        System.out.print("\nEnter an integer:=");
        num=sc.nextInt();
        do
        {
            sum=0;
            while(num!=0)
            {
                r=num%10;
                sum=sum+r;
                num=num/10;
            }
            if(sum>9)
                System.out.println(sum);
        }while(num>9);
        System.out.println("\n\nSum of digits up to single
                           digit"+ " is:= "+sum);
    }
}
```

OUTPUT:

```
Enter an integer:=234
Sum of digits up to single digit is:= 9
```

Explanation: For example, if the number is 4275 then sum of digits is $4 + 2 + 7 + 5 = 18$. As 18 is greater than 9, the process is repeated and get the result becomes $1 + 8$, i.e., 9. As this is the digit required, so the process is stopped. In the program for finding sum of digits while loop has been used, but for sum of digits up to single digit do-while loop has been used. When sum > 9, sum is assigned to num and for this num, sum of digits is determined using while loop.

```
/*PROG 4.23 MINI AREA CALCULATOR USING SWITCH-CASE AND DO-WHILE */
```

```
import java.io.*;
import java.util.*;
class JPS26
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String []args)
    {
        Scanner sc=new Scanner(System.in);
        float h,b,s,w,l,r,area;
        int choice;
        final float PI=3.14f;
        do
        {
            show("Welcome to Area Zone\n");
            show("1. Area of Triangle\n");
            show("2. Area of Circle\n");
            show("3. Area of Rectangle\n");
            show("4. Area of Square\n");
            show("5. Exit\n");
            show("Enter your choice(1-5) :=");
            choice =sc.nextInt();
            switch(choice)
            {
                case 1:
                    show("Enter the base and height of
                          triangle\n");
                    b=sc.nextFloat();h=sc.nextFloat();
                    area=0.5f*b*h;
                    show("Area of triangle is "+area);
                    break;
                case 2:
                    show("Enter the radius of circle\n");
                    r=sc.nextFloat();
                    area=PI*r*r;
                    show("Area of circle is "+area);
                    break;
                case 3:
                    show("Enter the length and width of
                          rectangle\n");
                    l=sc.nextFloat();w=sc.nextFloat();
                    area=l*w;
                    show("Area of rectangle is "+area);
                    break;
            }
        }while(choice!=5);
    }
}
```

```

        case 4:
            show("Enter the side of square\n");
            s=sc.nextFloat();
            area=s*s;
            show("Area of square is "+area);
            break;
        case 5:
            show("Bye Bye.....");
            System.exit(0);
        default:
            show("Better you know number \n");
    }
}while(choice>=1&&choice<=5);
}
}

```

OUTPUT:

```

Welcome to Area Zone
1. Area of Triangle
2. Area of Circle
3. Area of Rectangle
4. Area of Square
5. Exit

Enter your choice (1-5):=
1
Enter the base and height of triangle
2 4
Area of triangle is 4.0

```

Explanation: In the program, the whole switch-case is put into do-while loop. In each case, areas of triangle, circle, rectangle and square are found out. After fulfilling one choice for the user, the menu again appears because of do-while loop. On entering 5 in the choice, the program terminates.

4.10 break AND continue STATEMENT

4.10.1 The break Statement

The break statement is used to come out early from a loop without waiting for the condition to become false. One such usage of break in the switch-case statements has been observed. When the break statement is encountered in the while loop or any of the loop that would be seen later, the control immediately transfers to first statement out of the loop, i.e., loop is exited prematurely. If there is nesting of loops, the break will exit only from the current loop containing it.

As an example of break statement, consider the following code:

```

int x=1;
while(x<=5)
{
if(x==3)

```

```

break;
System.out.println("Inside the loop "+x);
x++;
}
System.out.println("Outside the loop "+x);

```

In the above code when **x** is **3**, **if** condition becomes true; the body of the **if** statement is single **break** statement, so all the statements in the loop following the **break** are skipped and control is transferred to the first statement after the loop, which is **System.out.println** and prints the statement **Outside the loop x = 3**.

```
/*PROG 4.24 CHECKING WHETHER A NUMBER IS PRIME OR NOT USING  
BREAK AND WHILE */
```

```

import java.io.*;
import java.util.*;
class JPS28
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int num, c = 2;
        boolean flag = false;
        System.out.print("\n\nEnter the number:= ");
        num = sc.nextInt();
        while (c <= num)
        {
            if (num % c == 0)
            {
                break;
            }
            c++;
        }
        if (!flag)
            System.out.println("\nNumber is prime");
        else
            System.out.println("\nNumber is not prime");
    }
}
```

OUTPUT:

```

Enter the number:= 23
Number is prime

```

Explanation: One version of program which checks number for prime has been given earlier. This is the second version of the same program but written using **break**. Here it is checked if the number is divisible by any number $\leq \text{num}/2$, it cannot be prime; we set **flag = true** and come out from the loop. The flag was initialized to false in the beginning so if **num%c == 0** is **true**, control set **flag = true**, which means number is not prime. If **flag** remains **false**, it means control never transferred to **if** block, i.e., number is prime. So outside the loop, this value of flag is checked and printed accordingly.

4.10.2 The continue Statement

The `continue` statement causes the remainder of the statements following `continue` to be skipped and continue with the next iteration of loop. So we can use `continue` statement to bypass certain number of statements in the loop on the basis of some condition if given generally.

The syntax of `continue` statement is simply the following:

```
continue;
int t=0;
while(t<=10)
{
    t++;
    if(t%2!=0)
        continue;
    System.out.println(" "+t);
}
```

```
/*PROG 4.25 SQUARE ROOT OF NUMBER USING CONTINUE AND WHILE */
```

```
import java.*;
import java.util.*;
class JPS29
{
    public static void main(String args[])
    {
        int n, i = 1;
        double num;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter how many numbers:= ");
        n = sc.nextInt();
        while (i <= n)
        {
            System.out.print("\nEnter the double number:= ");
            num = sc.nextDouble();
            if (num < 0)
            {
                System.out.println("Sqrt of -ve number
                                is not possible");
                i++;
                continue;
            }
            System.out.println("Square root:= "
                            +(Math.sqrt(num)));
            i++;
        }
    }
}

OUTPUT:
Enter how many numbers: = 4
```

```

Enter the double number: = 15
Square root: = 3.872983346207417
Enter the double number: = -23
Sqrt of -ve number is not possible
Enter the double number: = 55
Square root: = 7.416198487095663

```

Explanation: This program is simple. The user was initially asked about the limit, i.e., count of numbers of whom square root is to be find out, that is stored into the variable *n*. The loop was then run for *n* times. In each repetition, the user was asked a double number whose square root is to be found out. For finding square root of the number, built-in function `Math.sqrt()` was made use of. In case number is negative, the appropriate message is displayed and continued, i.e., the user is prompted for the next number.

4.10.3 Labelled break and continue Statement

The disadvantage of break and continue is that they are applicable for the current loop only. That is, `continue` causes next iteration of the current loop in which it is present. Similarly, `break` causes exit from the current loop. In case of nesting of loops, when we want to come out from the outermost loop or `continue` from some other loop instead of current loop, simple “`break` and `continue`” does not help.

To solve this problem, Java provides the labelled `break` and labelled `continue` statements that allow coming out from deeply nested loop and `continue` with the outer loop, respectively. The syntax is as follows:

```
break label1;
```

and

```
continue label1
```

where `label1` is the label where the control is to be transferred. The label may be any valid identifier name. As labelled `break` and `continue` are used with the loops, a label must appear prior to `break` and `continue` statements. One example each of `break` and `continue` are given below.

```

/*PROG 4.26 DEMO OF LABELLED CONTINUE */

class JPS30
{
    public static void main(String args[])
    {
        int i, j;
        lab1:
        for (i = 1; i <= 5; i++)
        {
            for (j = 1; j <= 4; j++)
            {
                if (i % 2 == 0) continue lab1;
                System.out.println("i="+i+"\tj="+j);
            }
        }
    }
}

```

OUTPUT:

```
i= 1    j= 1
i= 1    j= 2
i= 1    j= 3
i= 1    j= 4
i= 3    j= 1
i= 3    j= 2
i= 3    j= 3
i= 3    j= 4
i= 5    j= 1
i= 5    j= 2
i= 5    j= 3
i= 5    j= 4
```

Explanation: In the for loop, if `(i%2 == 0)`, the expression `continue lab1` is used to continue from `lab1` from the outer loop instead of continuing from the current loop if label were not present. Each even value for the control variable ‘`i`’ for outer loop is skipped using **continue lab1;** expression.

```
/*PROG 4.27 DEMO OF LABELLED BREAK */

class JPS31
{
    public static void main(String args[])
    {
        int i, j;
        lab1:
        for (i = 1; i <= 5; i++)
        {
            for (j = 1; j <= 4; j++)
            {
                if (i % 2 == 0)
                    break lab1;
                System.out.println("i="+i+"\tj="+j);
            }
        }
    }

    OUTPUT:

    i=1    j=1
    i=1    j=2
    i=1    j=3
    i=1    j=4
```

Explanation: In the inner for loop, if `(i%2 == 0)`, the expression `break lab1` exits from the outer loop and not from the inner loop in which it is placed. This is because label `lab1` is placed on the same line from where outer for loop starts.

4.11 PONDERABALE POINTS

- Java supports all decision making and control statements like if, if-else, switch-case, for, while and do-while.
- The flow of execution may be transferred from one part of a program to another based on the output of the conditional test carried out. It is known as conditional execution.
- if-else, if-else-if and switch-case are known as selective control structures.
- A null statement is represented by a semicolon.
- Null statements are useful to create time delay and to end a label statement where no useful operation is to be performed.
- A do-while loop is called bottom testing loop as condition is checked at the end. Even for the false condition it runs at least once.
- The for loop can be used to act as infinite loop; the code is as follows:
- In Java if (x) is equal to if ($x \neq \text{false}$), where x must be a Boolean value.
- In Java if ($\neg x$) is equal to if ($x = \text{false}$), where x must be a Boolean value.
- Java supports labelled break and continue, which can be used to come out from a deeply nested loop (break) and to continue from an outer loop (continue).

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

REVIEW QUESTIONS

- What are the different streams available in system package?
- What is the difference between print and println statement?
- Classify the control flow statement.
- Draw a flowchart and write a program to pick the largest of three given numbers.
- One foot = 12.0 inches, one inch = 2.54 centimeters; write a program to compute inches and feet of 76.2 cms.
- The following table shows the employees code and the percentage of bonus for the value of basic pay.

Employee code	Bonus
100	5
200	1
300	2
400	25

- In what way does the switch statement differ from if-else statement?
- Write a program to get week day number (1...7) from the user and then

If day = 1 print “Have a nice day”

If day = 2, 3, 4, 5, 6 print “Welcome become to working day”

If day = 7 print “Have a nice week end day”

Use the if-else, if-else-if ladder and switch case structure.

- Write a program; get the input marks from the user through keyboard by checking the following conditions.

If marks < 40 — fail

If marks > 50 — good

If marks > 75 — very good

If marks > 90 — excellent

- A frog starts climbing 30 ft well. Each hour frog climbs 3 ft and slips back 2 ft. How many hours does it take to reach top and get out?

- Differentiate entry control loop and exit control statements.

- What is wrong with this code?

```
switch(character)
{
    case 'a':
        i = 10;
        break;
    case 'b':
        j = 11;
        break;
```

```

    case 'c':
        k = 12;
        break;
        L = 14;
    }
}

```

13. Write a program to solve linear and quadratic equations.
 14. How many times does the `println()` statement execute?
- ```

for(int l =1; l<l++)
System.out.println("My test
 loop");
i = 1
i<10
i++

```
15. What is an empty statement?
  16. Write a program to compute power of 2 using for loop.
  17. What is automatic type conversion? How are widening and narrowing achieved?
  18. Write a program to display the following output using for loop:

(a) 1

```

1 2
1 2 3
1 2 3 4
1 2 3 4 5

```

(b) 1

```

2 2
3 3 3
4 4 4 4
5 5 5 5 5

```

(c) \* \* \* \* \*

```

* * *
* *
*
*
```

19. What will be the output for the given code?

```

for(int l =-4; l<=4; l = l+2)
{
 System.out.println(2/l);
}

```

20. What will be the output for the given code?

```

int x = 10;
switch(x)
{
}

```

```

default:
System.out.println("This is
 first");
case 9:
System.out.println(" x = 9");
break;
case 11:
System.out.println(" x =
 11");
break;
case 12:
System.out.println(" x =
 12");
}

```

21. What will be printed out the following code is compiled and run?

```

int i = 1;
switch(i)
{
 case 0:
 System.out.println("OOPS");
 break;
 case 1:
 System.out.println("C");
 case 2:
 System.out.println("C++");
 default:
 System.out.println("JAVA");
}

```

22. What will be the output of the following code?

```

class JPS
{
 public static void main(String
 args[])
 {
 int i, j;
 outer:
 for (i = 0; i < 3; i++)
 inner: for (j = 0; j < 3;
 j++)
 {
 if (j == 2)
 continue outer;
 System.out.println("i" +
 i + "j" + j);
 }
 }
}

```

## Multiple Choice Questions

1. Identify the wrong statement:
  - (a) `if(a > b);`
  - (c) `if(a > b) { ; }`
  - (b) `if a > b;`
  - (d) (b) and (c)
2. Each case statement in `switch ()` is separated by
  - (a) `break`
  - (c) `exit()`
  - (b) `continue`
  - (d) `goto`

3. Which conditional expression always returns false value?  
 (a) if (a == 0)      (c) if( a = 10)  
 (b) if (a = 0)      (d) if (10 == a)
4. Which conditional expression always returns true value?  
 (a) if (a == 1)      (c) if (a = 0)  
 (b) if(a = 1)      (d) if (1== a)
5. If default statement is omitted and there is no match with case labels,  
 (a) no statements within switch-case block will be executed  
 (b) syntax error is produced  
 (c) all the statements in switch-case construct will be executed  
 (d) the last case statement only will be executed
6. Identify the unconditional control structure from the following options  
 (a) do-while      (c) goto  
 (b) switch-case    (d) if
7. The minimum number of times while loop is executed is  
 (a) 0                (c) 2  
 (b) 1                (d) Cannot be predicted
8. The minimum number of times for loop is executed is  
 (a) 0                (c) 2  
 (b) 1                (d) Cannot be predicted
9. The minimum number of times do-while loop is executed is  
 (a) 0                (c) 2  
 (b) 1                (d) Cannot be predicted
10. Infinite loop is  
 (a) useful for time delay  
 (b) useless  
 (c) used to terminate execution  
 (d) not possible
11. The continue statement is used to  
 (a) continue the next iteration of a loop construct  
 (b) exit the block where it exists and continue future  
 (c) exit the outermost block even if it occurs inside the innermost  
 (d) continue the compilation even an error occurs in a program
12. Which is the incorrect statement?  
 (a) while loop can be nested  
 (b) for loop can be nested  
 (c) Both (a) and (b) are correct  
 (d) One type of loop cannot be nested in other type
13. The break statement is used in  
 (a) selective control structure only  
 (b) loop control structure only  
 (c) both (a) and (b)  
 (d) switch-case control structure only
14. Which of the following statements results in infinite loop?  
 (a) for (i = 0; ; i++)  
 (b) for (i = 0; ; );  
 (c) for (; );  
 (d) All of the above
15. How many x are printed?  
 for (i = 0, j = 10; i<j; i++, j - -)  
 System.out.println(x);  
 (a) 10                (c) 4  
 (b) 5                (d) None of the above
16. In the following loop construct, which one is executed only once always for (expr1; expr2; expr3)  
 (a) expr1  
 (b) expr3  
 (c) expr1 and expr3  
 (d) expr1, expr2 and expr3
17. In Java if (x) is equal to  
 (a) if (x = false) where x must be a Boolean value  
 (b) if (x == false) where x must be a Boolean value  
 (c) if (x!= false) where x must be a Boolean value  
 (d) if (x = True) where x must be Boolean value
18. In Java if(!x) is equal to  
 (a) if (x = True) where x must be a Boolean value  
 (b) if (x == True) where x must be Boolean value  
 (c) if (x = false) where x must be Boolean value  
 (d) if (x = = false) where x must be Boolean value
19. Null statements are useful for  
 (a) time delay  
 (b) to run loop infinite times  
 (c) not running a loop statement  
 (d) none of the above

## KEY FOR MULTIPLE CHOICE QUESTIONS

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. b  | 2. a  | 3. b  | 4. b  | 5. a  | 6. c  | 7. d  | 8. d  | 9. b  | 10. b |
| 11. a | 12. d | 13. c | 14. d | 15. b | 16. a | 17. c | 18. d | 19. a |       |

# 5

# Working with Array in Java

## 5.1 INTRODUCTION

---

An array is a data structure which can store multiple elements of same type under one name. In other words, an array is a collection of variables of the same type, all of which are referred by a common name. The specific element of an array is accessed by an index. One single variable can store only one value at a time. In order to work with multiple data items at the same time, all of which have some common features, e.g., age of all students, salary of all employees, etc. then instead of creating separate variable for each data item an array can be used.

The declaration of an array is done in the following way:

```
int arr [];
```

By the above statement an array is named as `arr`. While declaring an array, the size of the array is not specified. The size is given later using new operator which is discussed in the next section. The size is specified in `[]` (square brackets), known as subscript operator. Array name is an identifier which follows the rules of writing an identifier. All the elements of an array are stored in contiguous with the first element accessed as `arrayname[0]`, second as `arrayname[1]` and so on. In our case, it will be `arr[0], arr[2]` upto `arr[4]`. The first element of an array starts at index 0 and the last element at index `arraysize1`.

## 5.2 CREATING ARRAYS IN JAVA

---

In order to create an array, it has to be first declared in the following way:

```
int [] list;
```

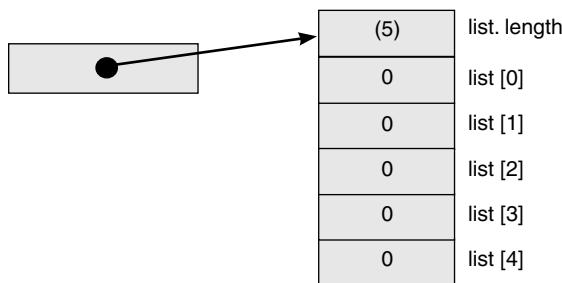
This creates a reference named `list` to an array of integer type of any length. Note that the syntax of declaring reference to an array as square bracket is in between array name and data type `int`. It can also be written as `int list []`.

Initially, the value of `list` is null (if it is a member variable in a class) or undefined (if it is a local variable in a method). The new operator is used to create a new array object, which can then be assigned to `list`. The syntax for using new with array is as follows:

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

This syntax creates an array of five integers. More generally, the constructor ‘`new BaseType [N]`’ is used to create an array belonging to the class `BaseType` `[ ]`. The value `N` in brackets specifies the length of the array, i.e., the number of elements that it contains. Note that the array ‘knows’ how long it is. The length of the array is an instance variable in the array object. In fact, in the length of an array, `list` can be referred to as `list.length`. (However, it is not allowed to change the value of `list.length`, so it is really a ‘final’ instance variable, i.e., one whose value cannot be changed after it has been initialized).

Actually, the length of property of the array is placed on top of the first element in memory as shown below.



The bracket in an array reference can contain any expression whose value is an integer. For example, if `indx` is a variable of type `int` then `list [indx]` and `list [2 * indx + 7]` are syntactically correct references to elements of the array `list`. Thus, the following loop would point to all the integers in the array `list` to standard output.

```
for(int i=0;i<list.length;i++)
{
 System.out.println(list[i]);
}
```

The first time through the loop, ‘`i`’ is 0 and `list[i]` refers to `list[0]`. So, it is the value stored in the variable `list[0]` that is printed. The second time through the loop ‘`i`’ is 1 and the value stored in the continuation condition `list.length` is no longer true.

Every use of a variable in a program specifies a memory location. Think for a moment about what the computer does when it encounters a reference to an array element, `list[k]`, while it is executing a program. The computer must determine which memory location is being referred to. To the computer, `list[k]` means something like this: ‘Get the pointer that is stored in the variable `list`. Follow this pointer to find an array object. Get the value of “`k`”. Go to the `k`th position in the array and that is the memory location you want’. There are two things that can go wrong here. Suppose, the value of `list` is null. If that is the case, then the `list` does not even refer to an array. The attempt to refer to an element of an array that does not exist is an error. This is an example of a ‘null pointer’ error. The second possible error occurs if `list` does refer to an array, but the value of ‘`k`’ is outside the legal range of indices for that array. This will happen if `k < 0` or if `k > = list.length`. This is called an `array index out of bound` error. When arrays in a program are used, one should be alert that both types of errors are possible. However, `array index out of bounds` errors is by far the most common error when working with arrays.

### 5.3 SOME IMPORTANT POINTS ABOUT ARRAY

1. Array is a homogeneous data structure that stores elements of similar type, i.e., five elements of `int` type, 10 elements of `float` type, 20 elements of `char` type, etc.
2. All the elements share the same name and each can be modified individually, for example:

```
int []a=new int a[5];
a[0]=30;
a[2]=36;
```

Now, changing the value in `a[0]` does not affect the value of all other elements.

3. Though each element uses the common name of array but each element is treated separately and stored at separate memory location.
4. One array can be assigned to another array as given in the code below.

```
int a[]={};
int []x=a;
a[0]=12;
System.out.println(x[0]); //prints 12
```

But here, results are assigned in creating a reference for the array. Hence, 'x' and 'a' represent the same array.

5. The most important point to note about arrays in Java is that they are allocated memory dynamically using new operator. One cannot create an array statically as int arr[5];
6. Java checks the boundary of the array. If index goes beyond, a run time error is generated.
7. The number of subscripts determines the dimensionality of an array. For example, int a[] is the declaration of one-dimensional array written as 1-D. Similarly, int a[][] declares 2-D array.
8. Java arrays are objects.
9. Arrays are created using a form of the new operator. No variable can ever hold an array; a variable can only refer to an array. Any variable that can refer to an array can also hold the null value, meaning that it does not refer to anything at the moment.
10. The elements of the array are, essentially, instance variables in the array object, except that they are referred to by number rather than by name.
11. Once the size of the array have been fixed, one cannot grow it or shrink it at run time, i.e., insertion and deletion is a costly affair with array.

## 5.4 INITIALIZING 1-D ARRAY

For an array variable, just as for any variable, one can declare the variable and initialize it in a single step. For example:

---

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

---

To initialize the array we can write in the following manner:

```
int []list ={1, 4, 9, 16, 25, 36, 49};
float []farr={1.2f, 3.4f, 5.6f, 4.5f};
char str[]={'h', 'e', 'j','o'};
boolean b[]={true, false, false, true};
```

Note that one can write an array in the following manner:

```
int [] list;
list = new int [] { 1, 8, 27, 64, 125, 216, 343};
```

But not in the manner as given below.

```
list = {1, 8, 27, 64, 125, 216, 343};
```

## 5.5 PROGRAMMING EXAMPLES (PART-1)

A few examples of creating 1-D array are as follows.

```
/* Prog 5.1 INITIALIZING AND DISPLAYING ARRAY VER 1 */

class JPS1
{
 public static void main(String args[])
 {
 int[] arr ={ 0, 1, 2, 3, 4 };
 for (int i = 0; i < arr.length; i++)
 System.out.print(arr[i] + " ");
 }
}

OUTPUT:
0 1 2 3 4
```

*Explanation:* The program is self-explanatory.

```
/* PROG 5.2 TAKING ARRAY ELEMENTS FROM USER */

import java.io.*;
import java.util.*;
class JPS3
{
 public static void main(String[]args)
 {
 Scanner sc = new Scanner(System.in);
 int size, i;
 int[] arr;
 System.out.print("\n\nEnter size of array:= ");
 size = sc.nextInt();
 arr = new int[size];
 for (i=0;i<size;i++)
 {
 System.out.print("\n Enter arr["+i+"]"
 " element:");
 arr[i] = sc.nextInt();
 }
 System.out.println("\n\nArray elements are \n");
 for (i=0;i<size;i++)
 System.out.println("arr["+i+"] := "+arr[i]);
 }
}
```

## OUTPUT:

```

Enter size of array: = 4
Enter arr[0] element:11
Enter arr[1] element:12
Enter arr[2] element:13
Enter arr[3] element:14
Array elements are
arr[0]:= 11
arr[1]:= 12
arr[2]:= 13
arr[3]:= 14

```

*Explanation:* The size of the array is taken from the user. The size taken is thus used for creating the array. Using for loop all array elements are taken from the user and later displayed.

```

/*PROG 5.3 OUTPUT OF UN-INITIALIZED ARRAY */

import java.io.*;
import java.util.*;
class JPS4
{
 public static void main(String[] args)
 {
 int i;
 int[] arr;
 arr = new int[5];
 System.out.println("Array elements are \n");
 for (i = 0; i < arr.length; i++)
 System.out.println("arr[" + i + "] = " + arr[i]);
 }
}

```

OUTPUT:

```

Array elements are
arr[0]= 0
arr[1]= 0
arr[2]= 0
arr[3]= 0
arr[4]= 0

```

*Explanation:* The default value of uninitialized int, float, double array is 0. For boolean, it is false and for string, it is null. Check out by changing the array type from int to some other type.

```

/*PROG 5.4 ACCESSING ELEMENTS BEYOND BOUNDARY */

```

```

import java.io.*;
import java.util.*;
class JPS5
{

```

```

public static void main(String[] args)
{
 int i;
 int[] arr = new int[] { 1, 2, 3, 4, 5 };
 System.out.println(arr[7]);
}
}

OUTPUT:
Exception in thread "main" java.lang.
ArrayIndexOutOfBoundsException: 7
at JPS5.main(JPS5.java:10)

```

*Explanation:* When elements are accessed beyond boundary no compilation error occurs, instead exception, i.e., runtime error, occurs. For more about this case, see the chapter titled ‘Exception Handling in Java’.

```

/*PROG 5.5 TO FIND MINIMUM ELEMENT OF 1-D ARRAY */

import java.io.*;
import java.util.*;
class JPS7
{
 public static void main(String args[])
 {
 int i, max, min;
 Scanner sc = new Scanner(System.in);
 int[] arr = new int[5];
 for (i = 0; i < arr.length; i++)
 {
 System.out.print("\n Enter arr["+i+ "] elements:");
 arr[i] = sc.nextInt();
 }
 System.out.println("\n Array elements are");
 for (i = 0; i < arr.length; i++)
 System.out.println("arr["+i+"]= "+arr[i]);
 max = arr[0];
 min = arr[0];
 for (i=1;i<arr.length;i++)
 {
 if (min>arr[i])
 min = arr[i];
 if (max < arr[i])
 max = arr[i];
 }
 System.out.println("Max element is " + max);
 System.out.println("Min element is " + min);
 }
}

```

## OUTPUT:

```

Enter arr[0]elements:12
Enter arr[1]elements:13
Enter arr[2]elements:14
Enter arr[3]elements:15
Enter arr[4]elements:16

Array elements are
arr[0]= 12
arr[1]= 13
arr[2]= 14
arr[3]= 15
arr[4]= 16
Max element is 16
Min element is 12

```

*Explanation:* To find the maximum element of an array, it is assumed that the first element of the array is maximum by storing the first element in the variable max. Then this max variable is compared with the remaining elements of the array. If max is less than any other element of the array then that element is assigned to max. For finding minimum element of the array, it is again assumed that the first element is the minimum element and stored in variable min. It is checked if min is greater than any other element of the array, and if so, that element is assigned to min.

```

/*PROG 5.6 SEARCHING AN ELEMENT IN THE ARRAY */

import java.io.*;
import java.util.*;
class JPS8
{
 public static void main(String[] args)
 {
 int i = 0, num;
 boolean found = false;
 final int S = 5;
 Scanner sc = new Scanner(System.in);
 int[] arr = new int[S];
 System.out.println("Enter " + S + " elements");
 while (i < s)
 arr[i++] = sc.nextInt();
 System.out.println("\n Array elements are:");
 for (i = 0; i < arr.length; i++)
 System.out.println("arr["+i+"]:= "+arr[i]);
 System.out.println("Enter the element which you
 want to search");
 num = sc.nextInt();
 for (i = 0; i < s; i++)
 {
 if (arr[i] == num)
 {
 found = true;
 break;
 }
 }
 }
}

```

```

 }
 if (found)
 System.out.println("Element exist at
 location := " + i);
 else
 System.out.println("Element does not exist
 in the array");
 }
}

OUTPUT:
Enter 5 elements
23
56
78
90
45
Array elements are:
arr[0] := 23
arr[1] := 56
arr[2] := 78
arr[3] := 90
arr[4] := 45
Enter the element which you want to search:=90
Element exist at location := 3

```

*Explanation:* The element to be searched is taken in the num. The variable found is initialized to false in the beginning. In the for loop, num is compared with each of the array element and if a match occurs found = true is made and one gets exited from the loop. The value of found = true ensures that a match has occurred. Outside the loop, the value of flag is checked and the result displayed accordingly. The program does not take into account how many times num appears in the array. It simply searches whether num is there in the array or not.

```

/*PROG 5.7 SORTING ELEMENTS OF ARRAY */

import java.io.*;
import java.util.*;
class JPS9
{
 public static void main(String[] args)
 {
 int i = 0, j, t;
 final int S = 6;
 Scanner sc = new Scanner(System.in);
 int[] arr = new int[S];
 System.out.println("\nEnter "+S+" elements\n");
 while (i < s)
 arr[i++] = sc.nextInt();
 System.out.println("\nOriginal Array \n");
 for (i = 0; i < s; i++)

```

```

 System.out.print(arr[i] + "\t");
 /*main logic starts here */
 for (i = 0; i < s; i++)
 for (j = i + 1; j < s; j++)
 if (arr[i] > arr[j])
 {
 t = arr[i];
 arr[i] = arr[j];
 arr[j] = t;
 }
 System.out.println("\n\nSorted Array");
 for (i = 0; i < s; i++)
 System.out.print(arr[i] + "\t");
}
}

OUTPUT:
Enter 6 elements
45 67 2 3 1 88
Original Array
45 67 2 3 1 88
Sorted Array
1 2 3 45 67 88

```

*Explanation:* Sorting means arrangement of array elements in either ascending or descending order. In the program, the array in ascending order is sorted.

In order to understand the logic behind the program, five elements of array are taken and the logic is traced as given below.

| Index     | a[0] | a[1] | a[2] | a[3] | a[4] |
|-----------|------|------|------|------|------|
| Initially | 4    | 0    | 3    | 1    | 2    |

For  $i = 0$ , inner for loop runs for  $j = 1$  to  $j = 4$  and it compares  $a[0]$  with all other elements of the array, i.e.,  $a[1], a[2], a[3], a[4]$ . If any of the elements is greater than  $a[0]$  then both the elements are swapped. For example,  $a[0]$  is 4 and  $a[1]$  is 0, if condition turns out to be true then  $a[0]$  and  $a[1]$  are swapped. Now the array becomes as given below.

| Index     | a[0] | a[1] | a[2] | a[3] | a[4] |
|-----------|------|------|------|------|------|
| Initially | 0    | 4    | 3    | 1    | 2    |

For  $j = 2$  to  $j = 4$ , if condition is false and array does not change. This completes one iteration of outer for loop. It can be checked that after the first iteration of outer for loop, the smallest element of the array occupies the first place in the array. Continuing in this manner for  $i = 1$  and running  $j = 2$  to 4, it will be seen that  $i = 1$  and  $j = 2$ , if condition is satisfied and  $a[1]$  and  $a[2]$  are swapped array which becomes as follows:

| Index     | a[0] | a[1] | a[2] | a[3] | a[4] |
|-----------|------|------|------|------|------|
| Initially | 0    | 3    | 4    | 1    | 2    |

For  $i = 1$  and  $j = 3$ , if condition is satisfied and  $a[1]$  and  $a[3]$  are swapped array which becomes as follows:

| Index     | a[0] | a[1] | a[2] | a[3] | a[4] |
|-----------|------|------|------|------|------|
| Initially | 0    | 1    | 4    | 3    | 2    |

So, after the end of inner for loop the array will be as follows:

| Index     | a[0] | a[1] | a[2] | a[3] | a[4] |
|-----------|------|------|------|------|------|
| Initially | 0    | 1    | 4    | 3    | 2    |

This means that the second smallest number is at the second place in the array. So, when the outer loop finishes, the array will be stored in ascending order.

## 5.6 TWO-DIMENSIONAL (2-D) ARRAY

A two-dimensional array can be thought of as a matrix with rows and columns. A multi-dimensional array is like an array of arrays. For example, if we declare a 2-D array in the following way:

```
int arr[][] = new int[3][4];
```

|    | C1        | C2        | C3        | C4        |
|----|-----------|-----------|-----------|-----------|
| R1 | arr[0][0] | arr[0][1] | arr[0][2] | arr[0][3] |
| R2 | arr[1][0] | arr[1][1] | arr[1][2] | arr[1][3] |
| R3 | arr[2][0] | arr[2][1] | arr[2][2] | arr[2][3] |

The first row and column starts at 0 and intersection of row and column becomes the position of element of the 2-D array, for example, first row(0) and second column(1) element will be arr[0][1].

In terms of array of arrays, it can be stated that there are three one-dimensional arrays and each 1-D array is having four elements.

A 2-D array can be created and initialized in the following way:

```
int [][]arr = { {1, 2, 3}, {4, 5, 6}};
```

This creates a 2-D array of two rows and three columns. It can also be declared in the following manner without initializing it.

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

or

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

In case of a 2-D array, the length prototype with array name gives only number of rows and not number of elements in the 2-D array itself. For example, see the code given below.

```
int [][]arr=new int[3][4];
System.out.println(arr.length);
//prints number of rows i.e. 3
for(i=0;i<3;i++)
System.out.println(arr[i].length+" "); //prints 4 4 4
```

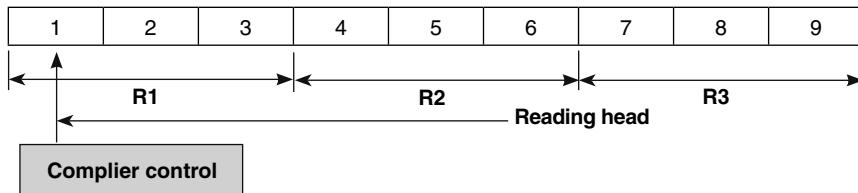
The above code proves that a 2-D array is nothing but a collection of 1-D array. The main array `arr` is 2-D array so `arr.length` gives a number of rows. As each row is a 1-D array, `arr[i].length` (where `i` may be between 0 and 3, inclusive both) gives number of columns in each row of the array.

The above representation of 2-D array is only for illustration purpose. Inside the memory, the array is represented either in row-major order or in column-major order. In row-major order, the rows are listed on the basis of number of columns. For example, the 2-D array `arr[3][3]` is represented in memory in row-major order in the following way:

| Row/Column | C1 | C2 | C3 |
|------------|----|----|----|
| R1         | 1  | 2  | 3  |
| R2         | 4  | 5  | 6  |
| R3         | 7  | 8  | 9  |

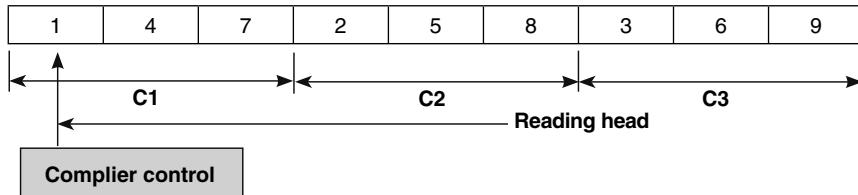
**Figure 5.1** A 2-D array of order `a[3][3]`

Diagrammatically, the row-major implementation can be represented in the following way:



**Figure 5.2** Row-major implementation of 2-D array

Java follows row-major order representation of 2-D array in memory. In column-major order, the columns are listed on the basis of the number of rows. For example, the 2-D array `arr[3][3]` is represented in memory in column-major order in the following way:



**Figure 5.3** Column-major implementation of 2-D array

In Java, an array is guaranteed to be initialized and cannot be accessed outside of its range. The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run time. If an array index is accessed outside the array range, exceptions are thrown. The same argument holds for strings.

```
/*PROG 5.8 DEMO OF 2-D ARRAY */
```

```
class JPS10
{
 public static void main(String args[])
 {
```

```

int[][]arr={{1, 2, 3 },{ 4, 5, 6},{7,8,9}};
 int i, j;
 System.out.println("\n\nThe matrix is ");
for (i = 0; i < arr.length; i++)
{
 for (j=0; j<arr[i].length;j++)
 System.out.print(arr[i][j]+" ");
 System.out.println("\n");
}
}

OUTPUT:
The matrix is
1 2 3
4 5 6
7 8 9

```

*Explanation:* A 2-D array of three rows and three columns is initialized with nine elements. For two-dimensional arrays, all the elements have to be first stored (depends on number of columns) for first row and then for second upto total number of rows. For this, two loops are required—one which controls the rows and another which controls the columns. The outer loop controls the rows and inner loop elements of each 1-D arrays. Through this, array elements in the form of matrix have been displayed.

```

/*PROG 5.9 DEMO OF 2-D ARRAY, INPUT FROM USER */

import java.io.*;
import java.util.*;
class JPS11
{
 public static void main(String []args)
 {
 int row, col, i, j;
 int [][]arr={{0}};
 Scanner sc=new Scanner(System.in);
 System.out.print("\nEnter number of rows :=");
 row=sc.nextInt();
 System.out.print("Enter number of cols:=");
 col=sc.nextInt();
 arr=new int[row][col];
 for(i=0;i<row;i++)
 {
 for(j=0;j<col;j++)
 {
 System.out.print("Enter the arr["+i+"]["+j+"]:= ");
 arr[i][j]=sc.nextInt();
 }
 }
 System.out.println("\nMatrix is \n");
 for(i=0;i<row;i++)

```

```

 {
 for(j=0;j<col;j++)
 System.out.print(" "+arr[i][j]);
 System.out.println();
 }
 }

OUTPUT:

Enter number of rows:=2
Enter number of cols:=2
Enter the arr[0][0]:= 11
Enter the arr[0][1]:= 12
Enter the arr[1][0]:= 13
Enter the arr[1][1]:= 14
Matrix is
11 12
13 14

```

*Explanation:* Dimensions for rows and columns are taken from the user and an array is declared. Next, the value for all rows and columns are taken using two for loops. The same entered values are displayed back.

```

/*PROG 5.10 TRANSPOSE OF A MATRIX */

import java.io.*;
import java.util.*;
class JPS12
{
 public static void main(String[] args)
 {
 int arr[][] = new int[3][2];
 int i, j;
 Scanner sc = new Scanner(System.in);
 System.out.println("\n\nEnter the 6 elements of
 matrix\n");
 for (i = 0; i < arr.length; i++)
 for (j = 0; j < arr[j].length; j++)
 arr[i][j] = sc.nextInt();
 System.out.println("\nThe matrix is");
 for (i = 0; i < arr.length; i++)
 {
 for (j = 0; j < arr[j].length; j++)
 System.out.print(" " + arr[i][j]);
 System.out.println();
 }
 System.out.println("\n\nThe transpose of matrix
 is");
 for (i = 0; i < arr[i].length; i++)
 {

```

```

 for (j = 0; j < arr.length; j++)
 System.out.print(" " + arr[j][i]);
 System.out.println();
 }
}

OUTPUT:

Enter the 6 elements of matrix
11 12 13 14 15 16
The matrix is
11 12
13 14
15 16
The transpose of matrix is
11 13 15
12 14 16

```

*Explanation:* The transpose of a matrix is matrix with its row and column interchanged, e.g., if the matrix is written in Java.

```

int a[][] = {
 {1,2}, {4, 5}, {6, 8}
};

```

Diagrammatically, it can be represented in the following way:

|   |   |
|---|---|
| 1 | 2 |
| 4 | 5 |
| 6 | 8 |

which means, a matrix of three rows and two columns. The transpose matrix is as follows:

```
int a[][] = { { 1, 4, 6}, {2, 5, 8}};
```

|   |   |   |
|---|---|---|
| 1 | 4 | 6 |
| 2 | 5 | 8 |

which means, a matrix of two rows and three columns. To print transpose of a matrix, simply change the row and column and while printing interchange i and j.

```
/* PROG 5.11 SUM OF DIAGONAL ELEMENTS OF A MATRIX */
```

```

import java.io.*;
import java.util.*;
class JPS13
{
 public static void main(String[] args)
 {
 int arr[][] = new int[3][3];
 int i, j, dsum = 0;

```

```

Scanner sc = new Scanner(System.in);
System.out.println("\n\nEnter the 9 elements
of matrix");
for(i=0;i<arr.length;i++)
for(j=0;j<arr[i].length;j++)
arr[i][j] = sc.nextInt();
System.out.println("The matrix is");
for(i=0;i<arr.length;i++)
{
 for(j=0;j<arr[i].length; j++)
 System.out.println(" " + arr[i][j]);
 System.out.println();
}
for(i=0;i<arr.length;i++)
 for(j=0;j<arr[i].length; j++)
 if (i == j)
 dsum = dsum + arr[i][j];
System.out.println("Sum of diagonal elements is:-
" + dsum);
}
}

OUTPUT:
Enter the 9 elements of matrix
1 2 3 4 5 6 7 8 9
The matrix is
1 2 3
4 5 6
7 8 9
Sum of diagonal elements is:=15

```

*Explanation:* The diagonal element of a 2-D array can be calculated only if the number of rows and columns is same. That type of matrix is called square matrix. Diagonal elements of a matrix are those elements where rows and columns are same, i.e., row 0 col 0, row 1 col 1, etc. To find the sum of diagonal elements, it is checked if the row is equal to the column and if found equal, their sum is taken into account. Look at the given matrix.

|          |          |          |
|----------|----------|----------|
| <b>2</b> | 3        | 5        |
| 6        | <b>8</b> | 9        |
| 12       | 7        | <b>4</b> |

**Sum = 2 + 8 + 4 = 14**

In this case, the sum of diagonal elements (shown in bold) is 14.

```
/*PROG 5.12 MULTIPLICATION OF TWO MATRICES */
```

```

import java.io.*;
import java.util.*;
class JPS14
{

```

```

static void show(String s)
{
 System.out.println(s);
}
public static void main(String[] args)
{
 final int row = 3;
 final int col = 3;
 int mat1[][] = new int[row][col];
 int mat2[][] = new int[row][col];
 int mat3[][] = new int[row][col];
 int i, j, k;
 Scanner sc = new Scanner(System.in);
 show("\n\nEnter the " +(row * col)+" elements of
 first matrix\n");
 for (i = 0; i < row; i++)
 for (j = 0; j < col; j++)
 mat1[i][j] = sc.nextInt();
 show("\nEnter the " +(row * col) + " elements of
 second matrix\n");
 for (i = 0; i < row; i++)
 for (j = 0; j < col; j++)
 mat2[i][j] = sc.nextInt();
 show("The first matrix is\n");
 for (i = 0; i < row; i++)
 {
 for (j = 0; j < col; j++)
 System.out.print(" " + mat1[i][j]);
 show(" ");
 }
 show("The second matrix is\n");
 for (i = 0; i < row; i++)
 {
 for (j = 0; j < col; j++)
 System.out.print(" " + mat2[i][j]);
 show(" ");
 }
 /*Main logic starts here */
 for (i = 0; i < 3; i++)
 {
 for (j = 0; j < 3; j++)
 {
 mat3[i][j] = 0;
 for (k = 0; k < 3; k++)
 mat3[i][j]=mat3[i][j]+mat1[i][k]*mat2[k][j];
 }
 }
 /*Main logic ends here */
 show("\n Multiplication of two matrices is\n\n");
 for (i = 0; i < 3; i++)
 {

```

```

 for (j = 0; j < 3; j++)
 System.out.print(" " + mat3[i][j]);
 show(" ");
 }
}

OUTPUT:
Enter the 9 elements of first matrix
1 1 1 1 1 1 1 1 1
Enter the 9 elements of second matrix
2 2 2 2 2 2 2 2 2
The first matrix is
1 1 1
1 1 1
1 1 1
The second matrix is
2 2 2
2 2 2
2 2 2
Multiplication of two matrices is
6 6 6
6 6 6
6 6 6

```

*Explanation:* For matrix multiplication of two matrices, the number of col of first matrix must be equal to the number of row of the second matrix, i.e., if the dimension of the first matrix is  $m$  by  $n$ , the dimension of the second matrix must be  $n$  by  $p$  and the resultant matrix will be of order  $m$  by  $p$ . The logic is as follows:

Two matrices of order 3 by 3 are taken; suppose the order of first matrix is  $m$  by  $n$  and that of second matrix is  $n$  by  $p$ .

|   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 |   | 2 | 4 | 6 |   | 29 | 11 | 23 |
| 4 | 3 | 2 |   | 6 | 2 | 3 | = | 36 | 24 | 37 |
| 2 | 5 | 6 | ↓ | 5 | 1 | 4 |   | 64 | 24 | 49 |

**Figure 5.4** Procedure for matrix multiplication

In the main logic, the first loop runs from 0 to  $m$ . The second inner loop runs from 0 to  $p$  and the third inner loop runs from 0 to  $n$  (common column of first matrix and row of second matrix). As the direction of arrow shows initially, the first row is chosen for value of  $i = 0$  and then for value of  $j = 0$  and  $k$  runs from 0 to 3. Hence, the value becomes as follows:

```

C [0] [0] = 0;
i = 0, j = 0, k = 0;
C [0] [0] = c [0] [0] + a [0] [0] * b [0] [0];
= 0 + 1*2;
= 2;
K = 1;

```

```
C [0] [0] = c [0] [0] + a [0] [1] * b [1] [0];
 = 2 + 2 * 6;
 = 2 + 12;
 = 14;
K = 2;
```

```
C [0] [0] = c [0] [0] + a [0] [2] * b [2] [0];
 = 14 + 3 * 5;
 = 14 + 15;
 = 29;
```

This is the value of first element of the result in the first column and first row of the resultant matrix.

After this, the third inner loop finishes. Now, the second column of the second matrix has to be chosen and therefore,  $j$  becomes  $i$  and for value of  $i = 0, j = 1, k$  again runs from 0 to 3.

```
C [0] [1] = 0;
```

```
 i = 0, j = 1, k = 0;
```

```
C [0] [1] = c [0] [1] + a [0] [0] * b [0] [1];
 = 0 + 1 * 4;
 = 4;
K = 1;
```

```
C [0] [1] = c [0] [1] + a [0] [1] * b [1] [1];
 = 4 + 2 * 2;
 = 4 + 4;
 = 8;
K = 2;
```

```
C [0] [1] = c [0] [1] + a [0] [2] * b [2] [1];
 = 8 + 3 * 1;
 = 11;
```

This is the value of the first row and the second column of the resultant matrix. After this, the third inner loop finishes. Now, the third column of the second matrix has to be chosen so that  $j$  becomes 2 and for the value of  $i = 0, j = 2, k$  again runs from 0 to 3.

```
C [0] [2] = 0;
```

```
 I = 0; j = 2; k = 0;
```

```
C [0] [2] = c [0] [2] + a [0] [0] * b [0] [2];
 = 0 + 1 * 5;
 = 5;
K = 1;
```

```
C [0] [2] = c [0] [2] + a [0] [1] * b [1] [2];
 = 5 + 2 * 3;
 = 5 + 6;
 = 11;
K = 2;
```

```
C [0] [2] = c [0] [2] + a [0] [2] * c [2] [2];
 = 11 + 3 * 4;
 = 11 + 12;
 = 23;
```

That is the value of the first row and the third column of the resultant matrix. After this, the value of ' $j$ ' exhausts and the value of ' $i$ ' now becomes 1 and the process repeats.

## 5.7 THREE-DIMENSIONAL (3-D) AND VARIABLE COLUMN LENGTH ARRAYS

Java allows us to set different column sizes for each row of array. For example, one can create an array in the following manner:

```
int [][]arr=new int[3] [];
arr[0]=new int[4]; //first row of 4 columns
arr[1]=new int[5]; //second row of 5 columns
arr[2]=new int[6]; //third row of 6 columns
```

The array in memory looks like the following way:

|           |           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|-----------|
| arr[0][0] | arr[0][1] | arr[0][2] | arr[0][3] |           |           |
| arr[1][0] | arr[1][1] | arr[1][2] | arr[1][3] | arr[1][4] |           |
| arr[2][0] | arr[2][1] | arr[2][2] | arr[2][3] | arr[2][4] | arr[2][5] |

```
int [][]arr=new int[3] [];
arr[0]=new int[4]; //first row of 4 columns
arr[1]=new int[5]; //second row of 5 columns
arr[2]=new int[6]; //third row of 6 columns
int x=1,i,j;
for(i=0;i<arr.length;i++)
for(j=0;j<arr[i].length;j++)
arr[i][j]=x++;
for(i=0;i<arr.length;i++)
{
 for(j=0;j<arr[i].length;j++)
 System.out.print("|"+arr[i][j]+" |");
 System.out.println();
}
```

As explained earlier, `arr.length` gives the number of rows for the arrays `arr`. For finding length of the columns for each row, `arr[i].length` can be written where '`i`' is any valid row number.

### 5.7.1 3-D Array

A 3-D array can be thought of as a collection of 2-D array which can again be thought of as a 1-D array. For example, there can be a 3-D array declaration in the following order:

```
int arr[][][] 5 new int[2][3][4];
```

The above declaration can be thought of as an array of two elements where each element is a 2-D array of size 3 by 4. These usually do not occur in practice, so programs will be presented to demonstrate how to initialize and print them. To find the length of each dimension, the `length` property of the array can be used as shown below.

```
System.out.println(arr.length);
System.out.println(arr[0].length);
System.out.println(arr[0][0].length);
```

In the above case, the first statement prints 2, second statement prints 3 and third statement prints 2. An array of objects can be created, which will be discussed when dealing with classes and objects.

```
/*PROG 5.13 INTIALIZING AND DISPLAYING A 3-D ARRAY */

import java.io.*;
import java.util.*;
class JPS15
{
 public static void main(String[] args)
 {
 int arr[][][] = {{{1,2},{2,3},{3,4}},{ {3,2},{2,5},
 {7,2}}};
 int i, j, k;
 for (i = 0; i < arr.length; i++)
 {
 System.out.println("2-D Array " + (i + 1));
 for (j = 0; j < arr[i].length; j++)
 {
 for (k = 0; k < arr[i][j].length; k++)
 System.out.print(" " + arr[i][j][k]);
 System.out.println();
 }
 System.out.println("\n");
 }
 }
}

OUTPUT:
2-D Array 1
1 2
2 3
3 4

2-D Array 2
3 2
2 5
7 2
```

***Explanation:***

```
int arr[][][] = {{{1,2},{2,3},{3,4}},{ {3,2},{2,5},
 {7,2}}};
```

The above declaration of a 3-D array with initialization can be understood as follows:

{1, 2} represents 1-D array with two elements and {{1, 2}, {2, 3}, {3, 4}} represents the first 2-D array. Any element of this array will be represented by a[0][j][k] (j from 0 to 2 and k from 0 to 1). Similarly, any element of the second 2-D array will be represented as a[1][j][k] (j from 0 to 2 and k from 0 to 1).

```
/*PROG 5.14 TAKING INPUT FOR 3-D ARRAY AND DISPLAYING THEM BACK */
```

```
import java.io.*;
import java.util.*;
class JPS16
{
 public static void main(String[] args)
 {
 int arr[][] []=new int[2][3][2];
 int i,j,k;
 Scanner sc=new Scanner(System.in);
 for(i=0;i<arr.length;i++)
 {
 System.out.println("Enter elements for the
 2-D array "+(i+1));
 for(j=0;j<arr[i].length;j++)
 {
 for(k=0;k<arr[i][j].length;k++)
 {
 System.out.print("\nEnter arr["+i+"]"
 +"["+j+"]"+"["+k+"]");
 System.out.print("element:");
 arr[i][j][k]=sc.nextInt();
 }
 }
 }
 for(i=0;i<arr.length;i++)
 {
 System.out.println("2-D Array "+(i+1));
 for(j=0;j<arr[i].length;j++)
 {
 for(k=0;k<arr[i][j].length;k++)
 System.out.print(" "+arr[i][j][k]);
 System.out.println();
 }
 System.out.println("\n");
 }
 }
}
```

**OUTPUT:**

```
Enter elements for the 2-D array 1
Enter arr[0][0][0]element:11
Enter arr[0][0][1]element:12
Enter arr[0][1][0]element:13
Enter arr[0][1][1]element:14
Enter arr[0][2][0]element:15
Enter arr[0][2][1]element:16
Enter elements for the 2-D array 2
```

```

Enter arr[1][0][0]element :17
Enter arr[1][0][1]element :18
Enter arr[1][1][0]element :19
Enter arr[1][1][1]element :20
Enter arr[1][2][0]element :21
Enter arr[1][2][1]element :22

2-D Array 1
11 12
13 14
15 16

2-D Array 2
17 18
19 20
21 22

```

## 5.8 PONDERABLE POINTS

1. An array is an ordered sequence of finite data items of the same data type that shares a common name. The common name is the array name and each individual data item is known as an element of the array.
2. The variable `arr[i][j]` represents the element in the *i*th row and *j*th column.
3. The subscript for an array must be an integer value of any integer expression.
4. For all arrays, the length is static and the final property stores the length of the array, i.e., number of elements in the array.
5. In the declaration of an array, the size is not specified. The size is specified using the new operator.
6. In Java, all arrays are created dynamically and not statically.
7. Arrays can be declared as `int [] arr;` or `int arr[];` The allocation of memory is done as `arr = new int [5];`
8. Java allows the creation of multidimensional array of variant column length.

## REVIEW QUESTIONS

1. What is the difference between a component and an element of an array?
2. What types are valid for array indexes?
3. What happens if the sequential search is applied to an element that occurs more than once in the array?
4. What is wrong with this definition?  

```
Arrays arrays = new Arrays ();
```
5. What is an `ArrayIndexOutOfBoundsException` exception, and how does its use distinguish Java from other languages such as C and C++?
6. What is the simplest way to print an array of objects?
7. Can an array store elements of different types?
8. What is wrong with this code?  

```
char []name = "Java
Programming";
```
9. Write and run a Java program that inputs three names and print them in their alphabetical order.
10. Write a Java program that will print the subscript of an array and the corresponding element of it.
11. What is vector? What is the difference between a vector object and an array object?
12. What does it mean to say that Java does not allow multidimensional array?
13. Run a test program to see how the `Array.fill()` method handles an array of objects.

14. Write and test this method:

```
boolean isSorted(int[] a)
//returns true iff a[0]<=a[1]
<=a[2].....<=a[a.length-1]
```

15. Write and test this method:

```
int minimum(int[] a)
//returns the minimum element of
a[]
```

16. Write and test this method:

```
void shuffle(Object[] a)
//randomly permutes the elements
of a[]
```

17. Write and test this method:

```
double innerProduct(double[] x,
double[] y)
//returns the algebraic inner product
(the sum of the component-
//wise product) of the two given
arrays as (algebraic) vectors
```

18. Write and test this method:

```
int[][] pascal(int size)
//returns Pascal's triangle of
the given size
```

19. A number is said to be palindrome if it is invariant under reversion; that is, the number is the same if its digits are reversed. For example, 3456543 is palindromic. Write a program that checks each of the first 10,000 prime numbers and prints those that are palindromic.

20. Write and test this method:

```
double[][] outerProduct (double[] x,
double[] y)
//returns the algebraic outer
product of the two given arrays
//as (algebraic) vectors:p[i][j] =
a[i]*b[j]
```

21. Write a program that declares an array a of five ints:

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

Write a loop that requests an int and attempts to assign 0 to the corresponding element in a. note. Which values are legal and which give Exceptions?

22. Write a method

```
private static int[] readArray
(Scanner)
```

that returns a full array of the ints in the specified Scanner.

23. What do you think is printed by the statements below? What does Java actually print?

```
int[] a = new int[3];
System.out.println(a.length);
a = new int [5];
System.out.println(a.length);
a = new int[0];
System.out.println(a.length);
```

24. Write a class JPS33 that represents a  $3 \times 3$  integer matrix. Include a constructor

```
public JPS33(int a00, int a01,
int a02,
int a10, int a11, int a12,
int a20, int a21, int a22)
```

that creates the specified JPS33. Include a to String method that returns the three-line representation.

25. Write a method

```
public static boolean
isSquare(int[][])
```

that takes a rectangular matrix and determines if it is square.

## Multiple Choice Questions

- An array name is
  - an array variable
  - a keyword
  - a common name shared by all elements
  - not used in the program
- One-dimensional array is known as
  - vector
  - table
  - an array of arrays
  - matrix
- Array subscripts in Java always starts at
  - 1
  - 0
  - 1
  - Any value
- Maximum number of elements in the array declaration int x[10] is
  - 9
  - 11
  - 10
  - Undefined
- If the size of an array is less than the number of initializers
  - The extra values are neglected

- (b) It is an error  
 (c) The size is automatically increased  
 (d) The size is neglected
6. Missing elements of partly initialized arrays are  
 (a) set to zero (c) not defined  
 (b) set to one (d) invalid
7. Two-dimensional array elements are stored in  
 (a) column major order  
 (b) row major order  
 (c) both (a) and (b)  
 (d) random order
8. Two-dimensional array elements are stored  
 (a) row by row in the subsequent memory locations  
 (b) column by column in the subsequent memory locations  
 (c) row by row in the scattered memory locations  
 (d) column by column in the scattered memory locations
9. To initialize a 5-element array all having value 1 is given by  
 (a) int num[5] = {1};  
 (b) int num[5] = {1, 1, 1, 1, 1};  
 (c) int num [4] = {1,1,1,1,1};  
 (d) int num [ ] = {1};
10. The value within the [] in an array declaration specifies  
 (a) subscript value  
 (b) index value  
 (c) size of array  
 (d) value of the array element
11. In a multidimensional array with initialization  
 (a) the right-most dimension may be omitted  
 (b) the left-most dimension may be omitted  
 (c) nothing must be omitted  
 (d) all may be omitted
12. int arr [3][2][2] = {1,2,3,4,5,6,7,8,9,10,11,12}; What values does arr [2] [1][0] in the sample code above contains?  
 (a) 5 (c) 9  
 (b) 7 (d) 11
13. x[2] = 5; 2[x] = 5  
 Are x [2] and 2[x] identical in the sample code above?  
 (a) No, both variable assignments have invalid syntax.  
 (b) No, x [2] is correct, but 2[x] is an invalid syntax.  
 (c) Yes, both are identical because they resolved to pointer references.  
 (d) No, 2[x] is correct but x [2] is an invalid syntax.
14. Identify the correct declaration  
 (a) int a [10][10];  
 (b) int a [10,10];  
 (c) int a (10)(10);  
 (d) int a (10, 10);
15. Identify the wrong expression given int a[10];  
 (a) a[-1] (c) a[0]  
 (b) a[10] (d) ++a
16. The address of the starting element of an array is  
 (a) represented by subscript of the starting element  
 (b) cannot be specified  
 (c) represented by the array name  
 (d) not used by the compiler
17. Arrays are created using a form of the \_\_\_\_\_ operator.  
 (a) sizeof (c) new  
 (b) instanceof (d) None of the above
18. Java arrays are  
 (a) simple variables  
 (b) objects  
 (c) class  
 (d) none of these
19. In Java, arrays are allocated memory  
 (a) statically  
 (b) dynamically using new  
 (c) by default allocation  
 (d) none of the above

## KEY FOR MULTIPLE CHOICE QUESTIONS

- |       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 1. c  | 2. a  | 3. b  | 4. c  | 5. b  | 6. a  | 7. c  | 8. a  | 9. b  | 10. c |
| 11. b | 12. d | 13. c | 14. a | 15. d | 16. c | 17. c | 18. b | 19. b |       |

# 6

# Functions in Java

## 6.1 INTRODUCTION

A function is a self-contained block of code written once for a particular purpose, but can be used again and again. It is a basic entity for Java programming language. Even the execution of the program starts from the main function. It is the basic building block for modular programming.

Depending on the parameters and return type (discussed later on), functions are classified into four categories:

1. No return type and no arguments/parameters.
2. No return type but arguments/parameters.
3. Return type but no argument/parameters.
4. Return type with argument/parameters.

All the above categories except the third one are discussed in this chapter. The third category is discussed in the chapter titled ‘Classes and Object’.

Various methods have been presented so far for different programs. For example, the methods provided by Math library are `sqrt`, `pow`, `max`, etc. All these functions are library functions or built-in functions, i.e., they are already there in the Java programming language and can be used in different programs. Again, some of these functions belong to one of the categories given above. The limitations of these functions are that their code cannot be known; they can simply be used and no modification can be done to them.

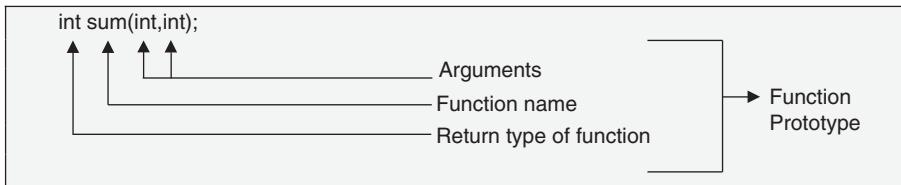
All the built-in functions have their prototype or declaration stored in their respective packages. For example, function declaration of `sqrt` and `pow` is stored in the package `Java.lang`. Particular functions can be written depending on the requirements which are called user-defined functions as they are defined by the user and put in any of the categories given above.

The important point to note here is that `main()` is not a library function. It is simply a restriction from Java compiler that one has to use this function for programming as execution starts from this function. So this can be put into the user-defined function. The discussion in this chapter will entirely be based on the user-defined functions.

## 6.2 FUNCTION DECLARATION AND DEFINITION

In Java, each function that is used must be declared and defined first. The declaration of the function is also termed as ‘prototyping’. A function prototyping tells the compiler three things about a function: function name, return type of the function, and number of arguments.

For example, a function `sum` which takes two arguments of type `int` and return an `int` can be written as follows:



The above declaration tells the compiler that `sum` is a function which accepts two parameters of type `int` and returns type of the function `int`. In Java, as function has to be defined at the place, its declaration and definition (discussed below) cannot be separated. The basics of functions can be understood using a small example.

```
class JPS
{
 static void show()
 {
 System.out.println("Hello from show");
 }
 public static void main(String[] args)
 {
 show();
 }
}
```

The line `static void show ()` is the function name or prototype of the function. The general syntax is as follows:

```
void function_name() { }
```

In the above context, `void` is the return type, `show` is the function name and `()` after function `show` indicates that this `show` function does not take any arguments. Whenever there is `()` after any name then that is a function. Obviously, you may have arguments within `()`. Here, `void` means function, which neither takes any argument nor returns any value. The `static` keyword is a must as objects are not being made use of here. As `main` is the `static` function and all coding can only be executed from the `main`, it can be called `show` from `main`. Now `main` being a `static` function, only other functions which are `static` can be called `main`.

The actual working of the function is done by the definition of the function which is given between the pair of braces.

```
{
System.out.println("Hello from show");
}
```

All the statements within `{}` of the function is called the body of the function. When `show()` is written, it means the function is called `show()`. Whenever `show()` statement is encountered, the control is transferred to the body of the function and all the statements written within it gets executed. When closing brace of the function is encountered, the function is returned to the next statement from where the function was called. In Java, every function will be called from some other function. Here, `show()` has been called within `main()`. So, `main()` is called the calling function and `show()` is called the called/callee function.

```
/*PROG 6.1 WORKING WITH TWO FUNCTIONS */

class JPS1{
 static void show()
 {
 System.out.println("Hello from show");
 }
 static void disp()
 {
 System.out.println("Hello from disp");
 }
 public static void main(String[] args)
 {
 System.out.println("In main");
 show();
 disp();
 System.out.println("Back in main");
 }
}
OUTPUT:
In main
Hello from show
Hello from disp
Back in main
```

*Explanation:* Here, we are working with two functions. As is clear from the output, initially first `System.out.println` in `main()` is executed and then `show()` is called and `System.out.println` within `show` is executed. When `show()` returns `disp()` is called and `System.out.println` within `disp()` is executed. When `disp()` is returned, the second `System.out.println` in `main()` is executed and at the end, the program terminates.

### 6.3 NO RETURN TYPE BUT ARGUMENTS

The general syntax of this category of function is as follows:

```
void function_name(data_type args,.....)
{
 Statements;
 Statements;
 ;
 ;
}
```

The parameters are passed to function when the function is called; they are passed from calling function to the called function. In called function context, the parameters are known as formal parameters and in calling function context, they are known as actual parameters.

```
/*PROG 6.2 DISPLAY OF INTEGER THROUGH FUNCTION VER 2 */

import java.io.*;
import java.util.*;
class JPS2
{
 static void show(int x)
 {
 System.out.println("You entered := " + (x++));
 }
 public static void main(String[] args)
 {
 Scanner sc = new Scanner(System.in);
 int a;
 System.out.print("\n\nEnter a number :=");
 a = sc.nextInt();
 show(a);
 System.out.println("After function a := " + a);
 }
}

OUTPUT:
Enter a number :=123
You entered := 123
After function a := 123
```

*Explanation:* The definition of the function void show(int) states that function show does not return any value but accepts an argument/parameters of int type, i.e., int variable or an int constant will be passed to this function. When the function is called, an int value is passed as shown by the statement show(a). Here, the function is called show and passing ‘a’ as an argument. This is known as call by value as the function and passing value of the variable are called ‘a’. The ‘a’ value sent must be collected in some variable in the function definition. This value is collected in variable ‘x’. Note that the type and number of arguments must match when defining the function. Variable ‘a’ in the function main is called the actual argument and the variable ‘x’ in the function show is called the formal argument. When ‘a’ is passed to the function show a copy of ‘a’ is sent which is collected in ‘x’. In the function, the value ‘x’ is simply printed. The change in ‘x’ in function show does not affect the original value of ‘a’.

**Note:** All primitive data constant or variables are always passed by value and object is always passed by reference.

```
/*PROG 6.3 PRINT FIBONACCI SERIES USING FUNCTION */

import java.io.*;
import java.util.*;
class JPS5
{
 public static void main(String[] args)
 {
 int num;
```

```

Scanner sc = new Scanner(System.in);
System.out.println("\nEnter an integer");
num = sc.nextInt();
System.out.println("\nFibonacci series is\n");
fibbo(num);

}

static void fibbo(int n)
{
 int i, a = 0, b = 1, c;
 System.out.print(" " + a);
 System.out.print(" " + b);
 for (i = 1; i <= n - 2; i++)
 {
 c = a + b;
 System.out.print(" " + c);
 a = b;
 b = c;
 }
}
}

OUTPUT:
Enter an integer
9
Fibonacci series is
0 1 1 2 3 5 8 13 21

```

*Explanation:* The Fibonacci series is the series where each term of the series is given by the following formula:

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n >= 2$$

That is, each term of the series is the sum of its previous two terms. The first two terms are assumed to be 0 and 1.

The number of terms to be displayed is taken in the variable num from the user. This is passed into the function fibbo. The first two terms are displayed as it. The rest of the terms are calculated within the loop. For each new term the sum of the two previous terms are first found, e.g., a and b into c. The value of b into a is then assigned and the new calculated value of c into b. The loop runs for  $n - 2$  times as the first two values 0 and 1 are displayed outside the loop.

```
/*PROG 6.4 DISPLAY DECORATED NAME USING FUNCTION */
```

```

import java.io.*;
import java.util.*;
class JPS6
{
 public static void main(String[] args)
 {
 String name;
 Scanner sc = new Scanner(System.in);
 System.out.println("\n\nEnter your name");

```

```

 name = sc.next();
 line(25, '+');
 System.out.println("\n\n\t\tWelcome " + name);
 line(25, '*');
 System.out.println("\n\n");
 }
 static void line(int n, char p)
 {
 int i;
 System.out.println();
 for (i = 1; i <= n; i++)
 System.out.print(" " + p);
 }
}

OUTPUT:

Enter your name
Hari
+ + + + + + + + + + + + + + + + + + + +
 Welcome Hari
* *

```

*Explanation:* The function `line` accepts two parameters; the first one is an integer and the second is a character. The function prints the second argument, the first number of times, i.e., in the above program when function `line (25, '+')` is called, it first leaves a blank line and then using for loop prints '+' 25 times. Depending on the requirement, any value for the parameters may be put.

### 6.3.1 Returning Prematurely from Function

When the function is called, the code contained within the function is executed and closing braces of the function is encountered and the function execution comes to end. This is the normal flow of execution during function call. In some cases, when we want to return from function early without completing the whole of its execution, we may use `return` statement. This may be on the basis of certain decision making statements like `if` or some loop. To come out early from a method, we can use `return` statement in the following way:

```

void demo()
{
 statements;
 statements;
 statements;
 if(condition)
 return;
 statements;
 statements;
}

```

An example is given below.

```

/*PROG 6.5 RETURNING PREMATURELY FROM FUNCTION */

class JPS7
{
 static void demo(int x)
 {
 if (x < 0)
 {
 System.out.println("No negative value");
 return;
 }
 System.out.println("\n\nHello from demo\n");
 }
 public static void main(String[] args)
 {
 demo(30);
 demo(-30);
 }
}

```

**OUTPUT:**

```

Hello from demo
No negative value

```

*Explanation:* In the method ‘demo’ when a negative value is passed in the function, it returns prematurely by printing “No negative value” otherwise, it prints “Hello from demo” and returns maturely.

## 6.4 FUNCTION WITH PARAMETERS AND RETURN TYPE

The general syntax of this category of function is as follows:

```

return_type function_name(data_type arg,)
{
 statements;
 statements;
 ;
 return data;
}

```

where type of data must be `return_type`.

The parameters are passed on to the function. Inside the function, any processing or computation may be carried out onto the formal parameters along with some local variables, but in the end, the function must return a value of type `return_type`. That value replaces this function call in the `callee` function.

```

/*PROG 6.6 COMPUTE SQUARE OF FUNCTION AND RETURN USING
FUNCTION */

```

```

import java.io.*;
import java.util.*;
class JPS8
{

```

```

public static void main(String[] args)
{
 Scanner sc=new Scanner(System.in);
 double num, s;
 System.out.println("\nEnter a number");
 num=sc.nextDouble();
 s=sqr(num);
 System.out.println("\nSquare of "+num+" is "+s);
}
static double sqr(double x)
{
 double t;
 t = x * x;
 return t;
}
OUTPUT:
Enter a number
12
Square of 12.0 is 144.0

```

**Explanation:** The function `double sqr(int)` tells the compiler that `sqr` is a function which accepts an argument of type `double`, and the `double` before function name `sqr` represents return type of function `sqr`, i.e., a `double` value will be returned from function `sqr` through return statement. In the definition/body of the function, `x*x` is calculated into `t` and the value of `t` is returned through statement `return t`. When this happens, the `t` returns at the place from where the function `sqr` was called, so the statement `s = sqr (num)` becomes `s = t`. If the value of `x` happens to be 12, `t` will have 144.0 when it returns and the same will be assigned to `t`.

Alternative function definition may be written as:

```

double sqr(double x)
{
 return x*x;
}

```

|                                                 |
|-------------------------------------------------|
| /*PROG 6.7 FINDING THE FACTORIAL OF A NUMBER */ |
|-------------------------------------------------|

```

import java.io.*;
import java.util.*;
class JPS10
{
 public static void main(String[] args)
 {
 Scanner sc = new Scanner(System.in);
 int num, s;
 System.out.println("\nEnter a number");
 num = sc.nextInt();
 s = fact(num);
 System.out.println("Factorial of "+num+" is "+s);
 }
 static int fact(int x)

```

```

 {
 int f = 1, i;
 for (i = 1; i <= x; i++)
 f = f * i;
 return f;
 }
}

OUTPUT:
Enter a number
7
Factorial of 7 is 5040

```

**Explanation:** The factorial of a number  $x$  is the product of  $1*2*3*4\dots*x$ . For example, factorial of 4 is  $1*2*3*4$ , i.e., 24. The number whose factorial is to be found is passed into the function fact and is collected into  $x$ . We take a variable  $f$  into which factorial of  $x$  will be stored. This is initialized to 1. In the function fact, a for loop is run from 1 to  $x$ . We show the steps for value of  $x = 4$ .

|       |     |             |                |
|-------|-----|-------------|----------------|
| Step1 | i=1 | $f = f * i$ | $= 1 * 1 = 1$  |
| Step2 | i=2 | $f = f * i$ | $= 1 * 2 = 2$  |
| Step3 | i=3 | $f = f * i$ | $= 2 * 3 = 6$  |
| Step4 | i=4 | $f = f * i$ | $= 6 * 4 = 24$ |

```
/* PROG 6.8 MAXIMUM OF THREE NUMBERS FROM A FUNCTION WHICH
FIND MAXIMUM OF TWO NUMBERS */
```

```

import java.io.*;
import java.util.*;
class JPS11
{
 public static void main(String[] args)
 {
 int m, a, b, c;
 Scanner sc = new Scanner(System.in);
 System.out.println("\nEnter the three numbers \n");
 a = sc.nextInt();
 b = sc.nextInt();
 c = sc.nextInt();
 m = max(a, max(b, c));
 System.out.println("\na= "+a+"\tb= "+b+"\tc= "+c);
 System.out.println("\nMaximum of three numbers:= "+m);
 }
 static int max(int x, int y)
 {
 return x > y ? x : y;
 }
}

OUTPUT:
Enter the three numbers
123 456 789
a= 123 b= 456 c= 789
Maximum of three numbers: = 789

```

*Explanation:* In the expression  $m = \max(a, \max(b, c))$  the first  $\max(b, c)$  is called and when this returns, the call  $\max(b, c)$  is replaced by the maximum of two, either b or c, say t (assume). Then again, function max is called which returns max of a and t. Note how the maximum of three numbers is calculated with a function which finds maximum of two numbers only.

On the similar basis, max of 4 numbers can be calculated by writing the following:

```
m = max (max (a, b), max (c, d));
```

## 6.5 RECURSION

Recursion is a programming technique in which a function calls itself for a number of times until a particular condition is satisfied. It is very important to understand and once understood many long listing of code can be reduced to a few number of lines. Recursion is basically a word mostly used in mathematics to state a new term within previous term, such as the following:

$$X^{n+1} = X^n + 1 \text{ for } n \geq 1 \text{ and } X^1 = 1$$

Here,  $X_2$  can be calculated in terms of  $X_1$ ,  $X_3$  in terms of  $X_2$  and so on.

When solving a problem through recursion, two conditions must be satisfied:

1. The problem must be expressed in recursive manner.
2. There must be a condition which stops the recursion, otherwise there would be a stack overflow.

```
/* PROG 6.9 DEMO OF RECURSION */

class JPS12
{
 static int t;
 public static void main(String[] args)
 {
 if (t == 5)
 {
 System.out.println("\n\nQuit");
 System.exit(0);
 }
 System.out.println("\n Hello from main" + (++t));
 main(args);
 }
}
```

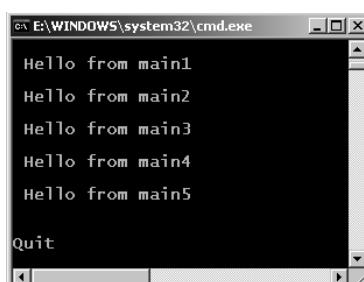


Figure 6.1 Output screen of Program 6.9

**Explanation:** The `t` is a static variable. It retains its previous value even after the function execution is complete. When `main()` executes for the first time, the value of `t` is 0. If the condition fails and `System.out.println` after if executes. When control reaches at `main(args)` the recursion starts as we are calling `main` from the `main`. The `main` starts again for this call with the value of `t = 1`. Note that `static int t;` is written inside the class but not in the `main` function. For value `t=1`, `System.out.println` after if gets executed and `main(args)` is called again and this time with the value of `t=2`. This continues until `t` does not become 5. When `t` becomes 5 and if the condition satisfies, the recursion stops and the program terminates.

If we do not take '`t`' as a static variable in the class but not in the function, then first of all, call program will give compilation error as `main` is static function and inside static function only other static function and static variable can be used. Second, if we try to declare `t` inside the function without a `static` qualifier, the infinite loop will result as when recursion starts (`t` will have to be initialized with 0), the previous value of `t` will not be retained and instead, each time the `main` will be called and `t` will be initialized with 0.

```
/* PROG 6.10 FACTORIAL OF A NUMBER USING RECURSION */

import java.io.*;
import java.util.*;
class JPS13
{
 public static void main(String[] args)
 {
 int ans, num;
 Scanner sc = new Scanner(System.in);
 System.out.println("\nEnter an integer");
 num = sc.nextInt();
 ans = fact(num);
 System.out.println("Factorial is " + ans);
 }
 static int fact(int n)
 {
 return (n < 1 ? 1 : n * fact(n - 1));
 }
}
OUTPUT:
Enter an integer
5
Factorial is 120
```

**Explanation:** Assume `n` is 4, now recursion works as follows:

|   |                                      |
|---|--------------------------------------|
| N | Function name                        |
| 4 | <code>4 * fact(3)</code>             |
| 3 | <code>4 * (3 * fact(2))</code>       |
| 2 | <code>4 * (3 * (2 * fact(1)))</code> |
| 1 | <code>4 * (3 * (2 * 1))</code>       |

When recursion starts, each call to function `fact` creates a new set of variables here only one. Whenever recursion starts, the recursive function calls do not execute immediately (in reality function addresses are put). They are saved inside the stack along with the value of variables. (A stack is a data structure which

grows upward from max\_limit to 1. Each new item ‘pushed’ in the stack takes its place above previously entered item. The items are ‘popped’ out in the reverse order in which they were entered, i.e., the last item is popped out first. This being the main reason that they are called LIFO (last in first out). This process is called winding in the recursion context. When recursion is stopped in the above program and when n becomes 1 and the function returns the value 1, all the function calls saved inside the stack are popped out from the stack in the reverse order and get executed, i.e., fact (1) returns to fact (2), fact (2) return to fact (3) and at the end, fact (3) returns to fact (4) which ultimately returns to the main. This process is called unwinding.

## 6.6 FUNCTION OVERLOADING

Overloading refers to the use of same function for different purposes. Function overloading refers to creating number of functions with the same name which performs different tasks. Function overloading is also known as function polymorphism. Function overloading relieves us from remembering so many function names with the type of arguments they take. In function overloading, we can create a number of functions with the same name but the number of arguments or type of arguments must be different. A few examples are as follows:

```
1. int sum(int);
 float sum(float);
 double sum(double);
```

In this example, three functions are declared with the same name `sum`. Each function takes just one parameter but all are of different types, i.e., the number of parameters in all overloaded function `sum` is the same but the type of parameter is different.

```
2. void show(int, int);
 void show(int);
 void show(int, int, int);
```

In this example, the number of parameters is not the same in the `show` functions but the type is the same.

```
3. void show(int, char);
 void show(char, int, float);
 void show(int);
 int show(int, int);
 float show(char, char, char);
```

In this example, there is a mix of overloaded `show` functions. Some of them have same number of arguments but type is different, while some others have same type of arguments but numbers of arguments are different.

When there are a number of overloaded functions in a program, in which function to call is determined by either checking type of argument or number of argument, note that the return type does not play any role in function overloading, because the which function to call is determined by checking the type and number of argument a function accepts. When control is transferred to function and function is about to return after execution, return type comes to play.

In function overloading the compiler first tries to find an exact match. If the exact match is not found the integral promotion is used. The user-defined conversion methods may also be used in case a class object is to be converted to any built-in type or vice versa.

```
/* PROG 6.11 DEMO OF FUNCTION OVERLOADING ver 1*/
import java.io.*;
import java.util.*;
class JPS15
{
 public static void main(String[] args)
 {
 show('h');
 show(15);
 show(34.67f);
 show("NMIMS University");
 }
 static void show(int x)
 {
 System.out.println("int x := "+x);
 }
 static void show(float x)
 {
 System.out.println("float x := "+x);
 }
 static void show(char x)
 {
 System.out.println("char x := "+x);
 }
 static void show(String x)
 {
 System.out.println("String x := "+x);
 }
}
OUTPUT
char x := h
int x := 15
float x := 34.67
String x := NMIMS University
```

*Explanation:* In the program, the function `show` is overloaded four times. The function takes a single parameter but each parameter is different in each function as can be seen in the program. In the `main` function, `show` is called with four literals of type `int`, `char`, `float` and `String`. Each parameter is passed to `show` and `show` is called four times. The compiler depending on the type of argument to function `show` calls the respective version of the `show` function, i.e., in case of `show(15)`, the `show` function of `int` version will be called and so on.

```
/*PROG 6.12 DEMO OF FUNCTION OVERLOADING VER 2, IMPLICIT PROMOTION */
```

```
import java.io.*;
import java.util.*;
class JPS16
{
 public static void main(String[] args)
```

```

{
 show(25);
 show('h');
 show(2.5f);
 show(4.56);
}
static void show(int x)
{
 System.out.println("\nint x := " + x);
}
static void show(double x)
{
 System.out.println("\ndouble x := " + x);
}
}

OUTPUT:
int x := 25
int x := 104
double x := 2.5
double x := 4.56

```

*Explanation:* In the program, the function `show` is overloaded which takes a single parameter of `int` and `double`. In the function call `show(25)`, `int` version of `show` is called as 25 is an integer. In function call `show('h')`, `int` version of `show` is called why? The compiler searches for an overloaded `show` function that takes a parameter of type `char`, but fails as there is no such overloaded function written. So it does an integral promotion from `char` to an integer and calls the `int` version of `show` function displaying ASCII value of '`h`'. Similarly, in function call `show(2.5f)`, the compiler looks for an overloaded function `show` which takes `float` type argument and if fails the overloaded function `show` with `double` type value is called after doing promotion of `float` to `double`.

```

/* PROG 6.13 FUNCTION OVERLOADING MAX OF TWO NUMBERS */

import java.io.*;
import java.util.*;
class JPS19
{
 public static void main(String[] args)
 {
 int x, y, intmax;
 float p, q, fmax;
 Scanner sc = new Scanner(System.in);
 System.out.println("Enter two integers");
 x = sc.nextInt(); y = sc.nextInt();
 System.out.println("Enter the two floats");
 p = sc.nextFloat(); q = sc.nextFloat();
 intmax = max2(x, y);
 fmax = max2(p, q);
 System.out.println("Max of two int is:=" + intmax);
 }
}

```

```

 System.out.println("Max of two float is:=" + fmax);
 }
 static int max2(int x, int y)
 {
 return (x > y ? x : y);
 }
 static float max2(float x, float y)
 {
 return (x > y ? x : y);
 }
}

OUTPUT:

Enter two integers
12 56
Enter the two floats
12.34
56.78
Max of two int is := 56
Max of two float is := 56.78

```

*Explanation:* In the program, there are two overloaded version of function `max2`, which takes two parameters of type `int` and `float` and returns the maximum of two numbers. Depending on the type of parameter, the appropriate version of `max2` function is called.

## 6.7 PONDERABLE POINTS

1. Based on the nature of creation there are two categories of functions: built-in and user defined.
2. The functions that are predefined and supplied along with the compiler are known as built-in functions.
3. The function `main()` can appear only once in a Java program, because execution commences from the first statement in the `main()` function. If there is more than one `main()` function, there will be a confusion while commencing the execution.
4. A function that perform no action is known as a dummy function. It is a valid function. Dummy function may be used as a place-holder, which facilitates adding new functions later on. For example:  
`void dummy () {}`
5. Advantages of recursion include the following:
  - (a) Easy understanding of the program logic.
  - (b) Helpful in implementing recursively defined data structure.
  - (c) Compact code can be written.
6. Use of return statement helps in early exiting from a function apart from returning a value from it.
7. Function prototyping, declaration or signature are all the same thing.
8. Function declaration tells three things to the compiler:
  - (a) Function name
  - (b) Type and number of argument it takes
  - (c) Return type of the function
9. Function overloading is the creation of a number of functions with the same name but differing either in type of argument or number of arguments.
10. There is no limit on the number of functions that can be overloaded in a program.
11. Return type does not serve any purpose in function overloading.

## REVIEW QUESTIONS

1. Write the definition of a function. Indicate the types of functions available in Java.
2. Differentiate between library and the user defined function.
3. How does a function work? Explain how arguments are passed and results are returned.
4. What are actual and formal arguments? Explain with suitable examples.
5. What are void functions?
6. Why is return statement not necessary when function is called by reference?
7. What is recursion? Explain its advantages.
8. Explain the types of recursion.
9. Is it possible to call library function recursively?
10. Does the function prototype match with function definition?
11. What is the main ( ) in Java? Why is it necessary in each program?
12. Write a program to compute the total amount payable with annual interest for a given number of years. The inputs are principal amount, rate of interest and number of years. Attempt the program with or without recursion.
13. Write a program to count for how many times a function called. Create a user defined function.
14. Write a Java program using a function roots to calculate and display all roots of the quadratic  $AX^2 + BX + C = 0$ .
15. Write a Java program which displays word equivalent of a number of only 3 digits, that is, 123 should be displayed as one hundred twenty-three.
16. Write a function that reads a number in binary form and converts it into hexadecimal form. Hexadecimal values must be stored also.
17. Write a program to calculate average temperature of five days. Create temp () function.
18. Write a program to add return value of three functions.
19. Consider the following recursive method JPS
 

```
private static int JPS(int n)
{
 if(n == 0)
 return 1;
 return 3*JPS(n-1)+1;
}
```

 Which corresponds to the recursive series:  
 $JPS(0) = 1$   
 $JPS(n) = 3 \cdot JPS(n-1) + 1$ 

Write a driver that prints out the first 10 values of the series. What is the closed form of JPS?
20. Consider the Fibonacci function:  
 $Fib(0) = 1$   
 $Fib(1) = 1$   
 $Fib(n) = Fib(n-2) + Fib(n-1)$ 

Use hand-tracing to compute the following values:

  - a. Fib(3)
  - b. Fib(4)
  - c. Fib(5)

Notice how much extra work is required to compute these values because we need to compute the same value, for example, Fib (2) many times.
21. Using the recursive definition of the Fibonacci sequence, what is the highest term that your computer system can comfortably compute? Use the long data type so that you can avoid integer wraparound as long as possible.
22. Consider the following method:  
`public static int JPS(int n)
{
 if(n==0) return 0;
 return n + JPS(n-1);
}`

What does this method compute?

## Multiple Choice Questions

1. How many main () function can be defined in a Java program?
 

|       |                         |
|-------|-------------------------|
| (a) 1 | (c) 3                   |
| (b) 2 | (d) Any number of times |
2. A function with no action
 

|                            |
|----------------------------|
| (a) is an invalid function |
|----------------------------|
3. (b) produces syntax error  
 (c) is allowed and is known as dummy function  
 (d) none of the above
3. The default return data type in function definition is
 

|          |            |
|----------|------------|
| (a) int  | (c) float  |
| (b) char | (d) double |

# KEY FOR MULTIPLE CHOICE QUESTIONS

1. a

2. c

3. a

4. b

5. c

6. a

7. a

8. b

9. b

10. d

11. a

12. c

13. d

14. d

15. a

# Classes and Objects

7

## 7.1 INTRODUCTION

A class is a basic unit of encapsulation and abstraction. It binds together data and methods that work on data. The class is an abstract data type (ADT) so the creation of a class simply creates a template. As discussed in the beginning of the Chapter 1, each and everything has to be defined within the class. The data members of the class are called field and the member functions are known as methods. The general syntax of creating a class in Java is shown as below.

```
class Class_name
{
 type member_variable1;
 type member_variable2;
 //.....
 type member_variable3;
 type methodname1(parameter-list)
 {
 //body of method
 }
 type methodname2(parameter-list)
 {
 //body of method
 }
 type methodnameN(parameter-list)
 {
 //body of method
 }
}
```

The `class` is a keyword. Following this keyword, `class.class_name` represents the name of the class. The class name must obey the rules of writing an identifier as the `class_name` is nothing but an identifier. The class is opened by opening brace (`{`) and closed by closing brace (`}`). Inside the class, data member and member functions are defined. The type/mode may be public, private or protected. If no type is specified then default type is assumed, which is known as friendly default. There are three different types of modes: public, private and protected. The mode is also known as visibility modifier or access specifier. All public data members and methods declared as public/protected can be used inside and outside the class. Private data members and functions can be used only inside the class. To know more about access specifier, see the chapter ‘Package and Interface’.

The data members of the class are used inside the member functions of the class. The data members are declared inside the class but used in the member functions of the class. Usually the data members are private and the member functions are public. So the data members can be used only inside the functions of the class. This is to safeguard private data from external access.

If no visibility mode is written, the default visibility mode is assumed for the class.

The declaration of a class alone does not serve any purpose. To make use of the class a variable of type class needs to be created. A variable of class type is known as an object. The class is loaded into memory when the first object of the class is created. The type of class defines the nature and shape of the object. The creation of object creates the memory space for it which depends on the size of the data members of the class. For each object, a separate copy of the data members is created. On the other hand, only one copy of the member function is created, which is shared by all the objects. The objects are also termed as instances of the class.

The objects call the member functions of the class using operator ‘.’ which is known as period or membership operator.

Before a member function can work on to the data member of the class, they must be initialized by calling a function that provides initial values to the data member of the class.

Consider the dummy class as given below.

```
class demo
{
}
```

This creates an empty class named `demo`. To create an object of the class `demo` the following is written:

```
demo d;
d= new demo ();
```

In Java, the line `demo d;` creates only a reference for an object of class `demo` type. This does not create object at all. In Java, no object is created statically. All objects are created dynamically using the `new` operator and constructor method of the class. (A constructor is a member function of the class whose name is the name of the class). The second line `d = new demo();` actually creates an object from the heap and the object `d` starts pointing to the newly created object of class `demo` type. This can be shown in Figure 7.1 (a–b).



**Figure 7.1** (a) Reference ‘d’ pointing no where (b) ‘d’ contains reference of objects

A new class with fields (data members) and methods (member functions) is given below:

```
class demo
{
 int num;
 void input(int x)
 {
 num = x;
 }
 void show()
 {
 System.out.println("num=" + num);
 }
}
```

In Java, the default visibility modifier is default public or friendly public. It is not pure public. The meaning of this is explained in detail in the chapter entitled ‘Package and Interface’. But for the time being, assume it is similar to public. Therefore, everything in the class is public. All fields of the class are known as instance variable and the methods are instance method. An object of a class is usually termed as instance of a class. In Java, each method or field must be separately assigned an access specifier. For example, in order to make num private and show protected in the above class, the following syntax can be written:

```
private int num;
protected void show() {
 System.out.println("num=" + num);
}
```

Every method has to be defined within the class. A method cannot be declared inside a class and definition outside the class. The methods in Java determine the messages an object can receive. For example, one can call the method show as assuming d is an object of class demo and initialize d.show(). This act of calling a method is commonly referred to as sending a message to an object. In the preceding example, the message is show () and the object is d. Object-oriented programming is often summarized as simply ‘sending messages to objects’. For each object a separate storage is allocated depending on the size of the fields within the class. A field is an object of any type that can be communicated with by its reference. The fields may be primitive data types or references to objects of other classes; in case of reference, fields may be primitive data types before it can be put to use. The primitive data types as members of the class posses default values if not initialized. They are listed in Table 7.1.

| S. No. | Primitive Type | Default Value |
|--------|----------------|---------------|
| 1      | boolean        | False         |
| 2      | char           | (null)        |
| 3      | byte           | (byte)0       |
| 4      | short          | (short)0      |
| 5      | int            | 0             |
| 6      | long           | 0L            |
| 7      | float          | 0.0f          |
| 8      | double         | 0.0d          |

**Table 7.1** Default values of data types for class

For char, the default value is not printed, rather a blank space will be visible. See the following program.

```
class demo
{
 int x;
 float y;
 char z;
 double d;
 boolean b;
}
class JPS1
{
 public static void main(String[] args)
 {
 demo d = new demo();
 System.out.println(d.x + " " + d.y + " " + d.z);
 System.out.println(d.d + " " + d.b);
 }
}
OUTPUT:
0 0.0
0.0 false
```

To access any of the field or method the dot operator can be used. For example, to access num and show in the earlier class example the following can be written:

```
demo dd=new demo ();
dd.num=20;
dd.show();
```

## 7.2 PROGRAMMING EXAMPLES

```
/*PROG 7.1 DEMO OF CLASS VER 1 */

class demo
{
 private int cx, cy;
 void input(int x, int y)
 {
 cx = x;
 cy = y;
 }
 void show()
 {
 System.out.println("cx=" + cx);
 System.out.println("cy=" + cy);
 }
}
class JPS2
{
 public static void main(String args[])
 {
 demo d = new demo();
 d.input(10, 20);
 d.show();
 }
}

OUTPUT:
cx=10
cy=201
```

*Explanation:* A class is created using the keyword `class` followed by the class name. The class name follows the rules of writing identifiers. Inside the class `demo`, there are two variables `cx` and `cy` of type `integer`. Before the `cx` and `cy` is written ‘private’ so they become private. If `private` is not written, they are considered as default `public`. The function `input_data` takes two parameters of type `int` and returns nothing as `void` in the return type. The function `show_data` displays the two data members’ `cx` and `cy`.

In the main, the statement `demo d = new demo();` creates an object of the class `demo` type. The variable `d` of class `demo` type is known as an instance of the class `demo` or an object. The creation of an object allocates memory depending on the size of the class which is the sum of the size of the data member. The functions do not add to the size of the class or object. Here, a memory of 8 bytes are allocated for the object `d`. The functions are given memory when a class is loaded into the memory.

All public members (data + function) can be accessed in the `main` with the help of object using dot (.) operator, which is also known as membership operator or period.

Through statement `d1.input_data(10, 20)` the function `input_data` is called and two `int` constants, 10 and 20, are passed. They are collected in the formal parameters `x` and `y` and then assigned to `cx` and `cy`. Now, `cx` and `cy` can be used in all the functions of class `demo`. While working with the class and object, the first thing that needs to be performed is to assign values to data members of the class through function or they can be initialized within the class itself. As `cx` and `cy` now contain the valid values, statement `d1.show_data()` displays the values of `cx` and `cy` through `System.out.println`. Note that as `cx` and `cy` are private they can be accessed only inside the member function of the class and not anywhere. Usually, data members are made as private and member function as public while designing a class.

```
/*PROG 7.2 DEMO OF CLASS VER 2, INITIALIZING DATA MEMBERS
WITHIN THE CLASS */

import java.io.*;
import java.util.*;
class demo
{
 private int cx = 20, cy = 40;
 void show()
 {
 System.out.println("cx = "+cx+"\tcy = "+cy);
 }
}
class JPS3
{
 public static void main(String args[])
 {
 demo d = new demo();
 d.show();
 demo d1 = new demo();
 d1.show();
 }
}

OUTPUT:
cx=10
cy=20
```

*Explanation:* The program is similar to the previous one with the difference that the data members of the class are initialized within class itself and created two objects instead of one. The disadvantage of initializing data members within the class is that all the objects share the same initialized value. In the earlier program, each object can initialize its own data members using `call to` function.

If cx and cy of the data members were not **private** then their values could be changed outside the class in the following way:

```
d.cx = 10; d.cy=50;
d1.c x = 50; d1.cy = 100;
```

```
/*PROG 7.3 DEMO OF CLASS VER 5 */

import java.io.*;
import java.util.*;
class person
{
 private int age;
 private String name;
 void input()
 {
 Scanner sc = new Scanner(System.in);
 System.out.println("Enter the age of person");
 age = sc.nextInt();
 System.out.println("Enter the name");
 name = sc.next();
 }
 void show()
 {
 System.out.println("\n++++++++++++++");
 System.out.println("\tPerson Details");
 System.out.println("\n*****");
 System.out.println("Name=" + name);
 System.out.println("Age=" + age);
 }
}
class JPS5
{
 public static void main(String args[])
 {
 person p = new person();
 System.out.println("\n\n*****");
 p.input();
 p.show();
 System.out.println("\n*****");
 }
}
OUTPUT:
*****+
Enter the age of person
26
Enter the name
Hari
+++++
Person Details

Name=Hari
Age=26
*****+
```

*Explanation:* The program is similar to the previous program but instead of providing constant values to the data members of the class input is taken from the console. Note that, several examples of taking input from console have already been discussed in the earlier chapters.

```
/*PROG 7.4 READING AND WRITING STUDENT DATA */

import java.io.*;
import java.util.*;
class student{
 private String sname;
 private int sage;
 private String sclass;
 void getdata(String sn, int sa, String sc)
 {
 sname = sn;
 sage = sa;
 sclass = sc;
 }
 void showdata()
 {
 System.out.println("Name :=\t" + sname);
 System.out.println("Age :=\t" + sage);
 System.out.println("Class:=\t" + sclass);
 }
}
class JPS6{
 public static void main(String args[]){
 student s1, s2;
 String n, c;
 int a;
 Scanner sc = new Scanner(System.in);
 s1 = new student();
 s2 = new student();
 System.out.println("Enter data for student-1\n");
 n = sc.next();
 a = sc.nextInt();
 c = sc.next();
 s1.getdata(n, a, c);
 System.out.println("Enter data for student-2\n");
 n = sc.next();
 a = sc.nextInt();
 c = sc.next();
 s2.getdata(n, a, c);
 System.out.println("\n=====");
 System.out.println("\tSTUDENT-1");
 System.out.println("=====\n");
 s1.showdata();
 System.out.println("\n+++++++\n");
 System.out.println("\tSTUDENT-2");
 }
}
```

```

 System.out.println("*****\n");
 s2.showdata();
 System.out.println("+++++\n");
 }
}

OUTPUT:

Enter data for student-1
Hari 25 M.Tech(CSE)
Enter data for student-2
Ranjana 21 PGDCA
=====
STUDENT-1
=====
Name := Hari
Age := 25
Class:= M.Tech (CSE)
+++++
STUDENT-2

Name := Ranjana
Age := 21
Class:= PGDCA
+++++

```

*Explanation:* In the student class, there are three data members namely, `sname`, `sage`, `sclass` and two member functions. One function is for inputting the data and another for displaying the data. In the `main`, two objects `s1` and `s2` are created. Two separate memory blocks for data members for each of the objects `s1` and `s2` are allocated and both have their different copy of each data members. We input values for these data members in local variables and call functions `getdata` and `showdata` for both the objects.

### 7.3 ACCESSING PRIVATE DATA

Data is declared as private so that it is only accessible inside the functions of the class and other external functions cannot use the private data. But sometimes it is required that the private data is accessible outside the class and for this, the one way is to make them public, but this defies the principle of data hiding. The other solution is to create public member functions that return the private data members. This is the way which is usually employed for accessing private data. Similarly, for private data one can have private member functions. But private member functions can be used only inside the other member functions of the class as they cannot be called from outside the class.

The following example is given to understand how to access private data.

```
/*PROG 7.5 ACCESSING PRIVATE DATA OUTSIDE THE CLASS VER 1*/
```

```

import java.io.*;
import java.util.*;
class demo
{
 private int num;

```

```

void fun()
{
 num = 30;
}
int getnum()
{
 return num;
}
class JPS8
{
 public static void main(String args[])
 {
 demo d = new demo();
 d.fun();
 System.out.println("Num is:= " + d.getnum());
 }
}
OUTPUT:
Num is:= 30

```

*Explanation:* In the program, a class demo has a private data member num. In the main, when function fun is called by an object d of class demo, the num is initialized to 30. In order to access this value of num outside the class, a public member function getnum() is written which returns the value of num. As the type of num is int the written type of function getnum is kept as int. In the main when this function is called as d.getnum() num is returned from the function getnum. Thus, private data member has been accessed outside the class.

```

/*PROG 7.6 ACCESSING PRIVATE DATA OUTSIDE THE CLASS VER 2 */

class Person
{
 private String name;
 private char sex;
 private float sal;
 void input(String n, char s, float f)
 {
 name = n;
 sex = s;
 sal = f;
 }
 String getname()
 {
 return name;
 }
 char getsex()
 {
 return sex;
 }
}

```

```

 }
 float getsal()
 {
 return sal;
 }
 }
 class JPS8
 {
 public static void main(String[] args)
 {
 Person p = new Person();
 p.input("Vijay Nath", 'M', 30000f);
 System.out.println("Name :="+p.getname());
 System.out.println("Sex :="+p.getsex());
 System.out.println("Salary :="+p.getsal());
 }
 }
 OUTPUT:
 Name :=Vijay Nath
 Sex :=M
 Salary :=30000.0

```

*Explanation:* In this program, for accessing the name of the private data members and also their sex and salary, three public member functions—`getname`, `getsex` and `getsal`—are written which return the value of name, sex and sal, respectively. Note that depending on the type of data member that is returned from public member functions, the return type is set accordingly.

```
/*PROG 7.7 WORKING WITH MORE THAN ONE OBJECTS */
```

```

class Emp
{
 private String ename;
 private float sal;
 void input(String n, float s)
 {
 ename = n;
 sal = s;
 }
 float getsal()
 {
 return sal;
 }
 String getname()
 {
 return ename;
 }
};
class JPS10
{
 public static void main(String args[])
 {

```

```

 Emp e1 = new Emp();
 e1.input("Hari", 25000);
 Emp e2 = new Emp();
 e2.input("Juhi", 20000);
 if (e1.getsal() > e2.getsal())
 System.out.println("\n\n"+e1.getname() + " 's
 salary is higher");
 else
 System.out.println("\n\n"+e2.getname() + " 's
 salary is higher");
 }
}

OUTPUT:
Hari 's salary is higher

```

*Explanation:* The class Emp has two data fields: ename and sal. In the main, two objects are created and the values to the data members assigned using input method. The class has two public methods which return the values ename and sal of the private field. In the main, salary of the two Emp objects is compared and appropriate message is displayed.

## 7.4 PASSING AND RETURNING OBJECTS

Similar to returning and passing arguments to functions of basic types like int, char, double, float, and char\*, one can pass objects of class to function and even return objects from functions. In order to pass objects, one has to simply write in the declaration of function, the class name whose objects will be passed. Similarly, for returning an object, the class name has to be written as return type. For example, for a function show which takes an object of demo class type and returns an object of demo class, the following declaration is written:

```
demo show (demo);
```

One important thing to note while passing the objects to functions is that objects are never passed to the functions as value. They are always passed by reference, i.e., no new object is created and instead, a new reference is created which points to previously existing object whose reference is passed to the method.

```

/*PROG 7.8 PASSING OBJECT TO METHOD */

class demo
{
 private int num;
 void input(int x)
 {
 num = x;
 }
 void twice(demo d)
 {
 num = 2 * d.num;
 }
 void show()
 {
 System.out.println("num:=" + num);
 }
}

```

```

}
class JPS11
{
 public static void main(String args[])
 {
 demo d1 = new demo();
 demo d2 = new demo();
 d1.input(50);
 d2.twice(d1);
 System.out.println("Object d1");
 d1.show();
 System.out.println("Object d2");
 d2.show();
 }
}

OUTPUT:
Object d1
num: =50
Object d2
num: =100

```

*Explanation:* The method `twice` receives an object of the class `demo` type. In the `main`, two objects `d1` and `d2` are created and a value of 50 to `num` of `d1` is assigned using `input` method. In the next line `d2.twice (d1)`, `d2` calls the method `twice` and sends `d1` as argument. Inside the method, `twice` reference of `d1` is assigned to `d`. Now both `d` and `d1` starts pointing to the same object. Any changes made inside `twice` on `num` data members by `d` are reflected back in `d1`. Inside the method `twice` `num` represents `num` of object `d2` which gets double the value of `d.num` (actually `d1.num`). Later on, the value of `num` for both the objects is displayed.

Now, the function is changed `twice` as given below.

```

void twice(demo d)
{
 num = 2 * d.num;
 d.num = 1000;
}

```

When a function is called as `d2.twice (d1)`; the reference of `d1` is passed to the method `twice`. This reference is stored in the `d`. Now, both `d` and `d1` refer to the same object and therefore any changes performed using either of the references `d` or `d1` actually occur on to the same object. Inside the method, the value of `d.num` changes to 1000. In the `main` when the value of `d1.num` is printed the same value is obtained.

```
/*PROG 7.9 RETURNING OBJECT FROM METHOD */
```

```

class demo
{
 private int num;
 void input(int x)

```

```

{
 num = x;
}
demo twice()
{
 demo temp = new demo();
 temp.num = 2 * num;
 return temp;
}
void show()
{
 System.out.println("num:= " + num);
}
}
class JPS13
{
 public static void main(String args[])
 {
 demo d1 = new demo();
 demo d2 = new demo();
 d1.input(50);
 d2 = d1.twice();
 System.out.println("Object d1");
 d1.show();
 System.out.println("Object d2");
 d2.show();
 }
}
OUTPUT:
Object d1
num: = 50
Object d2
num: = 100

```

*Explanation:* The declaration `demo twice ()` tells the compiler that the function `twice` does not take any argument and returns an object of the type `demo`. Note how the function is defined as given below.

---

```

demo twice()
{
 demo temp = new demo();
 temp.num = 2 * num;
 return temp;
}

```

---

In the line `demo twice ()`, the `demo` is the return type and `twice` specifies that it is a function. Inside the function, a temporary object `temp` is created. Inside this object `temp num` data member,  $2 * \text{num}$  of object `d1`; is assigned. Actually, this is the object which called the function `twice` in the `main` as `d2 = d1.twice()`. When the function `twice` returns the object `temp`, it is assigned to `d2` as `d2 = temp` which copies `num` of `temp` to `num` of `d2`.

```
/*PROG 7.10 SUM OF TWO TIME DURATION */

class time
{
 private int hours;
 private int minutes;
 private int secs;
 void input_time(int hh, int mm, int ss)
 {
 hours = hh;
 minutes = mm;
 secs = ss;
 }
 time sum_time(time A, time B)
 {
 int h, m, s;
 time temp = new time();
 s = A.secs + B.secs;
 m = A.minutes + B.minutes + s / 60;
 h = A.hours + B.hours + m / 60;
 temp.secs = s % 60;
 temp.minutes = m % 60;
 temp.hours = h;
 return temp;
 }
 void show_time(String str)
 {
 System.out.println(str);
 System.out.print("Hours = " + hours + "\t");
 System.out.print("Minutes = " + minutes + "\t");
 System.out.println("Seconds=" + secs);
 }
};
class JPS14
{
 public static void main(String args[])
 {
 time time1, time2, time3;
 time1 = new time();
 time2 = new time();
 time3 = new time();
 time1.input_time(3, 34, 45);
 time2.input_time(4, 26, 55);
 time3 = time3.sum_time(time1, time2);
 System.out.println("\n\n+++++++\n");
 time1.show_time("\n\tTime 1");
 System.out.println("*****\n");
 time2.show_time("\n\tTime 2");
 }
}
```

```

 System.out.println("+++++++\n");
 time3.show_time("Sum of two duration is ");
 System.out.println("*****\n\n");
 }
}

OUTPUT:

+++++++
Time 1
Hours = 3 Minutes = 34 Seconds=45

Time 2
Hours = 4 Minutes = 26 Seconds=55
+++++++
Sum of two duration is
Hours = 8 Minutes = 1 Seconds=40

```

*Explanation:* In the program, the sum of two time durations is found. The function declaration `time sum_time(time A, time B)` tells the compiler that `sum_time` is a function which takes two arguments (objects) of the class `time` type and return an argument (object) of the class `time` type. In the main, hours, minutes and seconds of two objects of the data members are initialized, i.e., `time1` and `time2` by calling the function `input_time`. In the statement,

```
time3 = time3.sum_time(time1, time2);
```

`time3` calls the function `sum_time` and passes `time1` and `time2` as argument. Inside the function `sum_time`, the number of seconds, minutes and hours is first calculated in local variables—`s`, `m` and `h`—and from them values to data members of `temp` objects are actually assigned. In the end, this `temp` object is returned and assigned to `time3` in the `main`.

## 7.5 COPYING OBJECTS

In Java, two objects cannot be copied directly by writing simply the assignment as `d1 = d2`; e.g., for copying object `d2` to `d1`. In Java, when `d2 = d1` is written, it means reference `d1` is assigned to `d2`. That is, `d2` gets the reference of `d1` and both points to the same object and both will be sharing the same data members. The changes done by one will be reflected to the other.

Consider the following code:

```

class demo
{
 public int num;
 void change()
 {
 num = 500;
 }
}

```

```

demo A=new demo();
demo B=new demo();
A.num=100;
B=A;
A.change();
System.out.println("After returning from function");
System.out.println("B.num:="+B.num+"\tA.num:="+A.num);

```

Initially, the value of num for object A is 100. In the statement B = A, all the members of A are not copied to B and instead B is getting a reference to A. This means, whatever A was referring, B will start referring the same object. After the statement B = A, both A and B are the two references for one object. The changes made through either of the references A or B are reflected to the object. The method change is called through the reference A. Inside the method change value of num changes to 500. Later on, when the value of num is printed using A and B, the same value is obtained. The same is the case when the objects are passed to methods. In Java, the objects to methods are always passed by the references as stated earlier.

```
/*PROG 7.11 DEMO OF COPYING OBEJCTS VER 1, DOES NOT WORK */
```

```

import java.io.*;
import java.util.*;
class demo
{
 private String sname;
 void show()
 {
 System.out.println("Name=" + sname);
 }
 void change()
 {
 sname = "Changed TO Pandey";
 }
 void input()
 {
 Scanner sc = new Scanner(System.in);
 System.out.println("Enter the name");
 sname = sc.next();
 }
}
class JPS15
{
 public static void main(String args[])
 {
 demo d1 = new demo();
 demo d2;
 d1.input();
 d1.show();
 d2 = d1;
 d2.change();
 d1.show();
 }
}

```

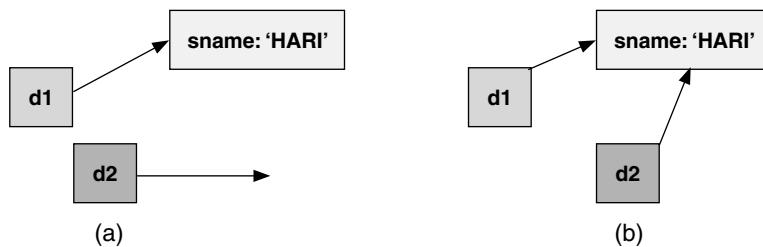
```

}

OUTPUT:
Enter the name
HARI
Name=HARI
Name=Changed TO Pandey

```

*Explanation:* In the program, the object d1 is created and initialized. Assume that the string entered for the data member sname is ‘HARI’. In the statement demo d2, only a reference of the demo class is created. Next d2 = d1 causes d2 to get the reference of d1. Now, both d1 and d2 start pointing to the same object of the demo class. This is as shown in Figure 7.2 (a–b).



**Figure 7.2** (a) Before  $d2 = d1$ ; (b) After  $d2 = d1$ ;

Note that in the program after  $d2 = d1$ , the change function is called using  $d2$  as reference which changes the contents of sname from ‘HARI’ to ‘Change To Pandey’. Next, when the sname is displayed by a call to show using  $d1$ , the same changed name is displayed. In order to really copy the data members, either the object has to be passed to the methods or return objects from the methods. An approach is shown in the next program.

```
/*PROG 7.12 DEMO OF COPYING OBJECTS VER 2, DOES WORK */
```

```

import java.io.*;
class person
{
 protected String sname;
 protected int age;
 protected String profession;
 void show()
 {
 System.out.println("Name =" + sname);
 System.out.println("Age =" + age);
 System.out.println("Profession =" + profession);
 }
 person copy()
 {
 person temp=new person();
 temp.sname=sname;
 temp.age=age;
 temp.profession=profession;
 return temp;
 }
}

```

```

 }
 void input(String sn, int a, String p)
 {
 sname=sn;
 age=a;
 profession=p;
 }
 }
class JPS16
{
 public static void main(String args[])
 {
 person p1 = new person();
 person p2;
 p1.input("Saurabh", 20, "Student");
 p2 = p1.copy();
 System.out.println("Person 1");
 p1.show();
 System.out.println("Person 2");
 p2.show();
 }
}
OUTPUT:
Person 1
Name =Saurabh
Age =20
Profession =Student
Person 2
Name =Saurabh
Age =20
Profession =Student

```

*Explanation:* The method `copy` when called creates a new temporary object and copies all the members of the object who actually called the method. In the end, this object is returned and assigned to `p2`.

## 7.6 ARRAY OF OBJECTS

Similar to the array of any basic data types an array of objects of any class can be created. This becomes handy when one wants to process something like the salary of a number of employees, accounts of persons, records of students, etc. In all these situations, an array of objects makes the work and processing easier and faster. In Java, each object of an array must be initialized first before it can be used. For a class `demo`, an array of objects can be created as `demo [] = new demo [5];`

The above line creates an array of object of size 5. But in reality, it is an array of references of class `demo` type and not an array of objects until they are initialized in the following way:

```

for(int i=0; i<arr.length;i++)
arr[i] = new demo();

```

After initializing, the first array of object is referred to as `arr[0]`, second as `arr[1]` and so on. The data members and methods of objects can be accessed as `arr[i].datamembers` and `arr[i].methodname` where '`i`' is a valid index.

```
/*PROG 7.13 DEMO OF ARRAY OF OBJECT VER 1*/
```

```
import java.io.*;
import java.util.*;
class Point
{
 private int px;
 private int py;
 void input(int x, int y)
 {
 px = x;
 py = y;
 }
 void show()
 {
 System.out.println("(" + px + "," + py + ")");
 }
}
class JPS17
{
 public static void main(String args[])
 {
 Point ptarr[]=new Point[5];
 int i;
 System.out.println("Points are");
 for(i=0;i<ptarr.length;i++)
 {
 ptarr[i]=new Point();
 ptarr[i].input(234+i*2,254+i*3);
 ptarr[i].show();
 }
 }
}

OUTPUT:
Points are
(234,254)
(236,257)
(238,260)
(240,263)
(242,266)
```

*Explanation:* The class Point store x and y coordinates of a point. Inside the main, an array ptarr of size 5 of class type Point is created. This is similar to creating 5 objects with different name. The array ptarr is an object array of class Point type. The ptarr[0] denotes the first object, ptarr[1] the second and so on. In the main, input function for all the objects is called using the for loop and pass arbitrary points to function. Each object will be having different points as data members for each object are unique. The points of each object are displayed using show inside the for loop.

```
/*PROG 7.14 MERIT LIST OF STUDENTS */

import java.io.*;
import java.util.*;
class student
{
 private String sname;
 private int m1, m2, m3;
 private float per;
 void input()
 {
 Scanner sc = new Scanner(System.in);
 System.out.println("Enter the stduent name");
 sname = sc.next();
 System.out.println("Enter the marks in subject-1");
 m1 = sc.nextInt();
 System.out.println("Enter the marks in subject-2");
 m2 = sc.nextInt();
 System.out.println("Enter the marks in subject-3");
 m3 = sc.nextInt();
 per = (m1 + m2 + m3) / 3.0f;
 }
 void show()
 {
 System.out.println(sname + "\t" + per);
 }
 float getper()
 {
 return per;
 }
}
class JPS19
{
 public static void main(String args[])
 {
 final int s = 3;
 student[] sarr = new student[s];
 int i, j;
 student temp = new student();
 for (i = 0; i < s; i++)
 {
 sarr[i] = new student();
 sarr[i].input();
 }
 for (i = 0; i < s; i++)
 for (j = i + 1; j < s; j++)
 if (sarr[i].getper() < sarr[j].getper())
 {
 temp = sarr[i];
 sarr[i] = sarr[j];
 sarr[j] = temp;
 }
 System.out.println("Student Merit List\n");
 }
}
```

```

 System.out.println("\nNAME \tPer(%) \n");
 for (i = 0; i < s; i++)
 sarr[i].show();
 }
}

OUTPUT:
Enter the stduent name
Hari
Enter the marks in subject-1
60
Enter the marks in subject-2
80
Enter the marks in subject-3
85
Enter the stduent name
Man
Enter the marks in subject-1
76
Enter the marks in subject-2
87
Enter the marks in subject-3
69
Enter the stduent name
Ruchika
Enter the marks in subject-1
60
Enter the marks in subject-2
78
Enter the marks in subject-3
90
Student Merit List
NAME Per(%)
Man 77.333336
Ruchika 76.0
Hari 75.0

```

*Explanation:* The class student has 5 data members: name, m1, m2, m3 and per. For storing marks in three subjects there are m1, m2 and m3. The per is for storing percentage and the sname is for storing name. It is assumed that all marks are from 100 (though no checking has been done through it). In the function input, after marking, the percentage is found and stored in per. In the main, sorting is performed on the basis of percentage as the merit list of the students needs to be generated. As per is private, we have made a public member function getper which returns per in the main sorting is done on this function getper basis. Note that the whole object has been swapped while sorting is being done. For this purpose, a temporary object temp is taken and subsequently the merit list is displayed using show and for loop.

## 7.7 STATIC CLASS MEMBERS

Static variables are those which persist even after the control returns from the function. In terms of static members as class members, they are the members for a class and not for an object. Static data members are needed when only one piece of storage is needed for a particular piece of data, regardless of how many objects are created or even if no objects are created. static method is not associated

with any particular object of this class. This means that it is a method that one can call even if no objects are created. One such method is the main method of all Java programs.

It is known that the functions of a class are defined only once and they can be called by an object of the class, but static functions are those functions which belong to the class only. Static members can be either functions or static data. Static data members (field + method) are sometimes known as class variables or class methods as they belong to whole of the class.

### 7.7.1 Static Member Functions

The various points about the static member functions are as follows:

1. Static functions are those which are made static by placing the keyword `static` before the function definition inside the class.
2. Static functions are one for class and can be called as

---

```
classname. function_name
```

---

If `demo` is the class name and `stat` is the static function name then it can be called as `demo.stat()`.

3. In a static function, only the static data members, other static variables or other static functions can be used.
4. Although a static member function is called using a class name, it can be called explicitly using the object of the class.
5. Inside the static methods, only the static data and methods can be used, but a static method can be called from a non-static method.

In the previous chapter, several examples of static method were given but not in detail. So a few practical programs with their explanation are given below.

```
/*PROG 7.15 DEMO OF STATIC METHOD */

class demo
{
 static void show()
 {
 System.out.println("Welcome from static show");
 }
}
class JPS20
{
 public static void main(String args[])
 {
 demo d = new demo();
 d.show();
 demo.show();
 (new demo()).show();
 }
}
OUTPUT:
Welcome from static show
Welcome from static show
Welcome from static show
```

*Explanation:* A static method can be called from using an object of the class or using a class itself. Both the ways have been demonstrated in the above program. As new demo () creates an object, the last line in the main is also a correct way for calling a static method.

```
/*PROG 7.16 CALLING A STATIC METHOD FROM A NON STATIC METHOD */

class demo{
 static void show()
 {
 System.out.println("\n\nHello from static show");
 }
 void display()
 {
 System.out.println("\nHello from non static
display");
 show();
 }
}
class JPS21
{
 public static void main(String args[])
 {
 demo d1 = new demo();
 d1.display();
 }
}
OUTPUT:
Hello from non static display
Hello from static show
```

### 7.7.2 Static Data Members

It is known that whenever an object is created, separate copies of data members are created for each object. But in the case of static data members, only one copy of static data members is available which is shared among all the objects created. The various points about the static data members are as follows:

1. They are created by placing the static keyword before variable declaration.
2. All static variables get their default initial value as 0 when the first object of a class is created.
3. A single copy of the static data member is created which is shared among all objects. The changes made by one object on a static data member are reflected back to all other objects.
4. The lifetime of a static variable is the entire program.
5. They are used when one value common to the whole class is kept.
6. They can be accessed either by object name or by class name.

As an example of static member consider the code given below.

```
class demo
{
 static int num = 100;
}
demo d=new demo();
System.out.println(demo.num++); //prints 100
System.out.println(++d.num); //prints 101
```

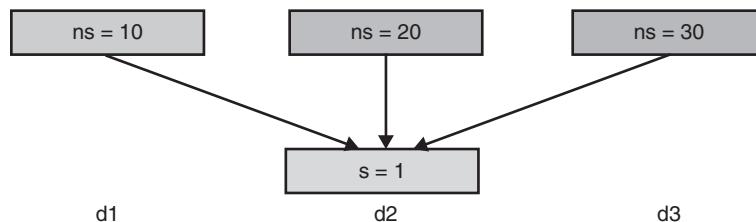
In this code, the static member num is accessed, once using a class and then using an object. As only one copy of num is available, the value of num increments twice.

Consider another example.

```
class demo
{
 static int s;
 int ns;
};

demo d1=new demo();
demo d2=new demo();
demo d3=new demo();
```

Assume value of s is 1 and values of ns for d1, d2 and d3 are 10, 20, and 30, respectively (Figure 7.3).



**Figure 7.3** Implementation of static data number

One for class, common to all objects

```
/*PROG 7.17 WORKING WITH STATIC AND NON STATIC DATA */

class demo
{
 static int snum = 20;
 int ns;
}

class JPS22
{
 public static void main(String args[])
 {
 demo d1 = new demo();
 d1.ns = 50;
 d1.snum++;
 demo d2 = new demo();
 d2.ns = 100;
 d2.snum++;
 System.out.println("\nns of d1=" + d1.ns);
 System.out.println("\nns of d2=" + d2.ns);
 System.out.println("\nStatic snum=" + demo.snum);
 }
}

OUTPUT:
ns of d1=50
ns of d2=100
Static snum=22
```

*Explanation:* The data member `snum` is `static` and so only one copy of it will be shared among all the objects created. But ‘`ns`’ is non-static and so for all objects of `demo` class a separate copy of `ns` will be available.

## 7.8 CONSTRUCTORS

A constructor is a special member function whose name is the same as the name of its class in which it is declared and defined. The purpose of the constructor is to initialize the objects of the class. The constructor is called so because it is used to construct the objects of the class. The constructor is used to construct the object at run time. A default constructor (‘no-arg’ constructor) is one without arguments that is used to create a class which has no constructor and the compiler will automatically create a default constructor which has been seen in all other programs where it has been created by writing `new classname();`

A small example of the constructor is given below.

```
class demo
{
 demo ()
 {
 System.out.println("Hello from constructor");
 }
}
class JPS
{
 public static void main(String args[])
 {
 demo d = new demo();
 }
}
OUTPUT:
Hello from constructor
```

The name of the class is `demo` and the following function definition is

```
demo()
{
 System.out.println("Hello from constructor");
}
```

a constructor of the class as the name of the function is the name of the class. The various features of the constructor are as follows:

1. The constructor invokes automatically when objects of the class are created. The declaration `new demo();` creates an object which automatically calls the constructor of the class and prints ‘Hello from constructor’.
2. The constructors are always public. If declared private then the objects can only be created inside the member functions, which serve no purpose.
3. The constructors are used to construct the objects of the class.
4. The constructor does not have any return type, not even `void` and therefore, it cannot return any value.
5. The constructor with no argument is known as default constructor of the class. The default constructor for the class `demo` will be `demo()`

6. The constructor that takes arguments like a function takes are known as parameterized constructors.
7. There is no limit of the number of constructors declared in a class but all must conform to rules of function overloading.

In the other programs discussed earlier, the objects were created using the constructor but no constructor in the program was created. Note that behind every object creation, a constructor is required. Then who was creating the objects in all these programs where the classes and objects were made use of? The answer is simple: When no constructor is created in the class, the compiler provides the default constructor for the class and that constructor is known as default constructor or do-nothing constructor. The sole purpose of this constructor is to construct the objects for the class.

```
/*PROG 7.18 CONSTRUCTOR RETURNING VALUES */

class demo
{
 demo()
 {
 System.out.println("Hello from constructor");
 }
 void demo()
 {
 System.out.println("Hello from function ");
 }
}
class JPS23
{
 public static void main(String args[])
 {
 demo d = new demo();
 d.demo();
 }
}
OUTPUT:
Hello from constructor
Hello from function
```

*Explanation:* In Java, the constructor can return values but they are not treated as constructor. They are treated as functions. Here `void demo()` is a function which returns nothing. In the `main`, `d.demo()` calls this function.

### 7.8.1 Constructors with Parameters

The constructors are similar to functions but they have the name as class name. Therefore, similar to functions, which takes arguments, one can also have constructors which can take arguments. The constructor which takes parameters is known as parameterized constructor. Depending on the type and number of arguments they may be overloaded. Again, when parameterized constructor and objects like `new demo()` assume `demo` is the class name are created, a compilation error will be generated. This is because whenever any type of constructor is created, the compiler does not provide one with the default constructor. It forces the programmer to create all such constructors which can build the object.

An example of this fact is given below.

```
/*PROG 7.19 DEMO OF PARAMETERIZED CONSTRUCTOR VER 1 */

class demo
{
 demo()
 {
 System.out.println("\nZero Argument Constructor");
 }
 demo(int x)
 {
 System.out.println("\n One Argument Constructor");
 }
 demo(int x, int y)
 {
 System.out.println("\n Two Argument Constructor");
 }
}
class JPS24
{
 public static void main(String args[])
 {
 demo d = new demo();
 d = new demo(1);
 d = new demo(1, 2);
 }
}
OUTPUT:
Zero Argument Constructor
One Argument Constructor
Two Argument Constructor
```

*Explanation:* The first line in the main calls **default constructor** for the class. In the next line, the one argument constructor is called due to new demo (1). In the next line, new demo (1, 2) calls the two argument constructor. Note that in all the three lines in the main the same reference has been used. The same can also be written in the following manner:

```
demo d = new demo();
demo d1 = new demo(1);
demo d2 = new demo(1, 2);
```

```
/*PROG 7.20 INTIALIZEING STRING DATA */

class demo
{
 private String str;
 demo()
 {
 System.out.println("\n\nHello from constructor");
 str = "Welcome in NMIMS";
 }
 demo(String s)
 {
 str = s;
```

```

 }
 void show(String s)
 {
 System.out.println(s);
 System.out.println("\nString is " + str);
 }
 }
class JPS26
{
 public static void main(String args[])
 {
 demo d;
 d = new demo();
 d.show("Object d");
 demo d1;
 d1 = new demo("Cutie Pie");
 d1.show("Object d1");
 }
}
OUTPUT:
Hello from constructor
Object d
String is Welcome in NMIMS
Object d1
String is Cutie Pie

```

*Explanation:* When the default constructor for object d is called String data member str is initialized to ‘Welcome in NMIMS’. When an argument constructor taking a String argument is called ‘Cutie Pie’ is assigned to str of object d1 and the previous String data is lost.

```

/*PROG 7.21 CONSTRUCTOR WITH ARRAY OF OBJECT */
class demo
{
 demo()
 {
 System.out.println("\nHello from constructor ");
 }
}
class JPS27
{
 public static void main(String args[])
 {
 demo[] d = new demo[5];
 for (int i = 0; i < d.length; i++)
 d[i] = new demo();
 }
}
OUTPUT:
Hello from constructor

```

*Explanation:* The important thing to note here is that the default constructor is not called when `demo[] d = new demo [5];` executes. It is called when `d[i] = new demo();` executes.

```
/*PROG 7.22 SEPARATING FLOAT AND INT PART */

class convert
{
 private double num;
 private int intp;
 private double realp;
 convert()
 {
 }
 convert(double n)
 {
 num = n;
 }
 void find()
 {
 intp = (int) (num);
 realp = num - intp;
 show();
 }
 void show()
 {
 System.out.println("Number =" + num);
 System.out.println("Integer Part =" + intp);
 System.out.println("Real Part =" + realp);
 }
}
class JPS28
{
 public static void main(String args[])
 {
 convert A = new convert(34.56);
 convert B;
 B = new convert(234.67);
 System.out.println("\n*****");
 System.out.println("\tFirst Object\n");
 System.out.println("*****");
 A.find();
 System.out.println("\n*****");
 System.out.println("\t\nSecond Object\n");
 System.out.println("*****");
 B.find();
 }
}

OUTPUT:

First Object

```

```

Number =34.56
Integer Part =34
Real Part =0.56000000000000023

Second Object

Number =234.67
Integer Part =234
Real Part =0.66999999999875

```

*Explanation:* The program is simple. First, the number num is converted into int by typecasting and stored in intp. It is then subtracted from the original number num to find out the real part.

In the program, the constructor convert () {} serves as empty/default/zero argument constructor. The default constructor serves nothing special here as it is having an empty body. This constructor can be used to provide the initial values to the members of the objects of the class. As a one argument constructor has been created the constructor can be called by writing convert c1 = new convert ("34.36"), but if it is written otherwise, it should be in the following way:

```

convert c1;
c1= convert("34.56");

```

Again, there must be a default constructor as stated in the above paragraph. If the object is not created by writing convert c1; there is no need to create the default constructor or empty constructor. But once the statement convert c1; is written there must be an empty constructor in the program. This is a must as in case of no constructor in the class, Java provides its default implicit constructor to construct the objects. (Remember, where there is no constructor there is no object). Once we have made constructors in our class of any type and in any number and we create the object by writing convert c1 without creating a default constructor, the compiler flashes the error 'default constructor required'. This constructor is a do-nothing constructor and is used to just satisfy the compiler. Try running the program by commenting the default constructor in the program.

```
/*PROG 7.23 ADDITION AND SUBTRACTION OF TWO COMPLEX NUMBER */
```

```

class complex
{
 double real;
 double imag;
 complex()
 {
 real = imag = 0;
 }
 complex(double r, double i)
 {
 real = r;
 imag = i;
 }
 complex sum(complex A, complex B)
 {
 complex temp = new complex();
 temp.real = A.real + B.real;
 temp.imag = A.imag + B.real;
 }
}

```

```

 return temp;
 }
 complex sub(complex A, complex B)
 {
 complex temp = new complex();
 temp.real = A.real - B.real;
 temp.imag = A.imag - B.imag;
 return temp;
 }
 void show()
 {
 System.out.println(real + "+j " + imag);
 }
}
class JPS29
{
 public static void main(String[] args)
 {
 complex c1 = new complex(2.0, 3.0);
 complex c2 = new complex(3.0, 2.0);
 complex c3 = new complex();
 complex c4 = new complex();
 c3 = c1.sum(c1, c2);
 c4 = c1.sub(c1, c2);
 System.out.print("\n\n C1 := ");
 c1.show();
 System.out.print(" C2 := ");
 c2.show();
 System.out.print(" SUM := ");
 c3.show();
 System.out.print(" SUB := ");
 c4.show();
 }
}
OUTPUT:
C1 := 2.0+j 3.0
C2 := 3.0+j 2.0
SUM := 5.0+j 6.0
SUB := -1.0+j 1.0

```

*Explanation:* In the program, there is one default constructor and the second constructor is with two parameters of type `double`. In order to find the addition and subtraction of two complex numbers, two functions `sum` and `sub` are written which takes two arguments of class `complex` type and return a value of class `complex` type. We also have another function `show` which displays the complex number. The class has two `private` data members, `real` and `imag`, which represents real and imaginary part of a complex number with the two statements.

```

complex c1 = new complex(2.0, 3.0);
complex c2 = new complex(3.0, 2.0);

```

We call the two arguments as constructors and set the real and imaginary part for objects `c1` and `c2`. For finding the sum of `c1` and `c2`, the function `sum` is called and `c1` and `c2` are passed as argument as

`c3 = c1.sum(c1, c2);` In the function sum, two real parts and two imaginary parts are added separately and stored in a temporary array which is returned and assigned to `c3`. Similarly, subtraction is performed by a call to sub-function and subtraction of `c1` and `c2` is assigned to `c4`. Later on, they are displayed by a call to `show`.

```
/*PROG 7.24 STACK SIMULATION */

import java.io.*;
import java.util.*;
class stack
{
 private int top;
 private int st[];
 private final int S = 10;
 stack()
 {
 top = -1;
 st = new int[S];
 }
 void push(int item)
 {
 if (top == 9)
 {
 System.out.println("\nStack is full");
 System.exit(0);
 }
 st[++top] = item;
 System.out.println("\nITEM PUSHED");
 }
 int pop()
 {
 if (top == -1)
 {
 System.out.println("\nStack is empty");
 System.exit(0);
 }
 return st[top--];
 }
}
class JPS30
{
 public static void main(String[] args)
 {
 stack s = new stack();
 int ch, item;
 Scanner sc = new Scanner(System.in);
 do{
 System.out.println("\n\nStack Simulation
 Demo\n");
 System.out.println("1. PUSH");
 System.out.println("2. POP");
 System.out.println("3. QUIT");
 ch = sc.nextInt();
 switch(ch)
 {
 case 1:
 item = sc.nextInt();
 s.push(item);
 break;
 case 2:
 item = s.pop();
 System.out.println("Popped Item is " + item);
 break;
 case 3:
 System.out.println("Exiting Program");
 System.exit(0);
 break;
 default:
 System.out.println("Invalid Choice");
 }
 }while(ch != 3);
 }
}
```

```
System.out.print("\nEnter your choice:=");
ch = sc.nextInt();
switch (ch)
{
 case 1:
 System.out.print("\nEnter the
 item:=");
 item = sc.nextInt();
 s.push(item);
 break;
 case 2:
 item = s.pop();
 System.out.println("\nItem poped=
 " + item);
 break;
 case 3:
 System.out.println("\nBYE BYE!!!!");
 System.exit(0);
 break;
 default:
 System.out.println("\nERROR!!!!!");
 break;
}
} while (ch >= 1 && ch >= 3);
}

OUTPUT:
Stack Simulation Demo
1. PUSH
2. POP
3. QUIT
Enter your choice:=1
Enter the item:=12
ITEM PUSHED
Stack Simulation Demo
1. PUSH
2. POP
3. QUIT
Enter your choice:=2
Item poped=12
Stack Simulation Demo
1. PUSH
2. POP
3. QUIT
Enter your choice:=3
BYE BYE!!!
Stack Simulation Demo
1. PUSH
2. POP
3. QUIT
Enter your choice:=4
ERROR!!!!!
```

**Explanation:** The program is simulating stack using an array. A stack is a data structure in which the last item inserted is taken out first. This being the reason they are known as LIFO (last in first out). Inserting an item in a stack is termed as push and taking an item out from the stack is termed as pop. Only one item can be pushed at a time which is added on to the top of the previous item pushed. Similarly, only one top most items can be popped back. For keeping track of the top item, a data member top is initialized to -1 when the object of the class stack is created. The program is simple: An array is taken as data member of class size 10 by the name st which is the stack. In order to push item into this stack st the top is incremented and item is assigned to st as `st[top] = item`. After each item is pushed, top is incremented. Before pushing an item, first it has to be checked whether there is a space for the new item in the stack. This is done in the following way:

```
if (top == 9)
{
 System.out.println("\nStack is full");
 System.exit(0);
}
```

If top == 9, the stack is full and the program is terminated. When an item is popped, the value is taken at `st[top]` and the top is decremented by 1 and `st[top--]` is returned. Before popping an item, it is checked whether the stack is empty or not. This is done in the following way:

```
if (top == -1)
{
 System.out.println("\nStack is empty");
 System.exit(0);
}
```

If top is -1, the stack is empty and the program is terminated. The diagrammatical representation of the stack operation is shown in Figure 7.4 (a–e).

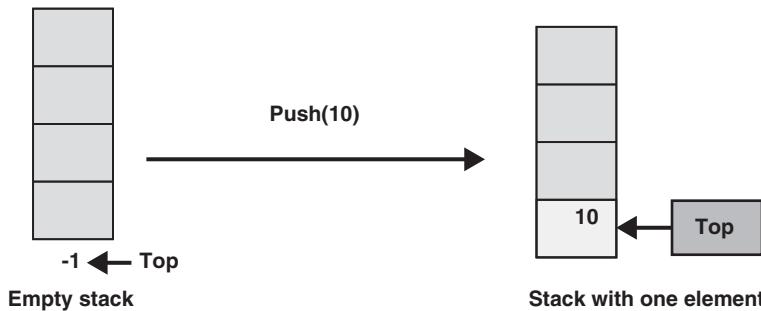


Figure 7.4 (a) Stack after push operation

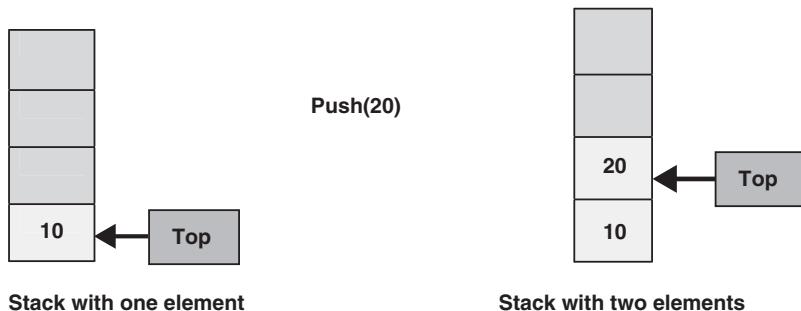


Figure 7.4 (b) Stack after push operation

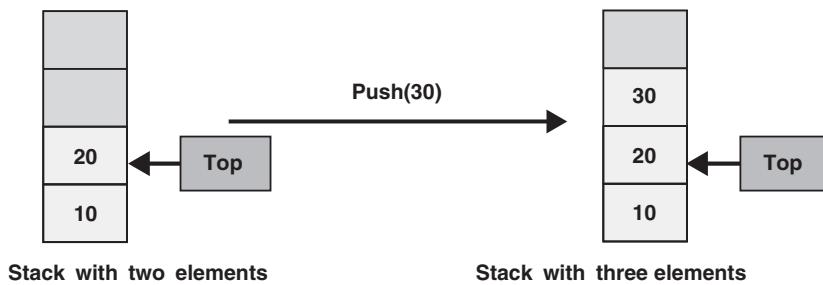


Figure 7.4 (c) Stack after push operation

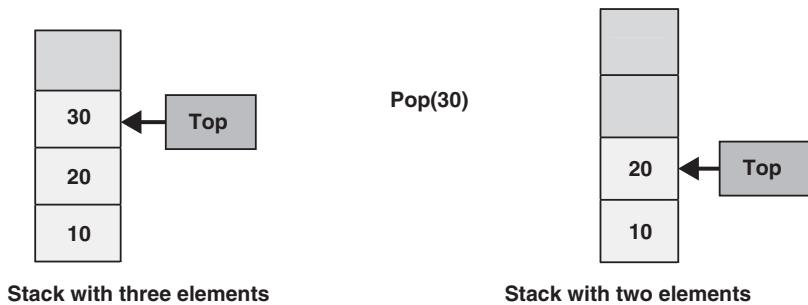


Figure 7.4 (d) Stack after pop operation

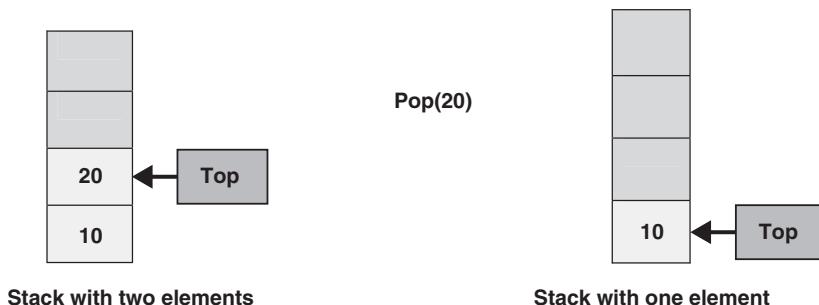


Figure 7.4 (e) Stack after pop operation

```
/*PROG 7.25 QUEUE SIMULATION */

import java.io.*;
import java.util.*;
class queue
{
 private int front, rear;
 private int que[];
 private final int S=10;
 queue()
 {
 front = rear = -1;
 que = new int[S];
 }
}
```

```
 }
 void insert (int item)
 {
 if(rear==S-1)
 {
 System.out.println("QUEUE IS FULL");
 System.exit(0);
 }
 if(front===-1)
 front =0;
 que[++rear]=item;
 System.out.println("ITEM INSERTED");
 }
 int del()
 {
 int item;
 if(front===-1)
 {
 System.out.println("QUEUE IS EMPTY");
 System.exit(0);
 }
 item=que[front];
 if(front==rear)
 front =rear =-1;
 else
 front++;
 return item;
 }
};

class JPS31
{
 public static void main(String args[])
 {
 queue q=new queue();
 int ch, item;
 Scanner sc= new Scanner(System.in);
 do
 {
 System.out.println("\n\nQUEUE DEMO");
 System.out.println("1. INSERT");
 System.out.println("2. DELETE");
 System.out.println("3. QUIT");
 System.out.print("\nEnter your choice:=");
 ch = sc.nextInt();
 switch(ch)
 {
 case 1:
 System.out.print("\nEnter the
item");
 item=sc.nextInt();
 q.insert(item);
 break;
```

```

 case 2:
 item=q.del();
 System.out.println("Item
deleted:=
 "+item);
 break;
 case 3:
 System.out.println("BYE
BYE!!!");
 System.exit(0);
 break;
 default:
 System.out.
println("ERROR!!!!");
 }
 }
}

OUTPUT:
QUEUE DEMO
1. INSERT
2. DELETE
3. QUIT

Enter your choice:=1
Enter the item12
ITEM INSERTED

QUEUE DEMO
1. INSERT
2. DELETE
3. QUIT

Enter your choice:=2
Item deleted:=12
QUEUE DEMO
1. INSERT
2. DELETE
3. QUIT

Enter your choice:=3
BYE BYE!!!

```

*Explanation:* The program simulates the working of a queue using array and class. The queue is a data structure with two pointers: front and rear. Whenever a new item is added to the queue, a rear pointer is used. It is incremented by 1. If it is the first item inserted the front also becomes 1. The front pointer is used when an item is deleted from the queue and the front pointer is decremented by 1. If it is the last item in the queue, the front and rear are both equal to -1. The queue is also known as FIFO (first in first out) as items are added always at the end of the queue (rear end) and are always deleted from front of the queue.

Initially, when the default constructor is called, the front and rear are initialized to -1. For insertion of item into the queue (implemented as an array queue), the rear is incremented but it is also checked whether there is space for the element to be inserted:

```

if(rear==S-1)
{
 System.out.println("QUEUE IS FULL");
 System.exit(0);
}

```

If the item was the first item inserted front was  $-1$  earlier so it's made to  $0$ . Item is inserted as que  $[+rear] =$  item. Pointer rear was also incremented.

For the deletion of item from the queue, the front pointer is used. The element in the queue at the front is returned as que  $[front]$ . Again, before deletion, it is checked whether the queue is empty or not in the following way:

```

if(front===-1)
{
 System.out.println("QUEUE IS EMPTY");
 System.exit(0);
}

```

Now, after deletion, if the front becomes equal to the rear (last item deleted) then both are assigned the value  $-1$  else front is incremented.

```
/*PROG 7.26 DYNAMIC ARRAY OF OBJECTS */
```

```

import java.io.*;
import java.util.*;
class demo{
 private int num;
 demo(int n){
 num = n;
 }
 void show(){
 System.out.println("\tnum :=" + num);
 }
}
class JPS32{
 public static void main(String[] args){
 demo d[] ={
 new demo(15),
 new demo(20),
 new demo(35),
 new demo(50),
 };
 System.out.println("\n*****");
 System.out.println("\tArray elements are");
 System.out.println("\n+++++");
 for (int i = 0; i < d.length; i++)
 d[i].show();
 }
}

```

**OUTPUT:**

```

 Array elements are
+++++

```

```

 Array elements are
+++++
 num :=15
 num :=20
 num :=35
 num :=50

```

*Explanation:* Note how an array of objects is initialized.

```

demo d[] ={
 new demo(15),
 new demo(20),
 new demo(35),
 new demo(50),
 };

```

Each `new demo(x)` (`x` may any int value) creates an object in the array by calling one argument constructor and later on, the value of `num` is displayed for all objects using `show`.

## 7.9 COPY CONSTRUCTOR

A copy constructor is a constructor in which a single object is passed as an argument. This is used to initialize an object from another object. Look at the statements given below.

```

demo d1= new demo();
demo d2= new demo(d1);

```

Note that the second statement makes a call to copy the constructor defined in the class. When there are two objects, say `d1` and `d2` and if `d2=d1` is written this results in copying only the reference `d1` to `d2` and after the assignment, both `d1` and `d2` refer to the same object. For a class `demo` copy, the constructor is written in the following way:

```

demo (demo d)
{
 //copy constructor code;
}

```

```
/*PROG 7.27 COPY CONSTRUCTOR IN JAVA */
```

```

class cheque
{
 boolean signed;
 cheque(boolean b)
 {
 signed = b;
 }
 cheque(cheque ch)
 {
 signed = ch.signed;
 }
}

```

```

public void show()
{
 if (signed)
 System.out.println("\nCheck has signed");
 else
 System.out.println("\nCheck has not signed");
}
}
class JPS33
{
 public static void main(String args[])
 {
 cheque ch1 = new cheque(true);
 cheque ch2 = new cheque(ch1);
 ch1.show();
 ch1.signed = false;
 ch2.show();
 ch1.show();
 }
}
OUTPUT:
Check has signed
Check has signed
Check has not signed

```

*Explanation:* The code shown in bold is the copy constructor in Java. In the main, when the statement shown in bold executes, the copy constructor is called for ch2 and is passed as argument. The signed Boolean data members get the signed value of ch1 as the signed was having a true value for ch1 and also the signed value of ch2 will be having a true value too. In the main, the show was first called for ch1. This displays that the check has been signed. Next, the value of signed for object ch1 is changed to false and then show for ch1 and ch2 is called. The show for ch2 shows check has signed and show for ch1 shows check was not signed. Thus, the copy constructor has worked.

## 7.10 THE this REFERENCE

The this reference is a special built-in reference. It is a special keyword which holds the reference to the current object. This means that the current object can be referred using this reference anywhere in the class. The this reference can be used only inside the class, i.e., only inside the member functions of the class and cannot be used outside the class. The this reference is a constant reference. Suppose, there is a method fun of the class demo and d1 and d2 are its objects. The method fun takes just one argument of type integer. The call to fun with d1 and d2 is given below.

```
d1.fun(10);
d2.fun(20);
```

How does the method fun come to know whether it is being called for object d1 or object d2? Internally, in all function call the compiler sends through an object the first argument as reference of the object which is hidden from the programmer. Thus, the above two calls become something like the ones given below.

```
demo.fun(d1,10);
demo.fun(d2,20);
```

The above is done internally by the compiler. In order to have the reference to the current object inside the method, that reference can be used using `this` keyword as the object secretly passed can be used by using `this`. The `this` keyword, which can be used only inside a method, produces the reference to the object for which the method has been called for. This reference can be treated just like any other object reference. It must be kept in mind that a method is called from within another method of the class, which need not be used. The method is simply called. The current `this` reference is automatically used for the other method.

Some of the important points about this reference are as follows:

1. It is an implicit reference used by the system.
2. It stores the reference of the current object in context.
3. It is a final reference to an object.
4. It can only be used within non-static functions of the class.
5. It is non-modifiable, and assignments to this are not allowed.

```
/*PROG 7.28 DEMO OF THIS REFERENCE VER 1 */
class Mouse

{
 String cname;
 String type;
 float price;
 Mouse(String cname, String type, float price)
 {
 this.cname = cname;
 this.type = type;
 this.price = price;
 show();
 }
 void show()
 {
 System.out.println("\nCompany:=" + this.cname);
 System.out.println("\nType :=" + this.type);
 System.out.println("\nPrice :=" + this.price);
 }
}
class JPS34
{
 public static void main(String args[])
 {
 Mouse m = new Mouse("DELL", "Optical", 350.95f);
 }
}

OUTPUT:
Company :=DELL
Type :=Optical
Price :=350.95
```

**Explanation:** In the main, the object m is the current object in context; so this reference stores the reference of object m. In the parameterized constructor for class Mouse, this.name, this.type and this.price refer to data fields of the current object m. They can also be written without this. But as the local parameters are having the same name as the fields of the class, this has to be used. Similarly, in show method this is not necessary.

```
/*PROG 7.29 RETURNING CURRENT OBJECT USING THIS */

class Game
{
 String pname;
 int points;
 Game(String pname, int points)
 {
 this.pname = pname;
 this.points = points;
 }
 void show()
 {
 System.out.print("Name := " + this.pname);
 System.out.println("\tPoints := " + this.points);
 }
 Game check(Game g)
 {
 if (this.points > g.points)
 return this;
 else
 return g;
 }
}
class JPS35
{
 public static void main(String args[])
 {
 Game g1 = new Game("Hari", 1560);
 Game g2 = new Game("Man", 1665);
 System.out.println("\n++++++");
 System.out.println("\tPlayer-1");
 System.out.println("*****");
 g1.show();
 System.out.println("\n*****");
 System.out.println("\tPlayer-2");
 System.out.println("++++++");
 g2.show();
 Game winner;
 winner = g1.check(g2);
 System.out.print("\n Winner is " + winner.pname);
 System.out.println(" with "+winner.points+
 " points");
 }
}
```

OUTPUT:

```
+++++
Player-1

Name := Hari Points :=1560

Player-2
+++++
Name := Man Points :=1665
Winner is Man with 1665 points
```

*Explanation:* The class Game has two fields: pname and points. Using the parameterized constructor of class Game, the data members of the class are initialized for objects g1 and g2. The method check takes an object and returns another object. This method checks and declares who is the winner by comparing the points of two players: g1 and g2. The object g1 cannot be returned from the method as it is not available in the method check. But it is a fact that this reference stores the reference of the current object in context. So this reference inside check method represents g1. The returned reference is assigned to the winner in the main and the result is displayed.

```
/*PROG 7.30 METHOD RETURNING THIS REFERENCE */

class demo
{
 int num;
 demo(int num)
 {
 this.num=num;
 }
 void show()
 {
 System.out.println("num :="+this.num);
 }
 demo incr()
 {
 num++;
 return this;
 }
}
class JPS36
{
 public static void main(String args[])
 {
 demo d1 = new demo(10);
 System.out.println("\nCall incr() three times");
 d1.incr().incr().incr().show();
 }
}
OUTPUT:
Call incr() three times
num :=13
```

*Explanation:* The method `incr` returns a reference to the current object. In the `main` when `d1.incr` is called, `num` is incremented to 11 and the reference of the current object is returned which again call `incr`. This is done for three times and so `num` becomes 13. In the end, `show` is called which displays the `num = 13`.

```
/*PROG 7.31 THIS REFERENCE FOR CALLING CONSTRUCTOR FROM
CONSTRUCTOR */

class demo{
 String str;
 int num;
 demo(String s) {
 str=s;
 System.out.println("Constructor with string
 argument");
 }
 demo(int x)
 {
 num=x;
 System.out.println("Consturctor with int
 argument");
 }
 demo(String str, int num){
 this(str);
 this.num=num;
 System.out.println("Constructor with string & int
 argument");
 }
 demo(){
 this("Default", 235);
 System.out.println("Constructor with default
 argument");
 }
 void show()
 {
 System.out.println("Str :="+str+"\tnum
 :="+this.num);
 }
}
class JPS37
{
 public static void main(String arg[])
 {
 demo d1 = new demo();
 d1.show();
 d1 = new demo(20);
 d1.show();
 d1 = new demo("fun");
 d1.show();
 }
}
```

```

 d1 = new demo("welcome", 345);
 d1.show();
 }
}

OUTPUT:

Constructor with string argument
Constructor with string & int argument
Constructor with default argument
Str :=Default num :=235
Consturctor with int argument
Str :=null num :=20
Constructor with string argument
Str :=fun num :=0
Constructor with string argument
Constructor with string & int argument
Str :=welcome num :=345

```

*Explanation:* The line in bold shows that this can be used to call the constructor within the constructor with arguments. Follow the program step by step and the output will be obtained as shown above.

## 7.11 GARBAGE COLLECTION AND FINALIZE METHOD

In computer science, garbage collection (also known as GC) is a form of automatic memory management. The garbage collector attempts to reclaim garbage or memory used by objects that can never be accessed or mutated again by the application. Garbage collection was invented by John McCarthy around 1959 to solve the problem of manual memory management in his Lisp programming language.

Java has the garbage collector to reclaim the memory of objects that are no longer used. This garbage collection is done automatically in Java without the user intervention. In Java, sometimes one may require to do some clean-up work. For this purpose, one can override the `finalize` method whose prototype is given as follows:

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

The `finalize` method is called by the garbage collector on an object when the garbage collection determines that there are no more references to the object. When the garbage collector is ready to release the storage used for the object, it will first call `finalize()` and only when the next garbage collection passes, it will reclaim the memory of the object. So, if `finalize()`, is chosen to be used it gives the ability to perform some important clean-up at the time of garbage collection.

In Java, an object is not always garbage collected which means that there is no guarantee that `finalize` will be called.

## 7.12 THE FINAL KEYWORD REVISITED

The `final` keyword was discussed in the first chapter for declaring constants. It can also be used with object references rather than with primitive data types. With a primitive, `final` makes the value a constant but with an object reference, `final` makes the reference a constant. Once the reference is initialized to an object, it can never be changed to point out another object. However, the object itself can be modified. See the code given in the program below.

```
/*PROG 7.32 DEMO OF FINAL REFERENCE */

class demo
{
 String str;
 final int num=20;
 static final float PI=3.14f;
 demo(String s)
 {
 str=s;
 }
}
class JPS38
{
 public static void main(String args[])
 {
 final demo d = new demo("fun1");
 demo d1 = new demo("fun2");
 d = d1;
 d = new demo("Changed");
 }
}

OUTPUT:

COMPILATION ERROR
```

```
E:\WINDOWS\system32\cmd.exe
C:\JPS\ch7>javac JPS38.java
JPS38.java:18: cannot assign a value to final variable d
 d = d1;
 ^
JPS38.java:19: cannot assign a value to final variable d
 d = new demo("Changed");
 ^
2 errors
```

**Figure 7.5** Error during the compilation of Program 7.32

The program gives the compilation error. The object reference `d` is a constant which cannot take reference of any other object. So the lines showed in bold are responsible for flashing the compilation errors. Also note that the member fields can be declared as constant and initialized so that their values are known as compile time.

### 7.12.1 Blank Finals

Java allows the creation of blank finals, which are fields that are declared as final but are not given an initialization value. The blank final must be initialized before it is used. However, blank finals provide more

flexibility in the use of the final keywords; for example, a final field inside a class can now be different for each object and yet nobody can modify it.

```
class demo
{
 final int blank_final;
 demo(int num)
 {
 blank_final = num;
 }
}
```

In the main, the following can be written:

```
demo d1=new demo(20); //blank_final is 20 for object d1
 //and is a constant
demo d2=new demo(40); //blank_final is 40 for object d2
 //and is a constant
```

### 7.12.2 Final Arguments to Methods

Arguments can be passed to methods and ensured that they are not modified inside it. An example is given below.

```
class demo
{
 String str;
 demo(final String s)
 {
 str = s;
 s = "hello"; //can't modify a final argument
 }
}
class Temp
{
 public static void main(String[] args)
 {
 demo d = new demo("Can't Change Me");
 }
}
```

**String s** is final in the constructor **demo** and so it cannot be modified.

## 7.13 PONDERABLE POINTS

1. In Java, all objects are created dynamically by the new operator.
2. All functions and methods must be a part of the class. One cannot declare methods inside the class and define them outside the class.
3. For declaring a method or field as private, protected or public, each field or method must be individually tagged with a visibility modifier.
4. In Java, a protected member cannot be accessed outside the class.
5. In Java, a constructor can act as a function and can return value.
6. The base/root class of all Java classes is the object class.
7. The this reference holds the reference of the current object in context.
8. The static members are also known as class member/data.
9. In Java, uninitialized constants can be created which are known as blank finals.
10. Garbage collector is a program which runs in the background and claims the unused memory blocks.
11. Java has no destructor.

## REVIEW QUESTIONS

1. What is an object and class? Explain with an example.
2. What is the difference between class method (static) and instance method?
3. What is the difference between public member and private member of a class?
4. Is it legal for a static method to invoke a non-static method? Yes or No. If no, give reason.
5. Can an abstract method be declared final or static?
6. When do we declare a method of a class static?
7. Describe the different level of access protection available in Java.
8. When do we declare a member variable method or class final?
9. State true/false:
  - (a) Abstract class cannot be instantiated.
  - (b) A final method can be overridden.
  - (c) Declaring a method synchronized guarantees that the deadlock cannot occur.
  - (d) Volatile modifier is used to modify by synchronous threads.
10. What is the error in the following code snippet?
 

```
class JPS_test
{
 abstract void view();
}
```
11. Write short notes on the following
  - (a) this
  - (b) transient
  - (c) volatile.
12. Compare and contrast concrete class and abstract class.
13. What is synchronization? Why do we need it?
14. Design a class to represent account, include the following members.
 

**Data Members**

Name of depositor—string  
 Account Number—int  
 Type of Account—boolean  
 Balance amount—double

**Methods**

  - (a) To assign initial values (using constructor)
  - (b) To deposit an amount after checking balance and minimum balance 50.
  - (c) To display the name and balance.
15. Distinguish between the two statements:
 

```
JPS c2 = new JPS(c1);

JPS c2 = c1;
```

c1 and c2 are objects of JPS class.
16. Java does not support destructor. Discuss.
17. Why do we need finalize () method?
18. State True or False: The compiler automatically creates a default constructor for the class, if it has no other constructor declared explicitly.
19. Create a class object interest with a constructor. Write a Java program to find the simple interests using the formula.  

$$\text{Simple Interest} = \frac{P \times N \times R}{100}$$

P—principle, N—number of year, R—rate of interest
20. Explain the constructor method. How does it differ from other member function?

21. Develop a program to illustrate a copy constructor so that a string may be duplicated into another variable either by assigning or copying.
22. Write a program to find the volume of a box that has its sides w, h, d as width, height, and depth, respectively. Its volume is  $v = w * h * d$  and also find the surface area given by the formula  $s = 2(wh + hd + dw)$ .
23. Write a program using
- Default constructor.
  - Arguments constructor.
  - Copy constructor.
- To find the area of a rectangle using the following formula  $\text{area} = \text{length} * \text{breadth}$ .
24. Write a program simply prints "Wonder of objects" inside the main method. Without using any print statement or concrete methods or dot operators inside the main method.
25. A class weight is having a data member pound, which will have the weight in pounds. Using a conversion function, convert the weight in pounds to weight in kilograms which is of double type. Write a program to do this.
- 1 pounds = 1 kg/0.453592
- Use default constructor to initial assignment of 1000 pounds.
26. Why does the compiler generate the following error: Exception in thread "main" java.lang.NoSuchMethodError: main?
27. Which of the following signatures is valid for the main() method? Justify.
- public static void main(String rk[])
  - static public void main(String argv)
  - public static void main(String [] [] args)
  - public static int main(String args[])
28. Can we have more than one default constructor?
29. For a class Animal, write a method sleeps() that uses only one instance variable of Animal. The code
- ```
Animal a = new Animal
("fido", "woof");
a.sleep();
```
- causes the output
- fido is sleeping
30. Define overloaded method and illustrate your definition with a concise and appropriate Java example or your own. Do not use a built-in method as your example; write one of your own.
31. Write code that proves the following statement: If a name can refer to either a local variable or a parameter, that name refers to the local variable.
32. Consider the following class:
- ```
class JPS
{
 private int var;
 //.....
 public static void JPS1()
 {
 JPS a = new JPS();
 System.out.println(a.var);
 }
 //.....
}
```
- Does this compile? Does static method JPS1 reference an instance variable? Why is it allowed here?
33. College students typically have at least the following properties: last name, first name, major, home, city, and home state. Following the guidelines of this chapter design a class Student. Write and run drivers that test and demonstrate the capabilities you have built into your class.
34. Consider the following code:
- ```
class JPS
{
    private int var;
    //.....
    public static void JPS1()
    {
        System.out.println(var);
    }
    //.....
}
```
- It will not compile. Explain why.
35. How does Java react if we use private instead of public? Explain.
36. How does Java react if we omit the static modifier? Explain.
37. How does Java react if we omit the phrase throws exception? Explain.

Multiple Choice Questions

1. A class is the basic unit of
 - (a) function
 - (c) encapsulation
 - (b) inheritance
 - (d) none of the above
2. The class is an abstract data type (ADT) so creation of class simply creates a
 - (a) function-like structure
 - (b) abstraction
 - (c) template
 - (d) none of the above
3. The default value of long data types for class is
 - (a) false
 - (c) (short)0
 - (b) 0L
 - (d) none of the above
4. The lifetime of a static variable is the
 - (a) entire program
 - (b) only in main method
 - (c) only inside the class
 - (d) none of the above
5. All static variables get their default initial value as _____ when first object of a class is created.
 - (a) 0
 - (c) 2
 - (b) 1
 - (d) 3
6. The constructor is always
 - (a) private
 - (c) protected
 - (b) public
 - (d) none of the above
7. Find the correct statement.
 - (a) We cannot pass argument in the constructor
 - (b) We can create only one constructor for a program
 - (c) Constructor does not have any return type, not even void
 - (d) None of the above
8. We can create _____ constructor in the program
 - (a) 1
 - (c) 5
 - (b) 3
 - (d) no limit
9. A copy constructor is a constructor in which _____ object is passed as argument.
 - (a) single
 - (c) three
 - (b) two
 - (d) none
10. The code given below shows the example for


```
JPS obj1 = new JPS();
JPS obj2 = new JPS (obj1);
```

 - (a) Constructor
 - (c) Copy constructor
 - (b) Destructor
 - (d) None of the above
11. this reference is
 - (a) implicit reference
 - (b) explicit reference
 - (c) both (a) and (b)
 - (d) none of the above
12. this reference is a built-in reference can only used within
 - (a) static function of the class
 - (b) non-static function of the class
 - (c) cannot be used
 - (d) none of the above
13. In Java automatic memory management is done by
 - (a) this reference
 - (b) destructor
 - (c) garbage collection
 - (d) none of the above
14. finalize method is used for
 - (a) reclaiming the memory
 - (b) automatic memory management
 - (c) overriding for clean-up work
 - (d) none of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

- | | | | | | | |
|------|------|-------|-------|-------|-------|-------|
| 1. c | 2. c | 3. b | 4. a | 5. a | 6.b | 7. c |
| 8. d | 9. a | 10. c | 11. a | 12. b | 13. c | 14. c |

Inheritance in Java

8

8.1 INTRODUCTION

The term inheritance implies that one class can inherit a part or all of its structure and behaviour from another class. Inheritance provides the idea of reusability, i.e., a code once written can be used again and again in a number of new classes. The class that does the inheriting is called a **subclass** of the class from which it inherits. If class B is a subclass of class A, it can also be said that class A is a **super class** of class B. (Sometimes the terms **derived class** and **base class** are used instead of subclass and super class) (Figure 8.1). A subclass can add to the structure and behaviour that it inherits. It can also replace or modify inherited behaviour (though not inherited structure). The relationship between a subclass and a super class can be shown in a diagram with the subclass below, and connected to, its super class.

In Java, when a new class is created, it can be declared that it is a subclass of an existing class. In order to define a class B as a subclass of a class A, it should be written as follows:

```
class B extends A
{
    //functions and fields;
}
```

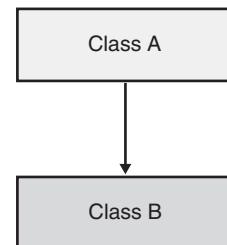


Figure 8.1 Implementation of inheritance

8.2 TYPES OF INHERITANCE

Inheritance is generally of five types (Figure 8.2): single level, multilevel, multiple, hierarchical and hybrid.

The syntax of deriving a new class from an already existing class is shown as follows:

```
class new_class_name extends mode old_class_name
{
}
```

Where **class** is the keyword used to create a class. **new_class_name** is the name of new derived class. **old_class_name** is the name of an already existing class. It may be a user-defined or a build-in class. The **extend** keyword is used for inheriting the class.

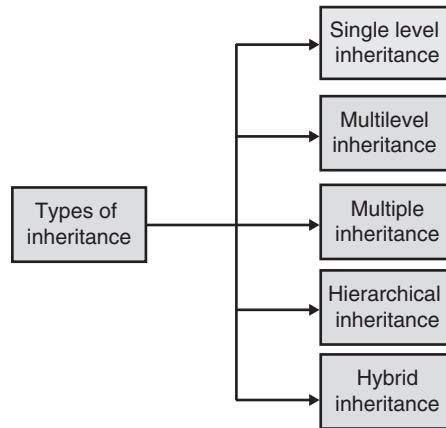
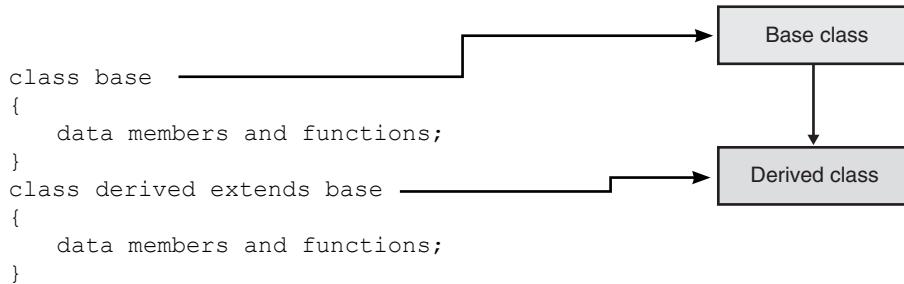


Figure 8.2 Different types of inheritance

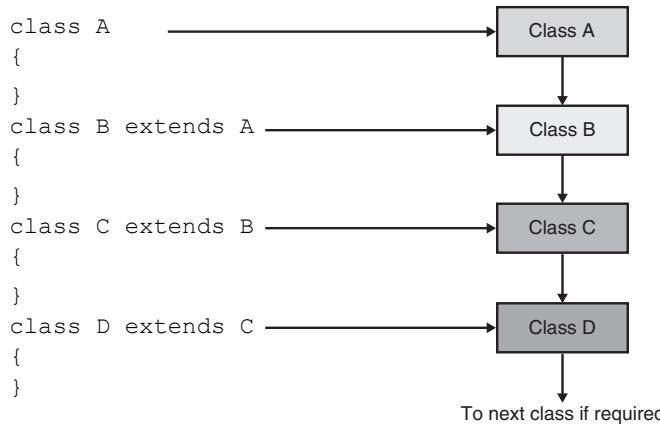
8.2.1 Single-level Inheritance

In single level inheritance, there is just one base and one derived class. It is represented as follows:
In Java code, this can be written as follows:



8.2.2 Multilevel Inheritance

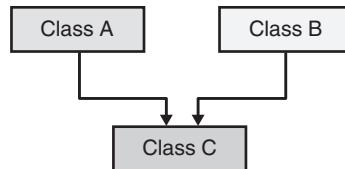
In multilevel inheritance, there is one base class and one derived class at the same time at one level. At the next level, the derived class becomes base class for the next derived class and so on. This is shown as:



The class A and class B together form one level, class B and class C together form another level and so on. For class B, class A is the parent and for class C, class B is the parent; thus in this inheritance level, A can be called the grandparent of class C, and class C the grandchild of class A.

8.2.3 Multiple Inheritance

In a multiple inheritance, a child can have more than one parent, i.e., a child can inherit properties from more than one class. Diagrammatically, this is as shown below:



Unfortunately, *Java does not support the multiple inheritances through classes though it supports using interfaces* which is discussed in the next chapter. That is, in Java one cannot write the following syntax:

```
class A {} class B {} class C extends A extends B
```

8.2.4 Hierarchical Inheritance

In this type of inheritance, multiple classes share the same base class. That is, numbers of classes inherit the properties of one common base class. The derived classes may also become base classes for other classes. This is shown in Figure 8.3.

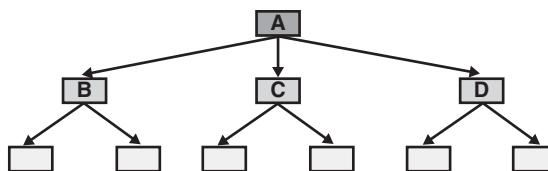


Figure 8.3 The hierarchical inheritance

For example, a university has a number of colleges under its affiliation. Each college may use the university name, the name of its chairperson, its address, phone number, etc.

Similarly, a vehicle possesses a number of properties or features and the common properties of all the vehicles may be put under one class vehicle, and different classes like two-wheeler, four-wheeler and three-wheeler can inherit the vehicle class.

As another example, in an engineering college, various departments can be termed as various classes which may have one parent common class—the name of the engineering college. Again for each department, there may be various classes, like lab staff, faculty class, etc. In Java code, the first level can be seen as follows:

```

class A
{
}
class B extends A
{
}
class C extends A
{
}
class D extends A
{
}
```

8.2.5 Hybrid Inheritance

Consider Figure 8.4 as shown below.

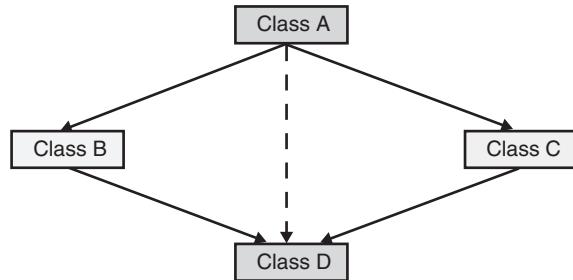
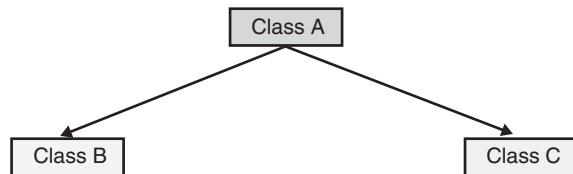
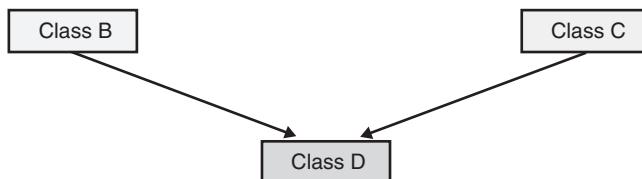


Figure 8.4 Implementation of hybrid inheritance

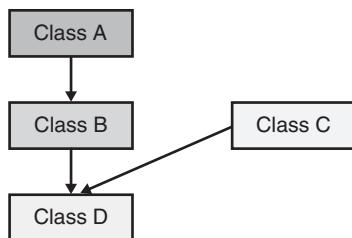
For the first half of the figure, the hierarchical inheritance is shown by breaking the figure:



The second half of the figure shows multiple inheritance:



The second figure for hybrid inheritance may be viewed as shown below:



As said earlier, Java does not support multiple inheritance through classes, so code for the above will be given while working with interfaces.

8.3 PROGRAMMING EXAMPLES

Explanation: For class A a single function `show_A` is declared. The line `class B` extends `A` tells the compiler that `B` is a new class and `A` is inherited in `B`. This makes `A` the parent of `B`, where `A` and `B` are also known as the base class and derived class, respectively. In `B`, any data or function has not been defined; therefore, it contains only inherited members from `A`. In the main function, an object of `B` is created and the function `show_A()` is called, which was inherited from `A`.

```
/*PROG 8.2 DEMO OF SINGLE LEVEL INHERITANCE VER 2 */  
  
import java.io.*;  
import java.util.*;  
class base  
{  
    private int sup a;
```

```

public void input1(int x)
{
    sup_a = x;
}
public void show1()
{
    System.out.println("\n Value of sub_a");
    System.out.println("*****");
    System.out.println("nsup_a:=" + sup_a);
    System.out.println("*****");
}
};

class derived extends base
{
    private int sub_a;
    public void input2(int x)
    {
        sub_a = x;
    }
    public void show2()
    {
        System.out.println("\n Value of sub_b");
        System.out.println("*****");
        System.out.println("sub_a:=" + sub_a);
        System.out.println("*****");
    }
}
class JPS2
{
    public static void main(String args[])
    {
        derived obj = new derived();
        obj.input1(200);
        obj.input2(400);
        obj.show1();
        obj.show2();
    }
}2

```

OUTPUT:

```

Value of sub_a
*****
sup_a:=200
*****
Value of sub_b
*****
sub_a:=400
*****

```

Explanation: In the program in base class, there is one private data member `sub_A` and two functions for input and show. After inheriting class base in class derived, the class derived has a total of four public member functions: two of its own and two inherited from class base. The class derived also has one private data member `sub_A`. In the main function, all four functions are called by an object of derived class.

```
/* PROG 8.3 ACCESSING PRIVATE VARIABLE */

class A
{
    private int num_A;
    void input_A(int x)
    {
        num_A = x;
    }
    int getnum()
    {
        return num_A;
    }
}
class B extends A
{
    private int num_B;
    void input_B(int x)
    {
        num_B = x;
    }
    void show()
    {
        System.out.println(" num_A := " + getnum());
        System.out.println(" num_B := " + num_B);
    }
}
class JPS5
{
    public static void main(String[] args)
    {
        B obj = new B();
        System.out.println("Value of num_A and num_B is ");
        obj.input_A(100);
        obj.input_B(200);
        obj.show();
    }
}

OUTPUT:

Value of num_A and num_B is
num_A := 100
num_B := 200
```

Explanation: In earlier chapters, it was shown how to access a private data member of a class. The method was to simply define a public member function which returns the value of the private data member. The num in A is private; therefore, a public member function getnum is written which returns the value of num. The function getnum is called in the show method of class B. In the main function, all functions are called through an object of class B.

```
/*PROG 8.4 AREA AND VOLUME CALCULATION USING INHERITANCE */

import java.util.*;
import java.io.*;
class Area
{
    protected int length, width;
    public void input1()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the length and width");
        length = sc.nextInt();
        width = sc.nextInt();
    }
}
class Volume extends Area
{
    private int height;
    public void input2()
    {
        input1();
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the height");
        height = sc.nextInt();
    }
    void show()
    {
        System.out.println("Length      := "+length);
        System.out.println("Width      := "+width);
        System.out.println("Area       := "+length * width);
        System.out.println("Height     := "+height);
        System.out.println("Volume     := "+length*width* height);
    }
}
class JPS6
{
    public static void main(String args[])
    {
        Volume v = new Volume();
        v.input2();
        v.show();
    }
}
```

OUTPUT:

```
Enter the length and width
2
2

Enter the height
2

Length      := 2
Width       := 2
Area        := 4
Height      := 2
Volume      := 8
```

Explanation: In the class area, there are two protected data members: length and width. Function input1 takes input from keyboard into these data items directly. The class Volume inherits the class Area. The class Volume contains one data member: height. The class calculates area by using the inherited data members, length and width, and finds volume with the aid of height with length and width. Note that only the derived class function input2 is called inside the main function. This function in turn calls input1 of the Area class and takes input from the keyboard. The function show calculates the area and volume and displays the same.

```
/*PROG 8.5 CONCATENATING TWO STRING */

import java.io.*;
import java.util.*;
class first
{
    private String name;
    void input_f()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the name ");
        name = sc.next();
    }
    String getname()
    {
        return name;
    }
}
class second extends first
{
    private String sname;
    void input_s()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the name");
        sname = sc.next();
    }
    void show()
    {
```

```

        String con;
        con = getname() + " " + sname;
        System.out.println("name := " + getname());
        System.out.println("sname := " + sname);
        System.out.println("\n*****");
        System.out.println("Joint name := " + con);
        System.out.println("*****");
    }
}
class JPS7
{
    public static void main(String[] args)
    {
        second obj = new second();
        obj.input_f();
        obj.input_s();
        obj.show();
    }
}

OUTPUT:

Enter the name
HARI
Enter the name
PANDEY
name := HARI
sname := PANDEY
*****
Joint name := HARI PANDEY
*****

```

Explanation: The class first takes one string from the console and stores it in the private data member ‘name’. To access this string name in the derived class ‘second’, a public member function `getname` is written in base class ‘first’. In the main function, the class second first calls the `input_f` method which initializes the string name of class first. The method call `input_s` initializes the string `sname` of second class. In the show method of class second, the private data member ‘name’ is obtained by a call to `getname` method, and `sname` is concatenated with it.

```
/*PROG 8.6 BEHAVIOUR OF PROTECTED MEMBERS IN INHERITANCE */
```

```

class A
{
    protected int pa;
}
class B extends A
{
    protected int pb;
}
class JPS8

```

```

{
    public static void main(String[] args)
    {
        B obj = new B();
        obj.pa = 20;
        obj.pb = 30;
        System.out.println("pa = " + obj.pa);
        System.out.println("pb = " + obj.pb);
    }
}

OUTPUT:
pa = 20
pb = 30

```

Explanation: In Java, protected members can be accessed outside the class and are inherited to derived class also.

```

/*PROG 8.7 DEMO OF TWO LEVEL INHERITANCE */

class first
{
    public void show_f()
    {
        System.out.println("\nHello from first");
    }
}
class second extends first
{
    public void show_s()
    {
        System.out.println("\nHello from second");
    }
}
class third extends second
{
    public void show_t()
    {
        show_f();
        show_s();
        System.out.println("\nHello from third\n");
    }
}
class JPS9
{
    public static void main(String[] args)
    {
        third t = new third();
        t.show_t();
    }
}

```

OUTPUT:

```
Hello from first
Hello from second
Hello from third
```

Explanation: The above program demonstrates two levels of inheritance. The class first is the base class (parent class) for class second (child class) and, class second is the base class (parent class) for class third (child class). In turn, class first is the grandfather of class third (grand child). In `show_t` function of class third, `show_s` is inherited directly from class second but `show_f` is inherited directly from class first through class second.

8.4 METHOD OVERRIDING

In method overriding, a base class method is overridden in the derived class. That is, the same method is written with the same signature as of the base class method but different implementation. Method overriding is often used in inheritance to override the base class implementation of the method and provide its own implementation. In method overloading, arguments and type of arguments are used to distinguish between two functions, but in method overriding no such distinction can be made. In method overriding, the signature of the two methods must match. This is shown in the program given below.

```
/*PROG 8.8 DEMO OF METHOD OVERRIDING */

class A{
    void show()
    {
        System.out.println("\nHello from A");
    }
}
class B extends A
{
    void show()
    {
        System.out.println("\nHello from B ");
    }
}
class JPS10
{
    public static void main(String args[])
    {
        B obj = new B();
        obj.show();
    }
}
OUTPUT:
Hello from B
```

Explanation: In the class B, the function `show` is overridden. The function `show` of class A is now hidden. From the main function, the statement `obj.show()`, on execution, calls the `show` function of the class B.

8.5 DYNAMIC METHOD DISPATCH

Binding is the process of linking the function call with the place where function definition is actually written so that when a function call is made, it can be ascertained where the control has to be transferred. Binding is also called linking, and is of two types:

1. Static binding
2. Dynamic binding.

When at compile time, it is known which function will be called in response to a function call, the binding is said to be static binding, compile time binding or early binding. Static binding is called so because before the program executes, it is fixed that a particular function be called in response to a function call. Each time the program executes, the same function will be called. As the linking is done at compile time, early to the execution of the program, it is also known as compile time binding or early binding.

When it is not ascertained as to which function is to be called in response to a function call, binding is delayed till the program executes. At run time, the decision is taken as to which function is to be called in response to a function call. This type of binding is known as late binding, runtime binding or dynamic binding. Dynamic binding is purely based on finding the addresses of pointers, and as addresses are generated during run time or when the program executes, this type of binding is known as run time or execution time binding.

One type of polymorphism seen earlier is compile time polymorphism, which is of two types: function polymorphism/overloading and operator overloading.

Another type of polymorphism is the run time polymorphism, in which at run time, it is decided, which function is to be called by checking the pointer and its contents. It is necessary in situations, where there are two functions with the same name in both derived class and base class. At run time, it is decided using references and objects, as to which function of which class is to be called.

Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than at compile time. It is the mechanism through which Java implements run time polymorphism. Before understanding this, it is important to understand the concept of Java in inheritance: '**A super class reference can refer to a derived class object**'. For example consider the code:

```
class A { }
class B extends A{ }
B b = new A ();           is perfectly valid
```

In this code, the reference is of B class but it is referring to a derived class object. Note that the reverse is not true.

The **dynamic method dispatch** can now be understood. When a reference of super class is used for calling a method at run time (which is overridden in derived class), it is checked as to which particular object the reference is referring to. It is possible that a super class reference might be referring to its own object or an object of its derived class. Now at run time, the method is called by checking the contents of the reference as to which type of object it is referring to and not by checking what type it is.

Consider the following code:

```
class A
{
    void show()
    {
        System.out.println("Hello from show of A");
    }
}
```

```

class B extends A
{
    void show()
    {
        System.out.println("Hello from show B");
    }
}
class C extends A
{
    void show()
    {
        System.out.println("Hello from show of C");
    }
}

```

In the above code, the `show` function of parent class A is overridden in class B and class C. Now when the function `show` is called as given below, the following shows which version of the `show` is called.

```

A ref = new A();
ref.show();           //show of A is called
ref = new B();
ref.show();           //show of B is called
ref = new C();
ref.show();           //show of C is called

```

Note: We have just used one reference `ref` of super class A and in which storing the objects of different classes. A full program with two methods is given below.

```

/*PROG 8.9 DEMO OF DYNAMIC METHOD DISPATCH */

class A
{
    void show()
    {
        System.out.println("\nHello from show of A");
    }
    void display()
    {
        System.out.println("Hello from display of A ");
    }
}
class B extends A
{
    void show()
    {
        System.out.println("\nHello from show of B");
    }
}
class C extends A

```

```

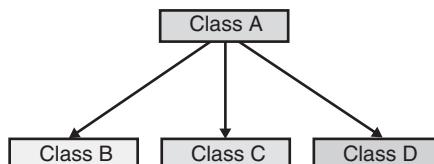
{
    void display()
    {
        System.out.println("Hello from display of C");
    }
}
class JPS11
{
    public static void main(String[] args)
    {
        A ref = new A();
        B b = new B();
        C c = new C();
        ref.show();
        ref.display();
        ref = b;
        ref.show();
        ref.display();
        ref = c;
        ref.show();
        ref.display();
    }
}
OUTPUT:
Hello from show of A
Hello from display of A
Hello from show of B
Hello from display of A
Hello from show of A
Hello from display of C

```

Explanation: In class A, two methods, `show` and `display`, are defined. The function `show` is overridden in class B but `display` is not. In the class C method, `display` is overridden but `show` is not. In the main function, reference `b` of class B is assigned to `ref`, which is a reference of super class A. Now when `show` and `display` are called, `show` of class B is called as it is overridden in class B and contents is the object of class B. The `display` method is not found in the class B so interpreter searches it in its super class A. It finds it there, and `display` of class A is called. Similarly when reference `c` of class C is assigned to `ref` and when `show` and `display` are called, `show` of class A and `display` of class C are called.

8.6 HIERARCHICAL INHERITANCE REVISITED

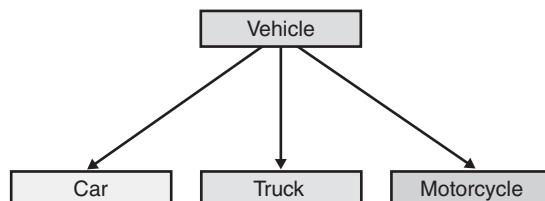
In this type of inheritance, one class can be a parent of many other classes. For example:



Here class A is the base class for all three classes: B, C and D. They can share all fields, methods of class A except private members.

Suppose that a program has to deal with motor vehicles, including cars, trucks and motorcycles. The program could use named Vehicles to represent all types of vehicles. The Vehicles class could include instance variables, such as registration Number and owner, and instance method, such as `transferOwnership()`. These are variables and methods common to all vehicles. Three subclasses of Vehicle—Car, Truck and Motorcycle—could then be used to hold variables and methods specific to particular types of vehicles. The Car class might add an instance variable ‘numberOfDoors’, the Truck class might have ‘numberOfAxels’ and the Motorcycle class could have a Boolean variable ‘hasSidecar’. The declarations of these classes in Java program would look, in outline, like the following:

```
class Vehicle
{
    int registrationNumber;
    String owner;
    void transferOwnership(String newOwner)
    {
        //function codes;
    }
    //other class stuff;
}
class Car extends Vehicle
{
    int numberOfDoors;
    //other class stuff;
}
class Truck extends Vehicle
{
    int numberOfAxels;
    //other class stuff;
}
class Motorcycle extends Vehicle
{
    boolean hasSidecar;
    //other class stuff;
}
```



Suppose that `myCar` is a variable of type `Car` that has been declared and initialized with the statement.

```
Car myCar=new Car();
```

Given this declaration, a program could refer to ‘myCar.numberOfDoors’, since ‘numberOfDoors’ is an instance variable in the class Car. But since class Car extends class Vehicle, a car also has all the structures and behaviour of a Vehicle. This means that ‘myCar.registrationNumber’, ‘myCar.owner’ and ‘myCar.transferOwnership()’ also exist.

Now, in the real world, cars, trucks and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type Car or truck or Motorcycle is automatically an object of type Vehicle. This brings out the following important facts: *A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.*

The practical effect of this in the example is that an object of type Car can be assigned to a variable of type Vehicle. That is, it would be legal to write the following:

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable ‘myVehicle’ holds a reference to a Vehicle object that happens to be an instance of the subclass, Car. The object remembers that it is in fact a Car and not just a Vehicle. The information about the actual class of an object is stored as part of that object.

On the other hand, if ‘myVehicle’ is a variable of type Vehicle, the assignment statement

```
myCar= myVehicle;
```

would be illegal because ‘myVehicle’ could refer to other types of vehicles that are not cars.

8.7 THE SUPER KEYWORD

Whenever the base class version of the function is called which is overridden in the derived class, the super keyword can be used. The super keyword always refers to the immediate base class. For example, consider the following code:

```
class A
{
    void show()
    {
        System.out.println("Hello from show of A");
    }
}
class B extends A
{
    void show()
    {
        show(); //will lead to recursion
        System.out.println("Hello from show of B");
    }
}
```

In the above case, inside the show of class B, if show (as written) is simply written, it will call itself, and recursion will follow. To call the base class version, super.show can be written, which means show of class A will be called. Super can also be used to any of the method or field of the base class or for calling constructor. See the example given below. More examples can be found later in this chapter.

```
/*PROG 8.10 DEMO OF SUPER VER 1*/
class A
{
    int num;
    void show()
    {
        System.out.println("\nHello from A num := " + num);
    }
}
class B extends A
{
    int num = 80;
    void show()
    {
        super.num = 50;
        super.show();
        System.out.println("\nHello from B num := " + num);
    }
}
class JPS12
{
    public static void main(String args[])
    {
        B obj = new B();
        obj.show();
    }
}
OUTPUT:
Hello from A num := 50
Hello from B num := 80
```

Explanation: The expression `super.num` represents `num` of the immediate base class, i.e., `class A`. Similarly `super.show` represents `show` of `class A`.

8.8 CONSTRUCTOR AND INHERITANCE

This section discusses how the constructors are called when they are present both in base and derived classes, and how the values are passed from derived class to base class. Assume a small example of single level inheritance in which `class A` is inherited by `class B`. Both the classes have their default constructors. When an object of `class B` is created, it calls the constructor of `class B`, but as `class B` has got `A` as its parent class, constructor of `class A` will be called first, followed by that of `class B`. This is so because when derived class has inherited base class, obviously it will be using the data member from base class. Now without calling the constructor of base class, data members of base class will not be initialized and if derived class uses the uninitialized data members of base class, unexpected result may follow. Calling a constructor of base class first allows base class to properly set up its data members so that they can be used by derived classes.

```
/*PROG 8.11 DEMO OF CONSTRUCTOR IN INHERITANCE VER 1 */
```

```
class first
{
    first()
    {
        System.out.println("\n Con of first called");
    }
}
class second extends first
{
    second()
    {
        System.out.println(" Con of second called");
    }
}
class JPS13
{
    public static void main(String[]args)
    {
        second obj = new second();
    }
}
OUTPUT:
Con of first called
Con of second called
```

Explanation: The new second() causes constructor of second class to be called. But as first is the base class of second class, constructor of first class will be called first, followed by that of second class.

```
/*PROG 8.12 CONSTRUCTOR AND MULTILEVEL INHERITANCE */
```

```
class first{
    first()
    {
        System.out.println(" \nCons of first called");
    }
}
class second extends first{
    second()
    {
        System.out.println(" \nCons of second called ");
    }
}
class third extends second
{
    third()
    {
```

```

        System.out.println(" \nCons of third called ");
    }
}
class JPS15
{
    public static void main(String[] args)
    {
        third obj = new third();
    }
}
OUTPUT:
Cons of first called
Cons of second called
Cons of third called

```

Explanation: Class second is the base class of class third and class first is the base class of class second. So, constructor of classes first, second and third are called in order.

```
/*PROG 8.13 PARAMETERIZED CONSTRUCTOR WITHIN INHERITANCE VER 1 */
```

```

class first
{
    private int fa;
    first(int x)
    {
        fa = x;
        System.out.println("\nCons of first called ");
    }
    void fshow()
    {
        System.out.println("fa= " + fa);
    }
};
class second extends first
{
    private int sa;
    second(int a, int b)
    {
        super(a);
        sa = b;
        System.out.println("\nCons of second called");
    }
    void sshow()
    {
        fshow();
        System.out.println("sa = " + sa);
    }
}
class JPS16

```

```

{
    public static void main(String[] args)
    {
        second s = new second(20, 30);
        s.sshow();
    }
}

OUTPUT:
Cons of first called
Cons of second called
fa = 20
sa = 30

```

Explanation: When statement `second s = new second (20, 30);` executes, object `s` calls the parameterized constructor of class `second` and passes parameters 20 and 30. Inside the constructor, they are assigned to `a` and `b`, respectively. Before entering into the body of constructor, `super(a)` calls the constructor of class `first` and passes `a` as argument. Inside the class `first`, it is assigned to `fa` and `con of first called` is displayed. When control returns, it enters into the body of the constructor and assigns `b` to `sa`. When `sshow` of class `second` is called, it first calls the `fshow` of class `first`. Again note that `super(a)` is the first statement in the constructor of the derived class.

In the earlier program, there are two `int` parameterized constructor in the derived class but this is not necessary. We can also have a single `int` argument constructor and yet passing a single `int` value by doing same manipulation over the single `int` data. As long as the required type and number of parameters are passed in the super class constructor, no problem arises, no matter where they have come from. For example, consider a section of the code rewritten from the above program:

```

second (int a)
{
    super(a%10);
    sa=a;
    System.out.println("Cons of second called");
}

```

In the main, the constructor can be used as follows:

```
second s = new second (46);
```

The above call assigns 6 to `fa` of `first` class (due to $46 \% 10$) and 46 to `sa`.

```
/*PROG 8.14 PARAMETERIZED CONSTRUCTOR WITHIN INHERITANCE VER 2 */
```

```

class first
{
    int num;
    first(int x)
    {
        num = x;
        System.out.println(" \nCons of first called");
        System.out.println(" num := " + num);
    }
}

```

```

}

class second extends first
{
    String str;
    second(int d, String s)
    {
        super(d);
        str = s;
        System.out.println("\nCons of second called");
        System.out.println("str := " + str);
    }
}
class third extends second
{
    private char cnum;
    third(int d, String s, char c)
    {
        super(d, s);
        cnum = c;
        System.out.println("\nCons of third called ");
        System.out.println("cnum := " + c);
    }
}
class JPS17
{
    public static void main(String[] args)
    {
        new third(2345, "Demo", 'J');
    }
}

OUTPUT:

Cons of first called
num := 2345

Cons of second called
str := Demo

Cons of third called
cnum := J

```

Explanation: In parameterized constructor with inheritance, the derived class constructor must provide initial values to the constructors of the base class. In the program, class `second` is the base class for class `third` and class `first` is the base class for class `second`. When constructor of class `third` is called from the `main` it is written as follows:

```
new third(2345, "Demo", 'J');
```

where `2345` is assigned to `d`, “`Demo`” is assigned to `s` and ‘`J`’ is assigned to `c`. As `second` class is the base class of class `third`, the first line `super(d, s)` calls the constructor of the base class `second` and passes arguments `d` and `s`. Inside the constructor of `second` class, `d` and `s` are assigned to `d` and `s`. Now as class `first` is the base class for class `second`, the first line in the constructor of `second` class `super`

(d) calls constructor of class `first` and passes `d` as argument. Inside the constructor of class `first`, this `d` is copied to `x` which is assigned to `num`. Control returns back to constructor of second class and `str = s` executes. When constructor of class `second` returns, it returns to the statement `cnum = c`. Rest is simple to understand.

Note: In inheritance, in the constructor of the derived class, if you are using the `super` keyword then it must be the first statement inside the constructor of the derived class.

8.9 OBJECT SLICING

Object slicing is a process of removing off the derived portion of the object when an object of derived class is assigned to a base class object. Only the base class data are copied to derived class object. Consider the following two classes:

```
class A
{
    int Ax, Ay;
}
class B extends A
{
    int Bx;
```

Through inheritance, data members `Ax` and `Ay` of class `A` are copied to `class B`; when in the main it is written as follows:

```
B b1 = new B();
A a1 = new A();
a1 = b1;
```

Only the data members of object `b1`, which were inherited from `class A`, are assigned to object `a1`. The data member `Bx` of object `b1` is not copied. In other words, object `b1` was sliced off.

```
/*PROG 8.15 DEMO OF OBJECT SLICING VER 1*/
class demo
{
    void show_demo()
    {
        System.out.println("\nHello from show of demo");
    }
}
class der_demo extends demo
{
    void show_der()
    {
        System.out.println("Hello from show of der_demo");
    }
}
class JPS18
```

```
{
    public static void main(String[] args)
    {
        demo d1 = new demo();
        der_demo d2 = new der_demo();
        d1 = d2;
        d1.show_der();
    }
}
```

OUTPUT:

```
C:\JPS\ch8>javac JPS18.java
JPS18.java:23: cannot find symbol
symbol : method show_der()
location: class demo
        d1.show_der();
               ^
1 error
```

Explanation: In the main function, when an object d2 of derived class der_demo is assigned to an object d1 of class demo, only the inherited portion of base class is assigned to d1. Although functions are usually not part of the object yet a function of one class can be called from object of that class or from objects of derived class. Here, from d1, the function show_der of derived class cannot be called.

```
/*PROG 8.16 DEMO OF OBJECT SLICING VER 2 */
```

```
class demo
{
    int bx, by;
    demo(int x, int y)
    {
        bx = x;
        by = y;
    }
    demo() { }
};

class der_demo extends demo
{
    int dx;
    der_demo(int x, int y, int z)
    {
        super(x, y);
        dx = z;
    }
}
class JPS19
{
    public static void main(String[] args)
    {
        der_demo d2 = new der_demo(10, 20, 30);
```

```

        demo d1 = new demo();
        d1 = d2;
        System.out.println(d1.bx + "\t" + d1.by + "\t" +
                           d1.dx);
    }
}

OUTPUT:

C:\JPS\ch8>javac JPS19.java
JPS19.java:28: cannot find symbol
symbol  : variable dx
location : class demo
System.out.println(d1.bx + "\t" + d1.by + "\t" + d1.dx);
                                         ^
1 error

```

Explanation: The class demo consists of two data members bx and by and class der_demo with just one data member dx. When in the main function, d1 = d2 is written, only the inherited data members bx and by are assigned to object d1. The data member dx from d2 is not assigned to d1. In effect, object d2 gets sliced. So dx cannot be called from an object of demo class.

8.10 FINAL CLASS

A final class is a class which is declared final by placing the keyword `final` before class definition. A final class has the property that it cannot be inherited. One example of final class is `System` class defined in package **java.lang**.

When you say that an entire class is final, you state that you do not want to inherit from this class or allow anyone else to do so. In other words, for some reason, the design of your class is such that there is never a need to make any changes, or for safety or security reasons, you do not want sub-classing.

Note that the fields of a final class can be final depending on how you write. The same rules apply to final for fields regardless of whether the class is defined as final. However, because it prevents inheritance, all methods in a final class are implicitly final since there is no way to override them. An example follows:

```

final class demo
{
    float num = 45.50f;
    String F = "Final";
    void fix() { }
}

```

8.11 ABSTRACT CLASS

An abstract class is the one that is not used to construct objects, but only as a basis for making subclasses. It exists only to express the common properties of all its subclasses. An abstract class is a class that leaves one or more method implementations unspecified by declaring one or more method abstract. An abstract method has nobody, i.e., no implementation (partially true). A subclass is required to override the method and provide an implementation. Hence, an abstract class is incomplete and cannot be instantiated, but can be used as a base class.

An abstract class usually has one or more abstract methods. This is a method that is incomplete; it has only a declaration and no method body. The syntax for an abstract method declaration is as follows:

```
abstract void fun();
```

A class containing abstract methods is called an abstract class. If a class contains one or more abstract methods, the class itself must be qualified as abstract. (Otherwise, the compiler gives an error message.) Objects of an abstract class cannot be created so an abstract class must be inherited by some other class. The derived class must provide the implementation of all abstract methods declared in the abstract class. If the derived class does not provide implementation for the abstract functions, it must be declared as abstract.

To create a class as abstract, `abstract` keyword simply needs to be written before the `class` keyword as follows:

```
abstract class demo
{
    optional abstract method signatures;
}
```

It is possible to create a class as abstract without including any abstract methods. This is useful when there is a class in which it does not make sense to have any abstract methods, and yet any instances of that class is required to be prevented.

If an abstract class is incomplete, what is the compiler supposed to do when someone tries to make an object of that class? It cannot safely create an object of an abstract class so one gets an error message from the compiler. This way, the compiler ensures the purity of the abstract class, and one need not worry about misusing it. Creating a reference of an abstract class is perfectly valid. A reference can hold the objects of other derived classes but can only functions which are part of the base class. A reference cannot call any function of the derived class. This will be shown later on.

If one inherits from an abstract class and wants to make objects of the new type, method definitions must be provided for all the abstract methods in the base class. Otherwise (and one may choose not to), the derived class is also abstract, and the compiler will force one to qualify that class with the `abstract` keyword.

In short, note the following points about abstract class:

1. An abstract class is declared using the keyword `abstract`.
2. It may or may not contain abstract methods.
3. An abstract class must be inherited by some other class.
4. If the derived class does not provide the implementation of any abstract method present in an abstract class, the derived class must be declared as abstract.
5. Objects of an abstract class cannot be created, but reference can be created.
6. Through reference of abstract class, only methods of the abstract class implemented in the derived class can be called.
7. `Abstract` keyword cannot be applied to static methods or constructor.

A few example programs of abstract class are given below.

```
/*PROG 8.17 DEMO OF ABSTRACT CLASS VER 1 */
```

```
abstract class Base
{
    abstract void show();
}
class first extends Base
```

```

{
    void show()
    {
        System.out.println("\n\nshow of first");
    }
}
class second extends Base
{
    void show()
    {
        System.out.println("\nShow of second");
    }
}
abstract class third extends Base
{
}
class JPS20
{
    public static void main(String[] args)
    {
        Base ref = new first();
        ref.show();
        ref = new second();
        ref.show();
    }
}
OUTPUT:
show of first
Show of second

```

Explanation: In the program, an abstract class `Base` is created. In the class, we have an abstract method `demo`, which all derived classes must give the implementation. The class `first` and `second` inherit the `Base` class and provide the implementation of `show` method. The class `third` does not provide the implementation, so it must be declared as abstract.

In the `main` function, a reference of class `Base` named `ref` is created. In this `ref`, the object of class `first` is assigned and call the `show` was called. An object of class `first` can also be created and the `show` called as follows:

```
first f = new first();
f.show();
```

The same reference `ref` of `Base` class is then used for `second` class object and `show` is called. Note it would be an error to try to instantiate an object of an abstract type.

```
Base obj = new Base (); // ERROR
```

That is, operator `new` is invalid when applied to an abstract class.

| |
|---|
| /*PROG 8.18 DEMO OF ABSTRACT CLASS VER 2 */ |
|---|

```

abstract class Base
{
    void show()
```

```

        {
            System.out.println("\nShow of base");
        }
    }
class first extends Base
{
    void show()
    {
        super.show();
        System.out.println(" \nshow of first");
    }
}
class JPS21
{
    public static void main(String args[])
    {
        Base ref = new first();
        ref.show();
        ref = new second();
        ref.show();
    }
}
OUTPUT:
Show of base
show of first
Show of second

```

Explanation: It was mentioned earlier that an abstract class can have non-abstract methods also. In the program in class Base, there is a method show which is having the body and not abstract. But being an abstract class, an object of this class cannot be instantiated. So this class must be inherited by some other class, and the object of that class can call the method show.

The class first inherits the class Base and overrides the show method. The base class version of show is called by using the super keyword. The class second simply inherits the class Base and is empty. In the main function, we create reference of class Base is created and the method show was called. Note, ref.show calls the method show of class first and not of its own, as the object stored is of class first. Second time, the ref holds the reference of an object of class second. When the method show is called as ref.show, first, the show method is searched in the class second. When it does not find one there, it looks for the method in the Base class. It finds the method there and calls it.

```

/*PROG 8.19 SHOWING THE FLYING STATUS OF VARIOUS INSECTS */

abstract class Insect
{
    public abstract void flystatus();
}
class Cockroach extends Insect
{
    public void flystatus()
    {
        System.out.println("\nCockroach can fly");
    }
}

```

```

        }
    }
class Termite extends Insect
{
    public void flystatus()
    {
        System.out.println("\nTermite cannot fly");
    }
}
class Grasshopper extends Insect
{
    public void flystatus()
    {
        System.out.println("\nGrasshopper can fly");
    }
}
class Ant extends Insect
{
    public void flystatus()
    {
        System.out.println("\nAnt cannot fly");
    }
}
class JPS24
{
    public static void main(String[] args)
    {
        Insect ptr[] = new Insect[4];
        ptr[0] = new Cockroach();
        ptr[1] = new Termite();
        ptr[2] = new Grasshopper();
        ptr[3] = new Ant();
        for (int i = 0; i < ptr.length; i++)
            ptr[i].flystatus();
    }
}

OUTPUT:
Cockroach can fly
Termite cannot fly
Grasshopper can fly
Ant cannot fly

```

Explanation: The abstract class Insect declares one abstract function flystatus. Any class which inherits this class has to redefine this function and tell his/her flying status, i.e., whether he/she can fly or not. The class Insect inherits four different classes: Cockroach, Termite, Grasshopper and Ant. Each class does provide implementation of function flystatus. In the main function, an array of references of class Insect type is created. In each reference of this array, the objects of various derived classes are assigned as shown below:

```

ptr[0] = new Cockroach();
ptr[1] = new Termite();

```

```
ptr[2] = new Grasshopper();
ptr[3] = new Ant();
```

In the for loop, flystatus function is called using this Insect array ptr, respective flystatus functions of each class are called.

```
/*PROG 8.20 DEMO OF ABSTRACT CLASS VER 5 */
```

```
abstract class Figure
{
    abstract double area();
    abstract int tellsides();
    double s1, s2;
    Figure(double a, double b)
    {
        s1 = a;
        s2 = b;
    }
}
class Rectangle extends Figure
{
    final int sides = 4;
    Rectangle(double a, double b)
    {
        super(a, b);
    }
    int tellsides()
    {
        return sides;
    }
    double area()
    {
        return s1 * s2;
    }
}
class Triangle extends Figure
{
    final int sides = 3;
    Triangle(double a, double b)
    {
        super(a, b);
    }
    int tellsides()
    {
        return sides;
    }
    double area()
    {
        return s1 * s2 / 2;
    }
}
```

```

class JPS25
{
    public static void main(String[] args)
    {
        Figure ref = new Rectangle(10, 15);
        double ar_fig = ref.area();
        int s = ref.tellssides();
        System.out.println("\nArea of Rectangle := " + ar_fig);
        System.out.println(" Number of sides := " + s);
        ref = new Triangle(5, 4);
        ar_fig = ref.area();
        s = ref.tellssides();
        System.out.println("Area of Triangle := " + ar_fig);
        System.out.println(" Number of sides := " + s);
    }
}

```

OUTPUT:

```

Area of Rectangle      := 150.0
Number of sides  := 4
Area of Triangle := 10.0
Number of sides  := 3

```

Explanation: In the program, there is an abstract class figure. The class has two data members, s1 and s2, which stand for two sides of the figure. The class has two abstract methods: area and tellssides. Each geometrical figure which inherits this class must calculate the area and tell the number of sides it has. The class also has constructor which takes two parameters of double type and assign the values to s1 and s2.

The class is inherited by two classes, Rectangle and Triangle, which provide the implementation of area and tellside. Both use a final member side, which represents number of sides a figure has.

In the main object of class, Rectangle and Triangle are created using constructors, and dimension of sides is passed. In both the calls, Rectangle (10, 15) and Triangle (5, 4), constructor of Figure class is called and s1 and s2 get their values. Note that references to object of Rectangle and Triangle class are handled using a reference to Figure class. Rest is simple to understand.

8.12 PONDERABLE POINTS

1. Java does not support multiple inheritance.
2. For inheriting a class to another class, Java uses the extend keyword.
3. In Java, there is no public, private or protected inheritance.
4. Method overriding is redefining the function of base class in derived class with the same signature.
5. A base class reference can hold the reference of a derived class object.
6. The abstract keyword makes a class abstract whose object cannot be instantiated but references can be.
7. An abstract class must be inherited by another class to make use of it.
8. An abstract class can have abstract as well as non-abstract methods in it.
9. An abstract method must not have body.

REVIEW QUESTIONS

1. What is the difference between composition (has-a) and inheritance (is-a)?
2. What is the difference between overriding and overloading a method?
3. Describe the syntax of single inheritance in Java.
4. What is inheritance and what are its uses?
5. What are the rules for abstract class?
6. When do we declare a variable, method, or class final?
7. When do we declare a method or class abstract?
8. Discuss the different level of access protection with example.
9. Create a base class called Shape, it contain two methods `getxyvalue()` and `showxyvalue()` for accepting co-ordinates and to displaying the same. Create a subclass called Rectangle. It also contains a method to display the length and breadth of the rectangle called `showxyvalue()`. Use the overriding concept.
10. Write a class called TV with the following attribute. Name of the company and Screen size using super keyword and Inheritance to create a color TV class, it has a attribute TVtype also a BW TV class it is also have TVtype attribute in Boolean data type.
11. Write a program that creates an abstract class called dimension. Create two subclasses, rectangle and triangle, that include appropriate methods for both the subclasses that calculate and display the area of the rectangle and triangle.
12. Write a program that has an overloaded method. The first method should accept no arguments, the second method will accept one string and the third method should display the message “Welcome to java” once. The second method should display the message “Welcome to polymorphism” twice and the third method should display the message “Welcome to overloading” three times.
13. What message does Java give when you put an abstract method inside a class which is not, in itself, declared abstract?
14. Why an abstract method cannot be defined as final? Justify with example.
15. Why an abstract method cannot be defined as static? Justify with example.
16. Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class publication that stores the title (a string) and price (type float) of publication. From this class, derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float). Each of these three classes should have a `getdata()` function to get its data from the user as the keyboard, and a `putdata()` function to display its data.
Write a `main()` program to test the book and tape classes by creating instance of them, asking the user to fill in data with `getdata()`, and then displaying data with `putdata()`.
17. Start with the publication, book and tape classes of Exercise-16. Add a base class sales that holds an array of three floats so that it can record the dollar sales of a particular publication for the last three months. Include a `getdata()` function to get three sales amounts from the user, and a `putdata()` function to display the sales figures. After the book and tape classes they are derived from both publication and sales. An object of class book or tape should input and output sales data along with its other data. Write a `main()` function to create a book object and a tape object and exercise their input/output capabilities.
18. Declare a class Vehicle. From this class derive two_wheeler, three_wheeler and four_wheeler class. Display properties of each type of vehicle using member function of the class.

Multiple Choice Questions

1. Java does not support
 - (a) single inheritance
 - (b) multilevel inheritance
 - (c) multiple inheritance
 - (d) hierarchical inheritance
2. For inheriting a class to another class Java uses the keyword
 - (a) super
 - (b) this
 - (c) final
 - (d) extends
3. The `super` keyword always refers to the
 - (a) derived class
 - (b) any base class
 - (c) immediate base class
 - (d) none of the above

4. One example of final class is _____ class defined in the package java.lang is
(a) system class (c) abstract class
(b) integer class (d) none of the above
5. Final class has the property that
(a) cannot be inherited
(b) can be inherited
(c) can be inherited publicly
(d) none of the above
6. An abstract class is declared using the keyword `abstract` that
(a) must contain abstract method
(b) cannot contain abstract method
(c) may or may not contain abstract method
(d) none of the above
7. Abstract keyword cannot be applied to
(a) destructor
(b) static methods or constructor
(c) derived class
(d) none of the above
8. String `toString()` method is used to
(a) wait on another string of execution
(b) return a string that describes the object
(c) call before an unused object is recycled
(d) none of the above
9. Object `clone()` method is used to
(a) create a new object that is the same as the object being cloned
(b) determine whether one object is equivalent to another
(c) both (a) and (b) are correct
(d) none of the above
10. The code given below
- ```
class demo1{ }
class demo2 extends demo1
{
}
demo2 d2 = new demo1();
```
- (a) is not valid  
(b) is perfectly valid  
(c) is valid using abstract class  
(d) none of the above

## KEY FOR MULTIPLE CHOICE QUESTIONS

1. c      2. d      3. c      4. a      5. a      6. c      7. b      8. b      9. a      10. b

# 9

# Packages and Interfaces

## 9.1 INTRODUCTION

A packing is a grouping of related types providing access protection and name space management. Note that all types refer to classes and interfaces. Many a time while working on a small project, one thing that is intended to do is to put all Java files into a single directory. It is quick, easy and harmless. However, if the small project gets bigger and the number of files keeps on increasing, putting all these files into the same directory and managing them would be a difficult task. In Java, this sort of problem can be avoided by using package.

Packages are nothing more than the way files are organized into different directories according to their functionality, usability as well as category they should belong to. A clear example of package is JDK. In JDK, all the packages that can be used in Java programs are part of the Java package, which itself contains a number of packages. In fact, there is a directory named `java` inside which each directory represents a package, e.g., `java.lang`, `java.util`, `java.awt`, `java.net`, etc. The notation `java.lang` means that inside Java package, there is a sub-package lang.

Basically, files in one directory (or package) would have different functionality from those of another directory. For example, files in `java.io` package do something related to I/O, but files in `java.net` package give one the way to deal with the network. In GUI applications, it is quite common to see a directory with a name ‘ui’ (user interface), which keeps files related to the presentation part of the application. On the other hand, a directory called `engine`, stores all files related to the core functionality of the application.

Packaging also helps avoid class name collision when the same class name is used as that of others. For example, in a class name called `Vector`, its name would crash with the `Vector` class from JDK. However, this never happens because JDK uses `java.util` as a package name for the `Vector` class (`java.util.Vector`). So the `Vector` class can be named as `Vector` or it can be put into another package like `com.mycompany.Vector` without fighting with anyone. The benefits of using the package are the ease of maintenance, organization and increase collaboration among developers. Understanding the concept of package will also help one to manage and use files stored in `JAR` files in more efficient ways.

## 9.2 PACKAGE TYPES

Packages are of two types: built-in and user defined.

### 9.2.1 Built-in Packages

The root of all Java packages is the built-in package. Inside this package, a number of other standard packages are defined. It is not possible to discuss all of the packages because of space constraint. Some of the most commonly used packages and a brief description of each is given in Table 9.1.

| Package Name      | Description                                                                                                                                                                                                                                                                                                               |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| java.applet       | Provides classes necessary for the creation of applets and related operations.                                                                                                                                                                                                                                            |
| java.io           | Provides classes for system input and output through data streams, serialization and file manipulation.                                                                                                                                                                                                                   |
| java.lang         | Provides classes that are default and required for fundamental Java programming. This package is default and imported to all Java programs.                                                                                                                                                                               |
| java.net          | Provides classes necessary for implementing networking applications.                                                                                                                                                                                                                                                      |
| java.util         | Contains the collections framework and classes, date and time facilities and miscellaneous utility classes (a string tokenizer, a random-number generator and a bit array, Vector, etc.).                                                                                                                                 |
| java.rmi          | The package and its sub-package provide classes for remote method invocation.                                                                                                                                                                                                                                             |
| java.awt          | Contains all the classes essential for creating user interfaces like buttons, textboxes, panels, etc. and for painting graphics and images.                                                                                                                                                                               |
| java.sql          | Provides the API for accessing and processing data stored source (usually relational database) in RDBMS like MS-Access, SQL Server, or Oracle using Java programming language.                                                                                                                                            |
| java.math         | Provides class for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal). BigInteger is analogous to the primitive integer type of Java except that it provides arbitrary precision. Hence, operations on BigIntegers do not overflow or lose precision. |
| java.text         | Provides classes and interfaces for handling text, dates, numbers and messages in a manner independent of natural languages. These classes are capable of formatting date, numbers and messages; parsing, searching and sorting strings and iterating over characters, words, sentences and line breaks.                  |
| java.awt.event    | Provides interfaces and classes for dealing with different types of events fired by AWT components.                                                                                                                                                                                                                       |
| java.awt.image    | Provides classes and interfaces for creating and modifying images.                                                                                                                                                                                                                                                        |
| java.rmi.registry | Provides a class and two interfaces for the RMI registry.                                                                                                                                                                                                                                                                 |
| java.rmi.server   | Provides classes and interfaces for supporting the server side of RMI.                                                                                                                                                                                                                                                    |
| java.util.zip     | Provides classes for reading and writing the standard ZIP and GZIP file formats.                                                                                                                                                                                                                                          |

**Table 9.1** Some built-in packages defined in Java

All these and other packages can be imported and classes defined within can be used. For this, import statement can be used as in other programs.

### 9.2.2 User-defined Packages

Built-in packages have been used in a number of Java programs. Now the focus is how to create one's own package and use it just like a built-in package.

1. *How to create a package:* Suppose, there is a file called `Hello.java` and this file needs to be put in a package named `Package1`. In order to create a package, a name for the package is chosen and a package statement put with that name at the top of every source file that contains the types (classes, interfaces) that are to be included in the package.

The package statement (e.g., `package pack1`) must be the first line in the source file. There can be only one package statement in each source file and it applies to all types in the file.

So the first thing that is to be done is to specify the keyword package with the name of the package required to be used (Package1 in our case) on the top of the source file before the code that defines the real classes in the package as shown in the Hello class below.

```
package package1;
public class Hello
{
 public static void main(String[] args)
 {
 System.out.println("Hello world");
 }
}
```

The file created must be placed in a directory named Package1 and file name must be Hello.java. The first statement package package1; tells the Java compiler that the file is to be made a part of the package package1. Remember that case is significant and the directory name must match with the package name exactly.

*2. Setting up the classpath and running program:* This variable is used for controlling the access to a specific package. The class search path (commonly known as ‘classpath’) is the path where the Java runtime environment searches for the classes and other resource files. The classpath can be set using either the classpath option when calling a JDK tool (the preferred method) or by setting the CLASSPATH environment variable. In the absence of any package, it is seen that all files and classes reside in one directory and any of the files and classes can be used in other Java programs within the same directory. This is valid, as by default, CLASSPATH variable contains default current working directory in its path which is represented by a dot (.) symbol.

When the number of classes are grouped together under one package and the same package is wanted to be used in a number of Java programs, Java needs to be instructed about the location of that package. Here comes the role of CLASSPATH environment variable. It can be set in the following way:

---

```
set CLASSPATH = classpath1; classpath2
```

---

For .class files in a named package, the classpath ends with the directory that contains the ‘root’ package (the first package in the full package name). The multiple path entries are separated by semicolons. With the set command, it is important to omit spaces from around the equals sign (=). The default classpath is the current directory. Setting the CLASSPATH variable or using the –classpath command-line option overrides that default, so in order to include the current directory in the search path, one must include ‘.’ in the new setting.

It can be understood in continuation of the program stated in the example above.

The package package1 is put under C (i.e., C drive). So the CLASSPATH is set in the following way:

---

```
Set CLASSPATH = .; C:\;
```

---

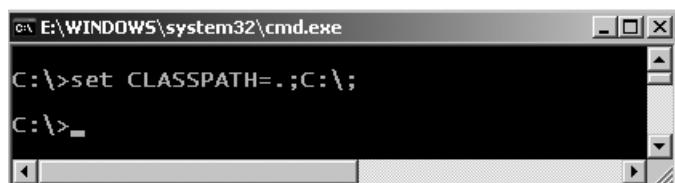
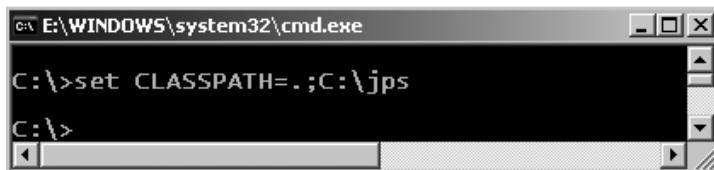
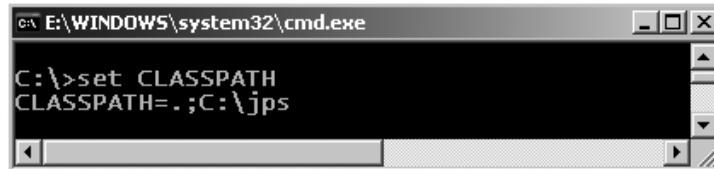


Figure 9.1 Showing the CLASSPATH setting



**Figure 9.2** Showing the CLASSPATH setting in JPS directory



**Figure 9.3** Contents in CLASSPATH

The CLASSPATH is set to point to two places, '.'(dot) and C:\ directory. The . is used as an alias for the current directory and .. for the parent directory. In the CLASSPATH this . is included for convention reason. Java will find the class file not only from C : directory but from the current directory as well. Also, the ; (semicolon) is used to separate the directory location, in case class files are kept in many places.

When compiling the Hello class, go to the package1 directory and type the command.

---

```
C:\JPS\package1>javac Hello.java
C:\JPS\package1>
```

---

If Hello is run using java Hello, the following error message is obtained.

---

```
C:\JPS\package1>java Hello
Exception in thread "main" java.lang.NoClassDefFoundError:
Hello (wrong name: package1>Hello)
```

---

The error message occurs because the Hello class belongs to the package package1. In order to run it JVM must be told about its fully qualified class name pacakge1.Hello instead of its plain class name (Hello).

---

```
C:\JPS\package1>java package1.Hello
Hello world
```

---

**Note:** fully qualified class name is the name of the Java class that includes its package name.

If we do not want to setup the CLASSPATH variable we can use -classpath option while compiling the Java program in the following way:

---

```
C:\JPS\package1>javac -classpath c:\ Hello.java
C:\JPS\package1>
```

---

For running the Java program without setting CLASSPATH variable again the -classpath option can be used in the following way:

---

```
C:\JPS\package1>java -classpath c:\ package1.Hello
Hello world
```

---

This comes handy when the classpath changes frequently or just for the time being when one wants to use the classpath.

The classpath variable can also be set in the following order:

1. Right click on My Computer and then go to properties
2. Click on Advanced Tab and then Environment Variables
3. Under the System Variable Frame heading, click on new and give a new name to the system variable CLASSPATH and set the value as desired
4. Save changes by clicking on OK and come out

To make this example more understandable, the Hello class can be put to system package (package1) can be understand C:\myclasses directory instead. The location of the package is just changed from C:\package1\Hello.java to C:\myclasses\package1\Hello.java. The CLASSPATH then needs to be changed to point out to the new location of the package package1 accordingly.

---

```
Set CLASSPATH = .; C:\myclasses;
```

---

Thus, Java will look for java classes from the current directory and C:\myclasses directory instead. The Hello can be run from anywhere as long as the package package1 is included in the CLASSPATH.

For example, look at the following program:

```
C:>set CLASSPATH=.;C:\
C:>set CLASSPATH
CLASSPATH=.;C:\
C:>cd package1
C:\package1>java package1.Hello
Hello world
C:\package1>cd..
C:>java package1.Hello
Hello world
```

*3. Importing built-in packages:* The method to import built-in packages using import statements has been seen in the earlier programs. The import statement is used for importing classes and related packages into Java programs. When creating one's own package and importing other packages also, the import statement must come after the package statement. The general syntax of import statement is as follows:

---

```
import package1[.package2].(classname|.*);
```

---

The arguments inside [] are optional and arguments within ( ) are compulsory. When you import the whole package in the following way:

---

```
import java.io.*; //implicit import
```

---

it means that you are importing all classes, constants, interfaces, etc. from the package java.io. In case, you just need one class from this package in your program, the whole of the package will be imported to your Java program, which may sometimes increase compilation time. This type of import form is known as **implicit import** statement as all classes are implicitly imported to the Java program. Moreover, in case the class name is explicitly mentioned within the package, the import statement becomes an **explicit import** statement as given below.

---

```
import java.io.DataInputStream; //explicit import
```

Note that there is no static linking in terms of Java and so importing packages implicitly or explicitly does not affect the size of the class and execution of the program.

The standard Java package is Java in which all of the commonly used classes and other package are stored.

**4. Importing user-defined package:** Create a new package demo\_pack by creating a new directory by the name demo\_pack. Assume we are in C: drive. Now move to directory demo\_pack and create a new file in it by the name demo1.java in the following way:

---

```
C:\demo_pack>edit demo1.java
```

---

Type the following program into it.

```
package demo_pack;
public class demo1
{
 public void show()
 {
 System.out.println("Welcome to Package");
 }
 public int sum(int x, int y)
 {
 return x + y;
 }
}
```

In the program, a file demo1 is created, inside which there are two public functions, show and sum, which are simple to understand. Now, compile the file which will create demo1.class file.

In order to import this package and use the class demo1 in the program, create a new file, e.g., main.java and type the following code into it.

```
import demo_pack.demo1;
class main
{
 public static void main(String[] args)
 {
 demo1 d = new demo1();
 d.show();
 int s = d.sum(20, 40);
 System.out.println("Sum = " + s);
 }
}
```

Before compiling the program, set the CLASSPATH and if not set, then set CLASSPATH = .;C:\;

Now, compile and run the program. The output will be as given below.

```
C:\demo_pack>javac main.java
C:\demo_pack>java main
Welcome to Package
Sum = 60
```

Hence, the class `demo1` is accessed and used from package `demo_pack`.

One more package `mypack` in the same drive (C:) and a new file `demo1.java` will be created now.

```
package mypack;
public class demo1{
 public long fact(int num)
 {
 if (num <= 1)
 return 1;
 else
 {
 long f = 1;
 for (int i = 1; i <= num; i++)
 f = f * i;
 return f;
 }
 }
 public int max(int x, int y)
 {
 return x > y ? x : y;
 }
}
```

Compile the file and get the class file.

Note that the class name in both the packages is `demo1`. In a program when both the packages, `demo_pack` and `mypack`, are imported and `demo1` class is to be used from both the packages, how can both the classes in the same program can be distinguished? The solution is that the class name has to be fully qualified using package name as `demo_pack.demo1` and `mypack.demo1`. See the program given below.

```
import demo_pack.*;
import mypack.*;
class JPS
{
 public static void main(String[] args)
 {
 demo_pack.demo1 d1 = new demo_pack.demo1();
 mypack.demo1 d2 = new mypack.demo1();
 int s = d1.sum(10, 20);
 System.out.println("Sum =" + s);
 long f = d2.fact(8);
 System.out.println("Factorial of 8 is " + f);
 int m = d2.max(10, 20);
 System.out.println("Max=" + m);
 }
}
OUTPUT:
Sum =30
Factorial of 8 is 40320
Max=20
```

As a final example, the package for reading different types of data is created from the console and displayed back using own methods. This is stored under the package MYIO. The package consists of just one Java source file name `MYINOUT.java`

Compile the file and set the class path appropriately. All the methods from this class can be used now which will prove quite helpful in writing programs. This package will also be used in various other programs which will be written in this and the next chapters. The class defines six public static methods for reading `integer`, `float`, `double`, `long`, `character` and `String` data types. It also defines two public static methods for displaying string: `show` and `showln` which are equivalent to `System.out.println` and `System.out.print` method, respectively. All the methods are static and so they can be used in the Java program by importing package `MYIO` and using as `MYINOUT.method_name`. Go through the following code:

```
/*PACKAGE MYIO DEVELOPED FOR BASIC INPUT OUTPUT */

package MYIO;
import java.io.*;
public class MYINOUT
{
 static DataInputStream input;
 public static void showln(String str)
 {
 System.out.println(str);
 }
 public static void show(String str)
 {
 System.out.print(str);
 }
 public static int int_val()
 {
 int temp = -1;
 try
 {
 input = new DataInputStream(System.in);
 temp = Integer.parseInt(Input.readLine());
 }
 catch (IOException E)
 {
 System.out.println("Error:" + E);
 }
 return temp;
 }
 public static float float_val()
 {
 float temp = 0.0f;
 try
 {
 input = new DataInputStream(System.in);
 temp=Float.valueOf(input.readLine()).
 floatValue();
 }
 }
}
```

```
 catch (IOException E)
 {
 System.out.println("Error:" + E);
 }
 return temp;
}
public static double double_val()
{
 double temp = 0;
 try
 {
 input = new DataInputStream(System.in);
 temp = Double.valueOf(input.readLine()).
 doubleValue();
 }
 catch (IOException E)
 {
 System.out.println("Error:" + E);
 }
 return temp;
}
public static long long_val()
{
 long temp = -1;
 try
 {
 input = new DataInputStream(System.in);
 temp = Long.parseLong(input.readLine());
 }
 catch (IOException E)
 {
 System.out.println("Error: " + E);
 }
 return temp;
}
public static String str_val()
{
 String str = null;
 try
 {
 input = new DataInputStream(System.in);
 str = input.readLine();
 }
 catch (IOException E)
 {
 System.out.println("Error:" + E);
 }
 return str;
}
public static char char_val()
{
```

```

 char ch = ' ';
 try
 {
 input = new DataInputStream(System.in);
 ch = (char)input.read();
 }
 catch (IOException E)
 {
 System.out.println("Error:" + E);
 }
 return ch;
 }
}

```

**5. Controlling access using packages:** It is understood that a class is the smallest unit of providing encapsulation and abstraction. Inside the class, visibility modifier like `private`, `public` and `protected`, etc., can be used for controlling the access of data to other classes. This notion of access control can be extended to the package also. In package, when a file is created by the name such as `myfile.java` then in the file there can be only one `public` class by the name `myfile`. If you can create a number of `.java` file and can make one class to use other classes within the same package without extending them.

As package is a collection of classes and interfaces; some methods in the package can be made as `public` and some as `private` or `protected`, etc. When this type of access protection is provided for classes inside the packages then depending on what type of access specifier is set for the class or method and where they are being used, Java provides four levels of visibility for class and its data members, which are as follows:

(a) *Subclass in same package:* For example, in our earlier discussion we had `demo1` class in `mypack` package. Now, if we create a new file `demo2.java` and inside it, `demo2` class extends `demo1` class then it will be an example of the category shown below.

```

//file demo2.java
package mypack;
public class demo2 extends demo1
{
 private String name;
 void setName(String s)
 {
 name = s;
 }
 String getName()
 {
 return name;
 }
}

```

In this category, except `private` members of `demo1`, class all other members will be available for use in `demo2` class.

(b) *Non-subclasses in same package:* As this category includes classes from within the same packages in other classes without inheriting them, it is called non-subclasses. This category is similar to the first

category as whether or not it extends classes from within the same package, we are able to use them in other classes except private data members.

(c) *Subclasses in different packages:* Assume that we are in a directory C:\JPS\ch9 and want to inherit the classes defined within package mypack . It means that it is an example of the third category of visibility using packages. Look at the example of the program given below.

```
import java mypack.demo2;
class use extends demo2
{
}
class JPS1
{
 public static void main(String [] args)
 {
 use obj = new use();
 obj.setName("Hari");
 System.out.println(obj.getName());
 System.out.println(obj.max(10,40));
 }
}
OUTPUT:
Hari
40
```

In case use class, extend class demo2 and uses methods of class demo2 from an object class used in the main function. This works fine and the output produced is shown in the program above. In this category, only protected and public data members and methods can be accessed within the use class, while Private and default access is not allowed. Look at one more example where we create a new file demo3.java in the mypack package below.

```
package mypack;
public class demo3
{
 public void pub_show()
 {
 System.out.println("public show");
 }
 private void pri_show()
 {
 System.out.println("private show");
 }
 protected void pro_show()
 {
 System.out.println("protected show");
 }
 void def_show()
 {
 System.out.println("default show");
 }
}
```

In the class, there are four functions each of which displays a string. Each function is tagged as private, protected and public. The fourth function is default without any access specified. Compile the file so that you get .class file. Now, create a new file in C:\JPS\ch9 (assume) and extend the class demo3 in the following way:

```

import mypack.demo3;
class use1 extends demo3
{
 void useshow()
 {
 pro_show();
 }
}
class JPS2
{
 public static void main(String[] args)
 {
 use1 obj = new use1();
 obj.useshow();
 }
}
OUTPUT:
C:\JPS\ch9>java JPS2
protected show

```

In the program, demo3 is extended into class use1. As mentioned above, only public and protected members will be allowed. But note that these protected members would not be allowed outside the use1 class. This being the reason, a function useshow is created and in this method, pro\_show function is called which is protected in demo3 class. If the program is run as given below, it will result in compilation error.

```

import mypack.demo3;
class use extends demo3
{
}
class JPS3
{
 public static void main(String[] args)
 {
 use obj = new use();
 obj.pro_show();
 }
}
OUTPUT
C:\JPS\ch9>javac JPS3.java
JPS3.java:10: pro_show() has protected access in mypack.demo3
 obj.pro_show();
 ^
1 error

```

(d) *Non-subclasses in other packages:* This includes classes which are neither in the same package nor are the derived class of any of the classes defined within the package but yet wants to access classes from the package. In order to understand this, the above class is used into some other package. For this purpose, one can assume to be in C:\JPS\ch9 package and write the program in the following way:

```
import mypack.demo3;
class JPS4
{
 public static void main(String[] args)
 {
 demo3 temp = new demo3();
 //temp.def_show();
 temp.pub_show();
 //temp.pri_show();
 //temp_pro_show();
 }
}
OUTPUT:
C:\JPS\ch9>java JPS4
public show
```

In this category, only the `public` members of the class can be accessed from the package and all commented lines will generate errors.

A complete access protection table using packages is shown in Table 9.2.

| Place                          | Visibility |         |           |        |
|--------------------------------|------------|---------|-----------|--------|
|                                | Private    | Default | Protected | Public |
| Same class                     | A          | A       | A         | A      |
| Same package subclass          | NA         | A       | A         | A      |
| Same package non-subclass      | NA         | A       | A         | A      |
| Different package subclass     | NA         | NA      | A         | A      |
| Different package non-subclass | NA         | NA      | NA        | A      |

A → Available  
NA → Not Available

**Table 9.2** Access protection table for packages

## 9.3 INTERFACES

Methods form the interface of the object with the outside world. The buttons on the front of a television set, for example, are the interface between the viewer and the electrical writing on the other side of its plastic casing. You press the ‘power’ button to turn the television on and off. The operating system like Unix, Linux and Windows form an interface between the PC and the user.

In Java, an interface is similar to an abstract class wherein its members are not implemented. In interfaces, none of the methods are implemented. There is no code associated with an interface. In its most common form, an interface is a group of related methods with empty bodies and constants. An interface is a specification or contract for a set of methods which a class that implements the interface must conform

to, in terms of the type signature of the methods. The class that implements the interface provides an implementation for each method, just as with an abstract method in an abstract class.

So, an interface can be considered as an abstract class with all abstract methods. The interface itself can have either `public`, `package`, `private` or `protected` access defined. All methods declared in an interface are implicitly abstract and implicitly public. It is not necessary and in fact considered redundant to declare a method in an interface to be an abstract. Data can be defined in an interface, but it does not happen usually. If there are data fields defined in an interface, they are implicitly defined as `public`, `static` and `final`.

In other words, any data defined in an interface are treated as public constants. Note that a class and an interface in the same package cannot share the same name. Interface is the only mechanism which allows implementing multiple inheritances in Java.

### 9.3.1 Interface Declaration

An interface is declared by using the keyword `interface` followed by the interface name. For example:

```
interface demo
{
 void fun();
}
```

Before `interface` keyword, you can also specify access specifiers such as `public`, `private`, `protected`, etc. All methods declared inside an interface are by default abstract. The general syntax of creating an interface is as follows:

---

```
access_specifier interface interface_name
{
 return_type method_name1(parameters);
 ;
 ;
 ;
 data_type final constant_name = value;
 ;
 ;
}
```

---

All the variables declared inside an interface are by default considered as `final` and `static`. They can only be used by the class implementing the interface. The implementing class cannot change the constants. The default access specifier is `public` for an interface and the same applies to all its members.

### 9.3.2 Interface Implementation

An interface that is declared can be used by the class. The syntax is as shown below.

---

```
class class_name implements interface_name
{
 //implementing interface's method;
 //own method and data;
}
```

---

An interface declaration is nothing more than a specification to which some classes that wish to implement the interface must conform to in its implementation. This means that a class that implements the interface must define implementations for each of the interface methods.

To make use of an interface, it has to be inherited by some class. An interface can also inherit the interface for extending purpose. For inheriting an interface, either to a class or an interface, ‘implements’ keyword is used. Look at the example below.

---

```
class usedemo implements demo
{
 public void fun()
 {
 System.out.println("Hello from fun");
 }
}
```

---

All methods of interfaces when implementing in the class must be declared `public` otherwise, it may result in compilation error.

Some other class might provide different implementation of method `fun` as given below.

---

```
class temp implements demo
{
 public void fun()
 {
 System.out.println("Using interface demo");
 }
}
```

---

Look at another example below.

---

```
interface Bicycle
{
 void changeGear(int newValue);
 void speedUp(int increment);
 void applyBreakes(int decrement);
}
```

---

In order to implement this interface, the name of the class would change (to `HEROBicycle`, for example) and the keyword `implements` would be used in the class declaration as given below.

---

```
class HEROBicycle implements Bicycle
{
 //implementation of methods of interface;
}
```

---

Interfaces form a contract between the class and the outside world and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all the methods defined by that interface must appear in its source code before the class will compile successfully.

The main use of interface is that one can define some codes in a very general way with a guarantee that by only using the methods defined in the interface, that all objects which implement the interface will have defined implementations for all the methods.

### 9.3.3 Programming Examples

The following program makes use of the interface with class.

```

/*PROG 9.1 DEMO OF INTERFACE VER 1*/

interface i_demo
{
 void show();
}

class first implements i_demo
{
 public void show()
 {
 System.out.println("\nshow of first");
 }
}

class second implements i_demo
{
 public void show()
 {
 System.out.println("\nshow of second");
 }
}

class JPS5
{
 public static void main(String[] args)
 {
 i_demo ref = new first();
 ref.show();
 ref = new second();
 ref.show();
 }
}

OUTPUT:

show of first
show of second

```

*Explanation:* The interface **i\_demo** has just one method **show()**. The **first** and **second** class implements this interface and provides the implementation for the method **show()**. In the main a reference of **i\_demo** is created. First, it gets the reference of an object of the class **first** and then calls the **show()** method. The **show** of class **first** is called. Later on, it gets the reference of an object of the class **second** and calls the method **show()** of the class **second**. Note that as explained before, the methods can also be called by creating reference of the class **first** and **second** using abstract classes as given below.

```

first f = new first();
f.show();

```

and

```

second s = new second();
s.show();

```

```
/*PROG 9.2 DEMO OF INTERFACE VER 2 */

interface i_demo
{
 void show();
}

class first implements i_demo
{
 public void show()
 {
 System.out.println("Show of first");
 }
 void display()
 {
 System.out.println("display of first");
 }
}
class JPS6
{
 public static void main(String[]args)
 {
 i_demo ref=new first();
 ref.show();
 ref.display();
 }
}

OUTPUT:

Compilation Error:
C:\JPS\ch9>javac JPS6.java
JPS6.java:23: cannot find symbol
symbol : method display()
location: interface i_demo
 ref.display();
 ^
1 error
```

*Explanation:* A reference of interface can only call the methods declared in it and implemented by the implementing class. The display is not a part of the interface rather a part of the class first. Therefore, it can only be called by an object reference of the class first. Hence, the code gives error. In order to rectify, the code can be changed as follows:

```
first f = new first();
f.show();
f.display();
```

```
/*PROG 9.3 FINDING COLORS OF VEGETABLE AND WHERE THEY GROW */
```

```
interface Vegetable
{
 void color();
 void wh_grow();
}
class Spinach implements Vegetable
{
 public void color()
 {
 System.out.println("\nColor of spinach is green");
 }
 public void wh_grow()
 {
 System.out.println("Spinach grows above ground");
 }
};
class Potato implements Vegetable
{
 public void color()
 {
 System.out.println("\nColor of Potato is browny
 white");
 }
 public void wh_grow()
 {
 System.out.println("Potato grows under ground");
 }
};
class Onion implements Vegetable
{
 public void color()
 {
 System.out.println("\nColor of Onion is red");
 }
 public void wh_grow()
 {
 System.out.println("Onion grows under ground");
 }
};
class Tomato implements Vegetable
{
 public void color()
 {
 System.out.println("\nColor of Tomato is red");
 }
 public void wh_grow()
 {
```

```

 System.out.println("Tomato grows above ground");
 }
}
class JPS7
{
 public static void main(String []args)
 {
 Vegetable ptr[]=new Vegetable[4];
 ptr[0] = new Spinach();
 ptr[1] = new Potato();
 ptr[2] = new Onion();
 ptr[3] = new Tomato();
 for(int i=0;i<4;i++)
 {
 ptr[i].color();
 ptr[i].wh_grow();
 }
 }
}

OUTPUT:

Color of spinach is green
Spinach grows above ground

Color of Potato is browny white
Potato grows under ground

Color of Onion is red
Onion grows under ground

Color of Tomato is red
Tomato grows above ground

```

*Explanation:* The interface ‘Vegetable’ declares two functions. The functions `color` is used for finding the colour of the vegetable and `wh_grow` for finding where the vegetable grows: underground or above ground. Any class that inherits the vegetable class has to redefine these functions and tell what the colour of vegetable is and where it grows. The interface ‘Vegetable’ is implemented, for example, by four different classes: spinach, potato, onion and tomato. Each class provides the implementation of both the functions. In the `main`, an array of reference `ptr` of interface Vegetable type is created. In each reference of this array, a dynamically created object of various derived classes is created which is shown below.

```

ptr[0] = new Spinach();
ptr[1] = new Potato();
ptr[2] = new Onion();
ptr[3] = new Tomato();

```

In the `for` loop, when functions `color` and `wh_grow` are called using this reference array `ptr`, respective functions, `color` and `wh_grow`, of each class are called.

```
/*PROG 9.4 CHECKING WHETHER SWEET CONTAINS MAWA AS ITS
MAIN INGREDIENT */
```

```
interface Sweet
{
 void mawastatus();
}
class Burfi implements Sweet
{
 public void mawastatus()
 {
 System.out.println("\n Burfi has mawa as" + "\n"
 "one of its main ingredient");
 }
};
class Kajukatli implements Sweet
{
 public void mawastatus()
 {
 System.out.println("\n Kajukatli does not have
 mawa as " + "\n one of its main ingredient");
 }
};
class Jalebee implements Sweet
{
 public void mawastatus()
 {
 System.out.println("\n Jalebee does not have mawa
 as " + "\n one of its main ingredient ");
 }
};
class Rasgulla implements Sweet
{
 public void mawastatus()
 {
 System.out.println("\n Rasgulla does not have Mawa
 as" + "\n one of its main ingredient");
 }
};
class JPS8
{
 public static void main(String[] args)
 {
 Sweet ptr[] = { new Burfi(), new Kajukatli(),
 new Jalebee(), new Rasgulla() };
 for (int i = 0; i < 4; i++)
 ptr[i].mawastatus();
 }
}
```

**OUTPUT:**

```
Burfi has mawa as
one of its main ingredient

Kajukatli does not have mawa as
one of its main ingredient

Jalebee does not have mawa as
one of its main ingredient

Rasgulla does not have mawa as
one of its main ingredient
```

*Explanation:* The interface Sweet declares one function mawastatus. Any class which inherits this class has to redefine this function and tell whether they contain mawa as one of its main ingredients. This interface Sweet is implemented, for example, by four different classes: Burfi, Kajukatli, Jalebee and Rusgulla. Each class does provide implementation of function mawastatus. In the main, an array of reference ptr of interface Sweet type is created. In the reference array ptr, the address of objects of four different classes are assigned.

```
Sweet ptr[] = { new Burfi(), new Kajukatli(),
 new Jalebee(), new Rasgulla() };
```

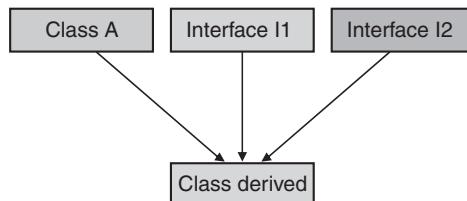
In the for loop when mawastatus function is called using this reference array ptr, respective mawastatus functions of each class are called.

### 9.3.4 Extending Classes and Interfaces

When extending a class and an interface, the class must be extended first and then the keyword implements has to be used. This means that the keyword extends must come before the keyword implements. The general syntax is as follows:

```
class new_class extends old_class implements inter1, inter2,
.....interN;
{
}
```

Use of class and inheritance together support the idea of multiple inheritances in Java. More than one interface can be implemented, but only one class can be extended. Diagrammatically, this can be shown as one, where a derived class has one base class and implements two interfaces.



An interface can extend another interface as shown in the following page.

```

interface inter1
{
 void fun1();
 void fun2();
}
interface inter2 extends inter1
{
 void fun3();
 void fun4();
 float limit = 23.99;
}

```

The interface `inter2` inherits `inter1`. Any class which implements `inter2` must provide implementation of all the four methods `funx`, where the value of `x` is from 1 to 4. The interface `inter2` also defines one final static constant `limit`, which can be used in the implementing classes.

Interfaces can be used for declaring constants, i.e., an interface can be declared to work exactly like `enum` used in C/C++. For example:

```

interface days
{
 int SUN = 1, MON = 2, TUE = 3, WED = 4, THU = 5, FRI = 6, SAT = 7;
}

```

Or

```

interface Months
{
 int JANUARY=1, FEBRUARY=2, MARCH=3, APRIL=4, MAY=5,
 JUNE=6, JULY=7, AUGUST=8, SEPTEMBER=9, OCTOBER =10,
 NOVEMBER = 11, DECEMBER =12;
}

```

The constants can be used in Java programs by implementing the interface and using in the following way:

```

class demo implements days,months
{
 switch(x)
 {
 case MON:
 ;
 break;
 case SUN:
 ;
 break;
 case TUE:
 ;
 break;

 }
}

```

```

 if(month==JAN)

 else if(month == DEC)

 }
}

```

/\*PROG 9.5 EXTENDING CLASS, IMPLEMENTING INTERFACE \*/

```

interface i_demo
{
 void show();
}
class first
{
 void display()
 {
 System.out.println("\n Show of first");
 }
}
class second extends first implements i_demo
{
 public void show()
 {
 System.out.println("\n Show of second");
 }
}
class JPS9
{
 public static void main(String[] args)
 {
 second ref = new second();
 ref.show();
 ref.display();
 }
}

```

**OUTPUT:**

```

Show of second
Show of first

```

*Explanation:* In the program, the class `second` extends the class `first` and simultaneously implements interface `i_demo`. In the `main`, an object reference of the class `second` is created and the function `show` and `display` are called.

## 9.4 PONDERABLE POINTS

1. A package is a collection of classes and interfaces.
2. A package can be imported to an application using the `import` statement.
3. Importing a package does not add to the size of the program.

4. Importing a package explicitly means that all classes and interfaces are not imported rather the desired class/interface is imported by explicitly mentioning the name of class/interface.
5. For creating a user-defined package `package` statement must be the first statement in the program file.
6. Each import statement must be placed on a separate line.
7. The `CLASSPATH` environment variable sets the path of classes used in the program.
8. Before using a user-defined package `CLASSPATH` must have been set.
9. An interface is a class where all the methods are declared but not defined and all fields are defined.
10. All the methods of an interface are by default abstract and the fields are static and final.
11. In order to make use of an interface, the keyword `implements` is used.
12. Interface allows simulating inheritance in Java.
13. While inheriting a class and interface to a class, the keyword `extends` must come first before the keyword `implements`.
14. A class can implement many interfaces but can extend only one class.
15. When implementing a method of an interface in a derived class, it must be preceded by the keyword `public`.
16. A reference of an interface can only call its methods and not the class in which it is implemented.

## REVIEW QUESTIONS

1. What is a package?
2. What is the difference between implicit and explicit import statement? Which one takes less time for compilation? Justify.
3. Write a code segment to use `java.awt.Graphics`
  - (a) Using import statement.
  - (b) Without using import statement.
4. How is a package designed? Explain the procedure with the help of suitable example.
5. What is the use of `javac -d` option?
6. Write a package called Clear, it contains one public method `clrscr()` to clear the screen, import the package and use it in another programs. Add another public method `starline()`. It prints the line of 15 stars.
7. What is the difference between public and default class.
8. Discuss the various types of access protection in package, with example.
9. Define interface. Give an example.
10. Compare and contrast the following:
  - (a) Interface and Class
  - (b) Interface and Abstract Class.
11. How to design and implement an interface?
12. How is one interface extended by the other?
13. Write a program in Java. A class Teacher contains two fields, Name and Qualification. Extend the class to Department, it contains Dept. No and Dept. Name. An interface named as College contains one field Name of the college. Using the above classes and Interface get the appropriate information and display it.
14. Justify the following statements: "import statement works much like the C/C++ #include statement".
15. Explain class path. What is the procedure to set user defined class path?

## Multiple Choice Questions

1. Which package provides classes necessary for the creation of applets and related operations?
  - (a) `java.awt.event`
  - (c) `java.lang`
  - (b) `java.applet`
  - (d) None of the above
2. The dynamically changing part of program memory can be divided into
  - (a) stack memory area
  - (b) heap memory area
  - (c) both (a) and (b)
  - (d) none of the above
3. A stack memory area is used to store the
  - (a) local variables declared in methods
  - (b) global variables
  - (c) static variable
  - (d) none of the above
4. A heap memory area is used to store the
  - (a) local variables declared in methods
  - (b) global variables
  - (c) memory for objects
  - (d) none of the above

5. An interface is a collection of
  - (a) constants
  - (b) variables
  - (c) classes
  - (d) constants and abstract methods
6. A package is a collection of
  - (a) related classes
  - (b) related interfaces
  - (c) related classes and interfaces
  - (d) none of the above
7. Java package is a grouping mechanism with the purpose of
  - (a) providing the library for Java program
  - (b) controlling the visibility of classes, interfaces and methods
  - (c) replacing header file used in C/C++
  - (d) none of the above
8. The import statement given below:  
1. `import java.io.*;`
2. `import java.io. DataInputStream;`
  - (a) 1- Explicit Import Statement, 2- Implicit Import Statement
  - (b) 1- Implicit Import Statement, 2- Explicit Import Statement
  - (c) 1-Implicit Import Statement, 2- Implicit Import Statement
  - (d) 1-Explicit Import Statement, 2- Explicit Import Statement
9. A class can implement many interfaces but can extend
  - (a) only one class
  - (b) two classes
  - (c) three classes
  - (d) no limit
10. Which of the following is used to make use of an interface?
  - (a) Import statement
  - (b) Extend
  - (c) Implements keyword
  - (d) None of the above

### KEY FOR MULTIPLE CHOICE QUESTIONS

---

1. b      2. c      3. a      4. c      5. d      6. c      7. b      8. b      9. a      10. c

# String and String Buffer

10

## 10.1 THE STRING CLASS

The String class represents character strings. All string literals in Java programs, such as hello,1234 and xyz123 are implemented as instances of this class. Strings are constants and their values cannot be changed after they are created. String buffers support mutable strings, i.e., which can be modified. They will be discussed in the next section. Since String objects are immutable, they can be shared.

The declaration of String class from the Java documentation is given below.

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

This states that String is a final class and object is its super class. The class implements three interfaces.

As an elementary example of String class, consider the various ways of declaring and initializing String variables which are given below.

```
String str="hello";
```

is equivalent to

```
char data[]={'h','e','l','l','o'};
String str = new String(data);
```

or,

```
String str=new String("hello");
```

Some more examples of how strings can be used are given below.

```
System.out.println("hello");
String s="fun";
System.out.println("hello"+s);
String temp="hello".substring(2,3);
String var=temp.substring(1,2);
```

The + operator can be used for concatenation of strings. This is the only use of + operator in which Java supports operator overloading. For example:

```
String s="hello";
String str=s +"world";
System.out.println(str); //will print "hello world"
String str1="This " + " is "+" demo";
```

is equivalent to

```
String str1 =" This is demo";
```

Similar to other primitive data types, a String variable must be initialized, i.e., the following program will give compilation error stating that 'variable str might not have initialized'

```
class JPS
{
 public static void main(String[] args)
 {
 String str;
 System.out.println(str);
 }
}
```

In order to rectify it, initialize the String variable str in the following way:

```
String str=null; or String str = new String();
```

**Note:** null is a special keyword which means that str does not contain reference of any String object. Printing a String object containing null will print null, but in the second case, using new operator it will print nothing.

## 10.2 CONSTRUCTORS FOR STRING CLASS

The String class provides a number of constructors that can be used in a variety of programming situations where strings are needed; some of the most common forms are discussed below.

### 1. **String()**

This form of constructor initializes a newly created String object so that it represents an empty character sequence. It can be used in the following way:

```
String str=new String();
System.out.println(str); //prints blank
```

### 2. **String (byte[]bytes)**

This form of constructor constructs a new String by decoding the specified array of bytes using the default charset of the platform. It can be used in the following way:

```
byte b[] = {'h','e','l','l','o'};
String s = new String(b);
System.out.println(s);
```

### 3. **String(byte[]bytes, int offset, int length)**

This form of constructor constructs a new String by decoding the specified subarray of bytes using the platform's default charset. Starting from offset, length number of characters is taken from byte array b. It can be used in the following way:

```
byte b[]={ 'C','o','o','l','e','r' };
String s = new String(b,1,4);
System.out.println(s); //prints oole
```

### 4. **String (char[]value)**

This form of constructor allocates a new String so that it represents the sequence of characters currently contained in the character array argument. It can be used in the following way:

```
char ch[] = {'T','w','o','p','i','e'};
String s = new String(ch);
System.out.println(s); //prints Twopie
```

**5. `String(char[] value, int offset, int length)`**

This form of constructor allocates a new String so that it represents the sequence of characters currently contained in the character array argument. It can be used in the following way:

```
char ch[]={'T','w','o','p','i','e'};
String s = new String(ch,3,3);
System.out.println(s); //prints pie
```

**6. `String(String original)`**

This form of constructor initializes a newly created String object so that it represents the same sequence of characters as the argument. In other words, the newly created string is a copy of the argument String. It can be used in the following way:

```
String str="Cutie Pie";
String s = new String(str);
System.out.println(s); // prints Cutie Pie
System.out.println(str); // prints Cutie Pie
```

## 10.3 METHODS OF STRING CLASS

The class String includes methods for examining individual characters of the sequence, such as for comparing strings, for searching strings, for extracting substrings and for creating a copy of a string with all characters translated to uppercase or to lowercase. Some of the commonly used methods are explained with small code snippets in Java.

**1. The `length()` method**

Signature: `int length()`

The `length()` gives the length of string in terms of number of characters. For example:

```
String str = "length";
int len =str.length();
```

will store 6 in variable len. It can be used in the following manner also.

```
len = "hello".length(); // len will have 5 as value
```

**2. The `charAt` method**

Signature: `char charAt(int index);`

This method returns the character at the specified index. The first character is at index 0. For example:

```
char ch="fun".charAt(1); //will store second character 'u' in ch
```

If the index is out of bound, i.e., not between 0 and `string_length-1` then `StringIndexOutOfBoundsException` Exception will be thrown.

**3. The `compareTo` method**

Signature: `int compareTo(String s);`

This method compares two strings lexicographically, i.e., according to dictionary and case of the letters on the basis of unicode character set. The uppercase letters have lower value than their corresponding lowercase counterparts. If the string being compared is greater than String s, a positive value is returned, negative value if String being compared is smaller than String s or 0 otherwise. The case of characters is also considered while comparing the two strings. The characters have been their ASCII value while comparing strings. For example:

```
String str1 = "Java", str2 ="java";
int c =str1.compareTo(str2);
```

```
if(c>0)
System.out.println(str1+" is greater");
else if(c<0)
System.out.println(str2+" is greater");
else
System.out.println("Both are equal");
```

As 'j' is greater than 'J', first if condition is true and 'java is greater' will be printed.

#### 4. The **compareToIgnoreCase** method

Signature: int compareToIgnoreCase(String s);

This method is similar to `compareTo` method with the difference that uppercase and lowercase letters are considered the same. Following the above example, 'java', 'Java' and 'JAVA' should all be treated the same.

#### 5. The **concat** method

Signature: String concat (String s);

This method concatenates the specified string s to the end of this string. If the length of the argument string is 0 the String object is returned. Otherwise, a new String object is created, representing a character sequence that is the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string. For example:

```
"devote".concat("es") returns "devotes"
"to".concat("get").concat("her") returns "together"
```

#### 6. The **equals** method

Signature: boolean equals(Object obj)

This method compares this string to the specified object. The result is true only if the argument is not null and is a String object that represents the same sequence of characters as this object. It is similar to `compareTo` method and the only difference is that it returns only true or false value of type Boolean. For example:

```
"fun".equals("fun") returns true;
"boom".equals("Boom") returns false;
```

#### 7. The **equalsIgnoreCase** method

Signature: boolean equalsIgnoreCase(Object obj)

This method compares string to the object without considering the case. For example:

```
"fun".equalsIgnoreCase("Fun") returns true;
"boom".equalsIgnoreCase("Boomer") returns false;
```

#### 8. The **endsWith** method

Signature: boolean endsWith (String suffix)

This method tests if this string ends with the specified suffix.

```
String str="Education";
String str1="Explanation";
boolean b1=str.endsWith("tion");
boolean b2=str1.endsWith("nation");
b1 and b2 gets true as their value.
```

#### 9. The **getChars** method

Signature: void getChars(int srcBegin, int srcEnd, char[] dst,
int dstBegin)

This method copies characters from this string into the destination character array. The first character to be copied is at index `srcBegin` and the last character at index `srcEnd-1` (thus, the total

number of characters to be copied is `srcEnd-srcBegin`). The characters are copied into the sub-array of `dst` starting at index `dstBegin` and ending at index:

`dstbegin + (srcEnd-srcBegin)-1`

For example:

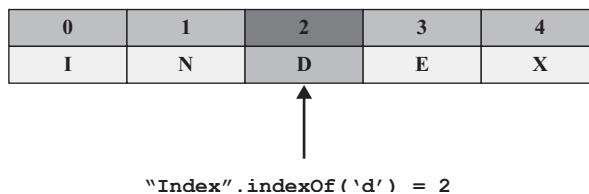
```
String str="Education";
char[]arr=new char[str.length()];
str.getChars(3,str.length(),arr,0);
System.out.println(arr); //displays "cation"
```

#### 10. The `indexOf` method

Signature: `int indexOf(int ch)`

This method returns the index within this string of the first occurrence of the specified character. If the character is not within the string -1 is returned. For example:

```
int x = "Index".indexOf('d'); //return 2
```



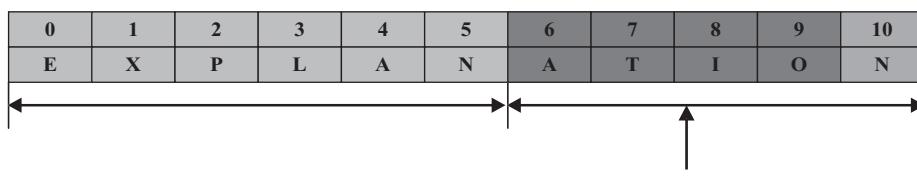
**Figure 10.1** Diagrammatical implementation of `indexOf` method

The second overloaded form of `indexOf` method is as follows:

```
int indexOf(int ch, int fromindex);
```

where `fromindex` is the starting index from which search is to start. For example:

```
int x = "Explanation".indexOf('n', 6);
```



**Figure 10.2** Implementation of overloaded `indexOf` method

Returns **10** as starting from **6th index** (7th character) 'n' is at 10th position as shown in Figure 10.2.

#### 11. The `lastIndexOf` method

Signature: `int lastIndexOf(int ch);`

This method returns the index within this string of the last occurrence of the specified character -1 if the character does not occur in the string. For example:

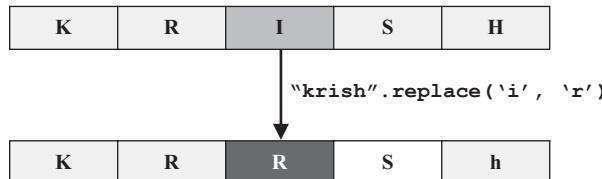
```
"abcdacd".lastIndexOf('a'); //returns 4
```

#### 12. The `replace` method

Signature: `String replace (char oldChar, char newChar)`

This method returns a new string from replacing all occurrences of `oldChar` in this string with `newChar`.

```
String s ="Krish".replace('i','r'); //store "Krrsh" in s.
```

**Figure 10.3** Implementation of `replace` method

The other form of replace is to replace a substring with a new string. For example:

```
String str = " This is new string";
String s = str.replace("new","old");
System.out.println(s);
```

The string s will have 'This is old string'.

### 13. The `startsWith` method

Signature: `boolean startsWith(String prefix);`

This method tests if this string starts with the specified prefix. For example:

```
"presume".startsWith("pre"); //returns true;
```

### 14. The `substring` method

Signature: `String substring(int beginIndex)`

This method returns a new string that is a substring of this string. For example:

```
String str = "This is new string";
String s = str.substring(5);
System.out.println(s); //s will print "is new string"
```

The other overloaded form is as follows:

```
String substring(int beginIndex, int endIndex)
```

This method returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex-1`. Thus the length of the substring is `endIndex-beginIndex`. For example:

```
String str="This is new string";
String s = str.substring(8,11);
System.out.println(s); //s will print "new"
```

### 15. The `toCharArray()` method

Signature: `char[] toCharArray()`

This method converts this string to a new character array. It returns a newly allocated character array whose length is the length of this string and whose contents are initialized to contain the character sequence represented by this string. For example:

```
String str = "Gorgeous";
char[]arr=str.toCharArray();
System.out.println(arr); //arr prints "Gorgeous"
```

### 16. The `toLowerCase` method

Signature: `String toLowerCase()`

This method converts all of the characters in this String to lower case. For example:

```
String str = "HEALTHY";
System.out.println(str.toLowerCase()); //will print "healthy"
```

## 17. The **toUpperCase** method

Signature: `String toUpperCase()`

This method converts all of the characters in this String to uppercase. For example:

```
String str = "heALTHY";
System.out.println(str.toUpperCase()); //willprint"HEALTHY"
```

## 18. The **toString** method

Signature: `String toString()`

This object (which is already a string) is itself returned. This method is used internally when something is printed which is not a string using `println` method. The data types or objects are first converted to String objects using the `toString` method and then displayed. One such usage has been observed in displaying the objects of class.

## 19. The **trim** method

Signature: `String trim()`

The method returns a copy of the string with leading and trailing white space omitted. For example:

```
String str = " \tFree\t ";
System.out.println(str.trim());
```

## 20. The **getBytes** method

Signature: `byte[] getBytes()`

This method encodes this String into a sequence of bytes using the platform's default charset, sorting the result into a new bytes array. For example:

```
String str = "Cutie Pie";
byte b[]=str.getBytes();
for(int i=0;i<b.length;i++)
System.out.println((char)b[i]); //prints Cutie Pie
```

Without typecasting by `char` output will be Unicode codes of the characters.

## 21. The **valueOf** method

There are five overloaded `valueOf` methods which convert primitive values into String objects. All methods are static and return a String object. The signature is as follows:

```
public static String valueOf(type b);
```

The type may be int, double, char, float and boolean. For example:

```
String dvalue = String.valueOf(123.456);
String fvalue = String.valueOf(654.854f);
String ivalue = String.valueOf(123);
String bvalue = String.valueOf(true);
String cvalue = String.valueOf('p');

System.out.println(dvalue); //prints 123.456
System.out.println(fvalue); //prints 654.854
System.out.println(ivalue); //prints 123
System.out.println(bvalue); //prints true
System.out.println(cvalue); //prints p
```

## 10.4 PROGRAMMING EXAMPLES (PART 1)

```
/*PROG 10.1 REVERSING A STRING*/
import MYIO.*;
class JPS1
{
 public static void main(String args[])
 {
 String str=null;
 MYINOUT.show("\nEnter the string:=");
 str=MYINOUT.str_val();
 char[]arr=new char[str.length()];
 for(int i=str.length()-1,j=0;i>=0;i--,j++)
 arr[j]=str.charAt(i);
 String revstr = new String(arr);
 MYINOUT.show("\nReverse string is:="+
 "+revstr+"\n");
 }
}
OUTPUT:
Enter the string:=NMIMS UNIVERSITY
Reverse string is:= YTISREVINU SMIMN
```

*Explanation:* As the own methods for handling the input and output package are being used, MYIO is imported in the program created in the chapter entitled Packages and Interfaces. The logic to reverse the string is simple. First, a new char array is created using the length() method of string as the size of the array. Now, using for loop and charAt method of string one character is extracted at a time from the string from last and put in the beginning of the array. For that variables i and j have been used. In each iteration of for loop i is decremented and j is incremented. In the end, the array arr is converted to a String and stored in revstr which contains reverse string. This revstr is then displayed.

```
/*PROG 10.2 CHECKING STRING FOR PALINDROME*/
import MYIO.*;
class JPS2
{
 public static void main(String args[])
 {
 String str = null;
 MYINOUT.show("\nEnter the string:=");
 str = MYINOUT.str_val();
 char[] arr = new char[str.length()];
 for (int i = str.length()-1,j=0;i>=0;i--,j++)
 arr[j] = str.charAt(i);
 String revstr = new String(arr);
 MYINOUT.show("\n Reverse string is:=" +revstr+"\n");
```

```

 boolean flag = str.equals(revstr);
 if (flag == true)
 MYINOUT.show("\n String is palindrome\n");
 else
 MYINOUT.show("\n String is not
palindrome\n");
 }
}

OUTPUT:

(First Run)
Enter the string:=PEEP
Reverse string is:= PEEP
String is palindrome

(Second Run)
Enter the string:=MPSTM
Reverse string is:= EMTSPM
String is not palindrome

```

*Explanation:* After reversing the string, the two strings are compared using `equals` method. If `equals` returns `true` it means the string is palindrome else the string is not palindrome.

```

/* PROG 10.3 ARRAY OF STRINGS */

import MYIO.*;
class JPS3
{
 public static void main(String args[])
 {
 String[] str ={ null };
 int size;
 MYINOUT.showln("\nEnter how many strings");
 size = MYINOUT.int_val();
 str = new String[size];
 int i;
 for (i = 0; i < size; i++)
 {
 MYINOUT.showln("\nEnter string number "+(i+1));
 str[i] = new String();
 str[i] = MYINOUT.str_val();
 }
 MYINOUT.showln("\nStrings entered are");
 for (i = 0; i < size; i++)
 MYINOUT.showln(str[i]);
 }
}

```

## OUTPUT:

```

Enter how many strings
3

Enter string number 1
NMIMS UNIVERSITY

Enter string number 2
PHI PUBLICATION

Enter string number 3
HARI MOHAN PANDEY

Strings entered are
NMIMS UNIVERSITY

PHI PUBLICATION
HARI MOHAN PANDEY

```

*Explanation:* String is a class in Java and creating an array of String means creating an array of object. Initially, all the String elements are initialized to null. Each String object has to be initialized before it can be put to use. In the for loop before taking String from the user, the line str[i] = new String(); does the same. The next value for all the strings is taken and using one more for loop they are displayed.

```

/* PROG 10.4 SORTING OF STRINGS */

import MYIO.*;
class JPS4
{
 public static void main(String args[])
 {
 String[] str = { null };
 String temp = null;
 int size;
 MYINOUT.showln("Enter how many strings");
 size = MYINOUT.int_val();
 str = new String[size];
 int i, j;
 for (i = 0; i < size; i++)
 {
 MYINOUT.showln("Enter string number "+(i+1));
 str[i] = new String();
 str[i] = MYINOUT.str_val();
 }
 MYINOUT.showln("Sorted strings are");
 for (i = 0; i < size; i++)
 for (j = i + 1; j < size; j++)
 if (str[i].compareTo(str[j]) > 0)
 {
 temp= str[i];

```

```

 str[i] = str[j];
 str[j] = temp;
 }
 for (i = 0; i < size; i++)
 MYINOUT.showln(str[i]);
}
}

OUTPUT:

Enter how many strings
4
Enter string number 1
Hari
Enter string number 2
Ravi
Enter string number 3
Vijaya
Enter string number 4
Ranjana
Sorted strings are
Hari
Ranjana
Ravi
Vijaya

```

*Explanation:* For sorting the strings, the same logic is followed as was done for sorting the array of integers, float, etc. Two for loops are used and using compareTo method, the first two strings are compared, and if first string is greater than the second one (lexicographically) they are swapped. This continues and in the end the sorted strings result.

```
/*PROG 10.5 DEMO OF TOSTRING METHOD */

import java.io.*;
class Fruit
{
 String name;
 String color;
 boolean contain_seed;
 Fruit(String n, String c, boolean cs)
 {
 name = n;
 color = c;
 contain_seed = cs;
 }
 public String toString()
 {
 String str = "Fruit Name = " + name;
 str = str + "\nFruit Color = " + color;
 str = str + "\nFruit contains seed = " +
 contain_seed + "\n";
 }
}
```

```

 return str;
 }
}
class JPS5
{
 public static void main(String[] args)
 {
 Fruit F1 = new Fruit("Banana", "Yellow", false);
 Fruit F2 = new Fruit("Apple", "Red", true);
 System.out.println("\tFruit 1");
 System.out.println(F1);
 System.out.println("\tFruit 2");
 System.out.println(F2);
 }
}

OUTPUT:

 Fruit 1
Fruit Name = Banana
Fruit Color = Yellow
Fruit contains seed = false

 Fruit 2
Fruit Name = Apple
Fruit Color = Red
Fruit contains seed = true

```

*Explanation:* The main lines shown in bold have arguments as object of class `Fruit`. Objects cannot be displayed in this manner. But the method `toString` helps one in these types of situations. In order to display objects in the above manner, all one needs to do is to define `toString` method as shown in the program. The method is called automatically to convert the object `F1` and `F2` into `Strings`. In the method `toString` the strings are concatenated which are to be displayed and in the end the string is returned. If `toString` is not overridden the compiler will flash error. This method can be used to print any object of an user-defined class.

## 10.5 THE STRINGBUFFER CLASS

A `StringBuffer` is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls. The declaration of the `StringBuffer` class as given in the Java documentation is as follows:

```

public final class StringBuffer extends Object
 implements Serializable, CharSequence

```

This states that `StringBuffer` is a final class extending the `Object` class. The class implements two interfaces.

The principal operations on a `StringBuffer` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The `append` method always adds these characters at the end of the buffer and the `insert` method adds the characters at a specified point.

For example, if `sbuf` refers to a string buffer object whose contents are ‘jul’, then the method class `sbuf.append('le')` would cause the string buffer to contain ‘julle’, whereas `sbuf.insert(1, 'le')` would alter the string buffer to contain ‘jleul’.

In general, if `sb` refers to an instance of a `StringBuffer`, `sb.append(x)` has the same effect as `sb.insert(sb.length(), x)`.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

## 10.6 CONSTRUCTOR OF STRINGBUFFER CLASS

The `StringBuffer` class defines three main constructors for using `StringBuffer` objects in the programs.

### 1. `StringBuffer()`

This form of constructor constructs a string buffer with no characters in it and an initial capacity of 16 characters. It can be used as follows:

```
StringBuffer sb = new StringBuffer();
```

### 2. `StringBuffer(int capacity)`

This form of constructor constructs a string buffer with no characters in it and the specified initial capacity. The initial capacity becomes the size of the buffer. The above form can be used as follows:

```
StringBuffer sb = new StringBuffer(5);
```

### 3. `StringBuffer(String str)`

This form of constructor constructs a string buffer initialized to the contents of the specified string. The initial capacity of the string buffer is 16 plus the length of the string argument. It can be used as follows:

```
StringBuffer sb = new StringBuffer("demo");
```

Here, the capacity of object `sb` is  $4 + 16$ , i.e., 20. Java keeps room for additional 16 characters to avoid reallocation problems. By providing extra capacity for 16 characters, the number of reallocation is reduced as it is time-consuming.

## 10.7 METHODS OF STRINGBUFFER CLASS

There are a number of methods which are common to `String` and `StringBuffer` class. Only those methods are discussed which are unique to `StringBuffer` class. However, listing of frequently used methods is given in Table 10.1.

| S. No. | Name of Method |
|--------|----------------|
| 1      | append         |
| 2      | capacity       |
| 3      | charAt         |
| 4      | delete         |
| 5      | deleteCharAt   |
| 6      | ensureCapacity |
| 7      | getChars       |
| 8      | indexOf        |
| 9      | insert         |

| S. No. | Name of Method |
|--------|----------------|
| 10     | lastIndexOf    |
| 11     | length         |
| 12     | replace        |
| 13     | reverse        |
| 14     | setCharAt      |
| 15     | setLength      |
| 16     | substring      |
| 17     | toString       |
| 18     | trimToSize     |

Table 10.1 Methods of `StringBuffer` class

### 1. The **capacity** method

Signature: int **capacity()**

This method returns the current capacity of the `StringBuffer` object. The initial capacity is of 16 characters when no characters are present in the buffer. As characters are entered into the buffer, the capacity is usually added to the length of `StringBuffer` object.

**Example:**

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity()); //prints 16
```

### 2. The **delete** method

Signature: `StringBuffer delete (int start, int end)`

This method removes the characters in a substring of this sequence. This substring begins at the specified `start` and extends to the character at index `end-1` or to the end of the sequence if no such character exists. If `start` is equal to `end`, no changes are made.

**Example:**

```
StringBuffer sb = new StringBuffer("Demolition");
sb.delete(4,6); // deletes characters at position 4 and 5.
System.out.println(sb); //prints "Demotion".
```

### 3. The **deleteCharAt** method

Signature: `StringBuffer deleteCharAt (int index)`

This method removes the `char` at the specified position in this sequence.

**Example:**

```
StringBuffer sb = new StringBuffer("Funn");
sb.deleteCharAt(2);
System.out.println(sb); //prints "Fun"
```

### 4. The **reverse** method

Signature: `StringBuffer reverse()`

This method causes this character sequence to be replaced by the reverse of the sequence.

**Example:**

```
StringBuffer sb = new StringBuffer("Fantastic");
System.out.println(sb.reverse()); //prints citsatnaF
```

### 5. The **setCharAt** method

Signature: `void setCharAt (int index, char ch)`

The character at the specified index is set to `ch`.

**Example:**

```
sb.setCharAt(2,'s');
sb.setCharAt(3,'s');
System.out.println(sb); //prints "Fuss"
```

### 6. The **ensureCapacity** method

Signature: `void ensureCapacity (int capacity)`

This method ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, a new internal array is allocated with greater capacity. The method can be used where it is known in advance that a large number of strings will be appended to the `StringBuffer` object.

**Example:**

```
StringBuffer sb = new StringBuffer(5);
System.out.println(sb.capacity());
sb.ensureCapacity(20);
System.out.println(sb.capacity());
```

**7. The setLength method**

Signature: void setLength(int newLength);

This method sets the length of the `StringBuffer` object. The sequence is changed to a new character sequence whose length is specified by the argument. If the `newLength` argument is greater than or equal to the current length, sufficient null characters are appended so that length becomes the `newLength` argument. In case, `newLength` is less than the current length characters are lost from the character sequence.

```
StringBuffer sb = new StringBufer("Demo of String Buffer");
System.out.println("Before setting length");
System.out.println("sb="+sb+", Length = "+sb.length());
sb.setLength(15);
System.out.println("After setting length");
System.out.println("sb="+sb+", Length=" +sb.length());
```

Output of the above code is as follows:

```
Before setting length
sb= Demo of String Buffer, Length = 21
After setting length
sb = Demo of String, Length = 15
```

**8. The append method**

The `append` method is used for appending any primitive type of data at the end of invoking `StringBuffer` object. Any primitive data type means that the method is overloaded for different types of elementary data types and also for the `Object` class. Its general form is as follows:

Signature: `StringBuffer append (type t);`

where `t` may be `char`, `int`, `long`, `double`, `float`, `Object`, etc. The method returns a `StringBuffer` object after the `append` operation is over. This makes it possible to chain together more than one `append` methods.

**Example 1:**

```
StringBuffer sb = new StringBuffer("Rosy");
sb.append("Lips");
sb.append(true);
sb.append(123);
System.out.println(sb); //print Rosy Lipstrue123
```

**Example 2:**

```
StringBuffer sb = new StringBuffer("Chubby");
sb.append("Cheeks").append("Rosy").append("Lips");
System.out.println(sb);
output is : Chubby Checks Rosy Lips
```

**9. The insert method**

The `insert` method has a number of overloaded forms. Some of them are discussed below.

```
public StringBuffer insert(int ioffset, type d);
```

The method inserts d into this `StringBuffer` instance at offset `ioffset`. The type may be `int`, `char`, `double`, `boolean`, `float`, `long`, `char[]`, `Object` and `String`.

**Example:**

```
StringBuffer sb = new StringBuffer(20);
sb.insert(0,"Hello");
sb.insert(2,true);
sb.insert(3,23.45);
System.out.println(sb);
```

The output of the code is Het23.45ruello

## 10.8 PROGRAMMING EXAMPLES (PART 2)

```
/*PROG 10.6 STRINGBUFFER DEMO VER 1*/
import MYIO.*;
class JPS6
{
 public static void main(String args[])
 {
 StringBuffer sb = new StringBuffer("Buffer");
 MYINOUT.showln("Capacity = " + sb.capacity());
 MYINOUT.showln("Buffer contents =" + sb);
 MYINOUT.showln("Buffer Length = " + sb.length());
 }
}
OUTPUT:
Capacity = 22
Buffer contents = Buffer
Buffer Length = 6
```

*Explanation:* The capacity shown as output is the sum of the initial capacity, i.e., 16 characters plus length of the contents 'Buffer' The rest is simple to understand.

```
/*PROG 10.7 STRINGBUFFER DEMO VER 2 */
import MYIO.*;
class JPS7
{
 public static void main(String args[])
 {
 StringBuffer sb = new StringBuffer("Buffer");
 MYINOUT.showln("Capacity = " + sb.capacity());
 MYINOUT.showln("Buffer contents = " + sb);
 MYINOUT.showln("Buffer Length = " + sb.length());
```

```

 sb.append("Temporary");
 sb.insert(6, " is ");
 sb.append(" storage");
 MYINOUT.showln("Capacity = " + sb.capacity());
 MYINOUT.showln("Buffer contents = " + sb);
 MYINOUT.showln("Buffer Length = " + sb.length());
 }
}

OUTPUT:

Capacity = 22
Buffer contents = Buffer
Buffer Length = 6
Capacity = 46
Buffer contents = Buffer is Temporary storage
Buffer Length = 27

```

*Explanation:* The program is simple to understand.

## 10.9 PONDERABLE POINTS

1. A `String` is a class in Java defined in the package `java.lang`.
2. All variables of `String` class are its object which must be initialized before they can be used.
3. All string variables are of fixed length which can be found out by using `length` method.
4. A `String` object can provide flexible strings whose length can increase and decrease.
5. The `StringBuffer` class provides flexible strings whose length can increase and decrease.
6. The default capacity of a `StringBuffer` object is 16 characters.

## REVIEW QUESTIONS

1. Distinguish between the following terms:
  - (a) `String` and `StringBuffer`
  - (b) `Length` and `length()`
  - (c) `equal` and `==`
  - (d) `length()` and `capacity()`
  - (e) `setCharAt()` and `insert()`
2. What is wrong in the following code:  
`Vector v = new Vector(10);  
v.addElement(21);`
6. What will be the output of the following program?

```

class JPS_test
{
 static String b;
 public static void main(String args[])
 {
 String a = new String(); //empty string
 System.out.println("a = " + a);
 }
}

```

3. Write a program that accepts a name list of five students from the command line and store in a vector.
4. Write a program to generate Fibonacci series where the data are saved in vector.
5. Write a method called `remove (String s1, int n)` that returns the input string with nth element removed.

```

 System.out.println("b = " + b);
 }
}

```

7. What will be the output of the following program?

```

class JPS_test
{
 public static void main(String args[])
 {
 String S1 = new String("Hari"); //empty string
 String S2;
 S2 = S1;
 if (S1 == S2)
 System.out.println("Hari O Hari");
 System.out.println("Sorry, I'm unhappy");
 }
}

```

8. Write and run the following Java program that does the following:

- (a) Declare a string object named s1 containing the string "Object Oriented Programming-Java 5".
- (b) Print the entire string.
- (c) Use the length() method to find the length of the string.
- (d) Use the charAt() method to find the first character in the string.
- (e) Use charAt() and length() methods to print the last character in the string.
- (f) Use the indexOf() and the substring() method to print the first word in the String.

9. Try to compile a program containing the lines

```

String s;
String s;

```

Does Java allow this? If not, what message do you get?

10. What message does the compiler give if we try to compile a program containing the statements:

```

s = "hello";
String s;

```

- 11. Try to compile a program containing the lines
- String s;
- PrintStream s;
- Does Java allow this? If not, what message do you get?
- 12. Suppose that a class defines a method with the prototype
- String first(String)
- Give two other methods that can be defined in the class.
- Give two other methods that cannot be defined in the class.
- 13. Show the order of evaluation for the expression:
- " h e l l o ".t o U p p e r C a s e ( ) .concat("there");
- 14. What error message does Java give if a program contains the following line?
- System.out.println("hello".toUpper-
- case);

## Multiple Choice Questions

1. String as a class in Java is defined in
  - (a) java.io package
  - (b) java.awt package
  - (c) java.lang package
  - (d) java.applet package
2. The default capacity of a StringBuffer object is
  - (a) 8 characters (c) 32 characters
  - (b) 16 characters (d) 64 characters
3. The method returns the character at the specified index
  - (a) char charAt(int index);
  - (b) char charAt[int index];
  - (c) int charAt(int index);
  - (d) int charAt[int index];
4. The method used for testing the if string ends with the specified suffix
  - (a) int endswith(String suffix)
  - (b) char endsWith(String suffix)
  - (c) boolean endsWith(String suffix)
  - (d) None of the above

5. The `getChars` method is used to copy characters from this string into the destination character array. The signature of the method `getChars` is:
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  - `int getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  - `boolean getChars (int srcBegin, int srcEnd, char[] dst, int dst Begin)`
  - None of the above
6. The `indexOf` method is used to return the index within this string of the first occurrence of the specified character. If character is not within the string, it returns
- 0
  - +1
  - 1
  - 2
7. In Java \_\_\_\_\_ method is used for testing if this string starting with the specified prefix
- `int startsWith(String prefix)`
  - `void startswith(String prefix)`
- (c) `boolean startsWith(String prefix)`  
 (d) `long startsWith(String prefix)`
8. If the method returns a copy of the string, with the leading and trailing white space omitted it is a
- `toString` method
  - `getBytes` method
  - `trim` method
  - `toCharArray` method
9. `StringBuffer sb = new StringBuffer(10);`  
 In the code snippet shown above 10 shows:
- index number for the string
  - capacity
  - reference number
  - none of the above
10. The `StringBuffer` class provides flexible strings whose length can
- increase only
  - decrease only
  - increase and decrease as per the need
  - neither increase nor decrease—fixed at the initial stage

## KEY FOR MULTIPLE CHOICE QUESTIONS

1. c      2. b      3. a      4. c      5. a      6. c      7. c      8. c      9. b      10. c

# 11

# Exception Handling

## 11.1 INTRODUCTION

In Java, errors can be divided into two categories: compile time errors and runtime errors. Compile time errors are syntactic errors during writing of the program. Most common examples of compile time errors are missing semicolon, comma, double quotes, etc. They occur mainly due to poor understanding of language or lack of concentration in writing the program.

Logical errors occur mainly due to improper understanding of the program logic by the programmer. Logical errors cause the unexpected or unwanted output.

Exceptions are runtime errors which a programmer usually does not expect. They occur accidentally, which may result in abnormal termination of the program. Java provides exception handling mechanism, which can be used to trap this exception and run programs smoothly after catching the exception.

Common examples of exceptions are division zero, opening file which does not exist, insufficient memory, violating array bounds, etc.

## 11.2 WHAT IS EXCEPTION?

Exception can be defined as follows: “An *exception* is an event that occurs during the execution of a program, which disrupts the normal flow of the program’s instructions”.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an **exception object**, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called **throwing an exception**.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible ‘somethings’ to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the **call stack** (Figure 11.1).

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler**. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

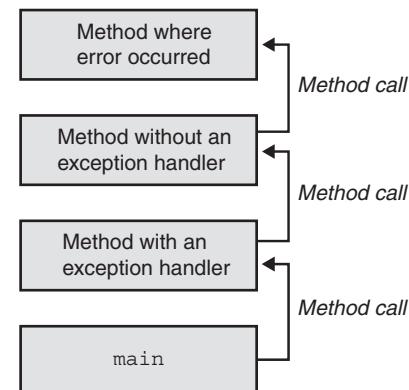
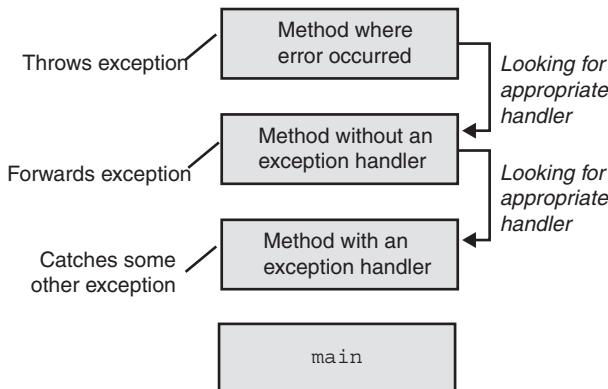


Figure 11.1 The call stack

The exception handler chosen is said to **catch the exception**. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in Figure 11.2, the runtime system (and, consequently, the program) terminates.



**Figure 11.2** Searching the call stack for the exception handler

### 11.3 BASIS FOR EXCEPTION HANDLING

A program is correct if it accomplishes the task it was designed to perform. It is robust if it can handle illegal inputs and other unexpected situations in a reasonable way. For example, consider a program that is designed to read some numbers from the user and then print the same numbers in sorted order. The program is correct if it works for any set of input numbers. It is robust if it can also deal with non-numeric input by, for example, printing an error message and ignoring the bad input. A non-robust program might crash or give nonsensical output in the same circumstance.

Getting a program to work under ideal circumstances is usually a lot easier than making the program robust. A robust program can survive unusual or ‘exceptional’ circumstances without crashing. One approach to writing robust programs is to anticipate the problems that might arise and to handle tests in the program for each possible problem. For example, a program will crash if it tries to use an array element `ARR[i]`, when ‘`i`’ is not within the declared range of indices for the array `ARR`. A robust program must anticipate the possibility of a bad index and guard against it. This could be done with an `if` statement.

```

if (i<0 || i>=ARR.length)
{
 //Do something to handle the out-of-range index, i
}
else
{
 //Process the array element, ARR[i]
}

```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It is not always clear what to do when an error is detected. Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of `if` statement.

Java provides a neater, more structured alternative method for dealing with errors that can occur while a program is running. The method is referred to as exception handling. The word ‘exception’ is meant to be more general than ‘error’. It includes any circumstance that arises as the program is executed, which is

meant to be treated as an exception to the normal flow of control of the program. Exception handling is the process to handle the exception if generated by the program at run time. The aim of exception handling is to write a code which passes exception generated by the program at run time. The aim of exception handling is to write a code which passes exception generated to a routine which can handle the exception and can take suitable action.

## 11.4 EXCEPTION-HANDLING MECHANISM

In Java, exceptions are objects which are thrown. All such codes which might throw an exception during the execution of the program must be placed in the `try` block. The usage of `try` block has been observed in a number of programs earlier. The exception thrown must be caught by the `catch` block. If not caught, the particular program may terminate abnormally. An exception can be thrown explicitly or implicitly. Exception can be explicitly thrown using the keyword `throw`. In explicit exception, a method must tell what type of exception it might `throw`. This can be done by using the `throw` keyword. All the other exceptions thrown by the Java runtime system are known as implicit exceptions. A `final` clause may be put after the `try-catch` block, which executes before the method returns. A `try` block must have a corresponding `catch` block, though it may have a number of `catch` blocks.

This `try` block is also known as **exception-generated block or guarded region**. The `catch` block is responsible for catching the exception thrown by the `try` block. It is also known as exception handler block. When the `try` block throws an exception, the control of the program passes to the `catch` block, and if argument matches, the exception is caught. In case no exception is thrown, the `catch` block is ignored and control passes to the next statement after the `catch` block.

The general syntax of exception-handling construct is shown as follows:

```

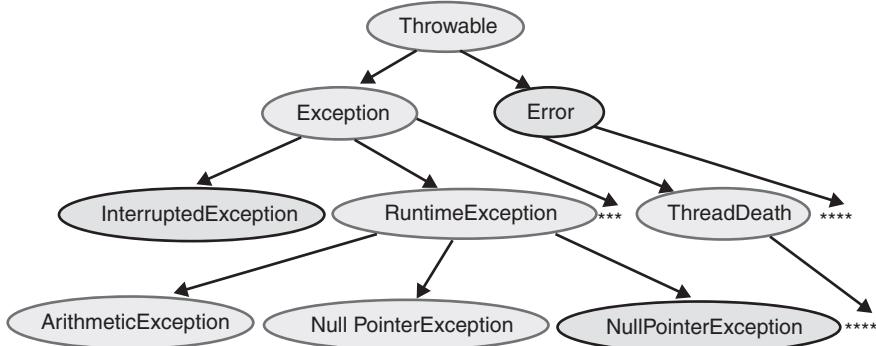
try
{
 statements;
 statements;
 statements;
}
catch(Exceptionclass1 object)
{
 statements for handling the exception;
}
catch(Exceptionclass1 object)
{
 statements for handling the exception;
}
.....
finally
{
 //statements executed before method returns
}

```

## 11.5 EXCEPTION CLASSES IN JAVA

In Java, exceptions are objects. When you throw an exception, you throw an object. You cannot throw just any object as an exception. The exception class hierarchy is shown in Figure 11.3. All exception class are defined in the package `java.lang`, which is default imported in all Java programs.

```
package java.lang;
```



**Figure 11.3** Exception class hierarchy in Java

The top most exception class is the `Throwable` class. `Throwable` serves as the base class for an entire family of classes, declared in `java.lang`, that a particular program can instantiate and throw. As can be seen in Figure 11.3, `Throwable` has two direct subclasses, `Exception` and `Error`. Exceptions (members of the `Exception` family) are thrown to signal abnormal conditions that can often be handled by some catch block although it is possible they may not be caught and, therefore, could result in a dead thread. `Exception` class is also used when generated exceptions are needed to be thrown. For that `Exception` class will be made as the base class. Errors (members of the `Error` family) are usually thrown for more serious problems, such as `OutOfMemoryError`, that may not be so easy to handle. In general, the code that is written should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API or by the Java virtual machine itself.

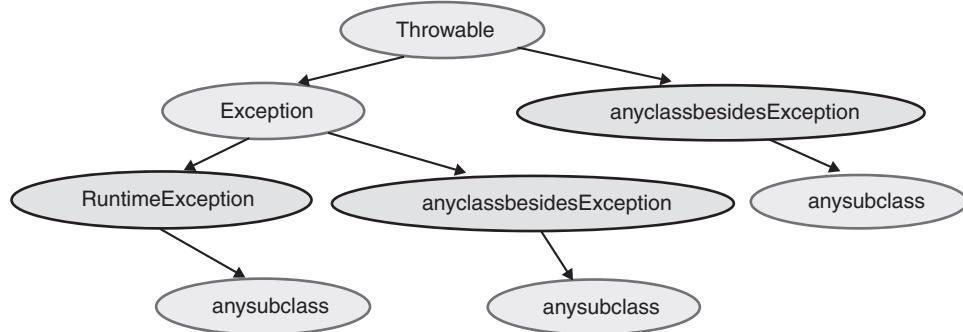
### 11.5.1 Runtime Exception

There is a whole group of exception types in this category. They are always thrown automatically by Java and one does not need to include them in exception specifications. Conveniently enough, they are all grouped together under a single base class called `RunTimeException`. It establishes a family of types that have some characteristics and behaviours in common. Also, one never needs to write an exception specification because a method might throw a `RunTimeException`. Because they indicate bugs, `RunTimeException` is not usually caught; it is dealt with automatically. Even though `RunTimeExceptions` are not caught typically, in one's own packages some of the `RunTimeExceptions` might be chosen to throw. Some of the examples or exceptions which come under this `RunTimeException` class are division by zero, array and string index out of bound, casting a class, illegal format exception, etc.

### 11.5.2 Checked versus Unchecked Exception

There are two kinds of exceptions in Java: checked and unchecked. Only checked exceptions need appear in throws clauses. The general rule is: any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause. Checked exceptions are so called because both the Java compiler and Java virtual machine check to make sure this rule is obeyed.

Whether or not an exception is ‘checked’, is determined by its position in the hierarchy of `Throwable` classes. Figure 11.4 shows that some parts of the `Throwable` family tree contain checked exceptions while other parts contain unchecked exceptions. To create a new checked exception, another checked exception is simply extended. All `Throwables` that are subclasses of `Exception`, but not subclasses of `RuntimeException` are checked exceptions.

**Figure 11.4** Checked and unchecked exception

All the checked and unchecked exceptions defined by Java are given in Tables 11.1 and 11.2..

| Exception Class                 | Reason of Occurrence                                             |
|---------------------------------|------------------------------------------------------------------|
| ArithmaticException             | Arithmatic error, such as divide by zero                         |
| ArrayIndexOutOfBoundsException  | Array index if out-of-bounds                                     |
| ArrayStoreException             | Assignment to an array element of an incompatible type           |
| ClassCastException              | Invalid cast                                                     |
| IllegalArgumentException        | Illegal argument used to invoke a method                         |
| IllegalMonitorStateException    | Illegal monitor operation, such as waiting on an unlocked thread |
| IllegalStateException           | Environment or application is in incorrect state                 |
| IllegalThreadStateException     | Requested operation not compatible with current thread state     |
| IndexOutOfBoundsException       | Some type of index is out-of-bound                               |
| NegativeArraySizeException      | Array created with a negative size                               |
| NullPointerException            | Invalid use of a null reference                                  |
| NumberFormatException           | Invalid conversion of a string to a numeric format               |
| SecurityException               | Attempt to violate security                                      |
| StringIndexOutOfBoundsException | Attempt to index outside the bounds of string                    |
| UnsupportedOperationException   | An unsupported operation was encountered                         |

**Table 11.1** Java's unchecked runtime exception subclasses

| Exception Class            | Reason of Occurrence                                                        |
|----------------------------|-----------------------------------------------------------------------------|
| ClassNotFoundException     | Class not found.                                                            |
| CloneNotSupportedException | Attempt to clone an object that does not implement the cloneable interface. |
| IllegalAccessException     | Access to a class is denied.                                                |
| InstantiationException     | Attempt to create an object of an abstract class or interface.              |
| InterruptedException       | One thread has been interrupted by another thread.                          |
| NoSuchFieldException       | One thread has been interrupted by another thread                           |
| NoSuchMethodException      | A requested method does not exist.                                          |

**Table 11.2** Java's checked exceptions defined in java.lang

## 11.6 WITHOUT TRY-CATCH

Before dealing with the exception and handling them, it will be seen in the absence of try-catch block, how exceptions are thrown by the runtime system and how they are handled.

Consider the following program given below:

```
/*PROG 11.1 WITHOUT TRY-CATCH BLOCK */

class JPS1
{
 public static void main(String[]args)
 {
 String str = "Exception";
 int x = str.charAt(str.length()+1);
 System.out.println("Control won't reach here");
 }
}

OUTPUT:

Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: String index out
of range: 10
 at java.lang.String.charAt(String.java:687)
 at JPS1.main(JPS1.java:6)
```

*Explanation:* In the program, an index is tried to be accessed from a String str which is out of bound; therefore, this causes an exception to be thrown.

When the Java runtime system sees the string index is out of bound, it constructs a new exception object and throws it. The exception thrown must be caught, but there is no catch block here for catching the

exception, so exception thrown results in terminating the program prematurely. In the absence of try-catch block, the default exception handler of the Java runtime system handles the exception, but at the cost of terminating the program abnormally.

Note the output of the program, the default handler returns a string which shows where the exception occurred in the program, what type of exception it was and the precise reason for the generation of exception. The last line of the output shows in which method or subroutine the exception was generated. The output is actually a stack trace, i.e., examining the system stack which was storing what went wrong. The stack trace shows all the methods in sequence that tells us the error in string form.

A stack trace provides information on the execution history of the current thread, and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that can be made use of when an exception has been thrown.

## 11.7 EXCEPTION HANDLING USING TRY AND CATCH

This section presents how programmatically exceptions are thrown within try block and handled by the catch block. To begin with the first program on exception handling is given below.

```
/*PROG 11.2 DEMO OF EXCEPTION HANDLING EXCEPTION IN
CONVERSION */

import java.lang.*;
class main
{
 public static void main(String args[])
 {
 try
 {
 int num;
 num=Integer.parseInt("ABC123");
 System.out.println("Control won't reach
 here");
 }
 catch(NumberFormatException E)
 {
 System.out.println("Exception thrown");
 System.out.println(E.getMessage());
 }
 System.out.println("Out of try-catch block");
 }
}

OUTPUT:
Exception thrown
For input string: "ABC123"
Out of try-catch block
```

*Explanation:* In the try block, a string is converted into integer. The string contains some alphabets along with digits. If the string were having just the digits, the conversion from string to integer was possible. But as it contains alphabets, conversion would not be allowed and an exception object of class type NumberFormatException will be thrown from try block. This object thrown will be caught

by the catch block in the object E, which is also of NumberFormatException class, so a match will be found. The getMessage method defined in the Exception can be used to display the internally generated message. The second line of output is internally generated message returned as String from getMessage().

```
/*PROG 11.3 DEMO OF EXCEPTION HANDLING ACCESSING ARRAY
INDEX OUT OF RANGE */

import java.lang.*;
class JPS3
{
 public static void main(String [] args)
 {
 try
 {
 int[]num = {0,1,2,3,4};
 int x = 25/num[5];
 System.out.println("Control won't reach
 here");
 }
 catch(ArrayIndexOutOfBoundsException E)
 {
 System.out.println("Exception thrown");
 E.printStackTrace();
 }
 }
}

OUTPUT:
Exception thrown
java.lang.ArrayIndexOutOfBoundsException: 5
at JPS3.main(JPS3.java:10)
```

*Explanation:* In the try block, an array element is accessed, the index of which is out of bound. So try block throws an exception object of type ArrayIndexOutOfBoundsException. This exception object thrown is caught by the catch block in the object E. Using this object E, the stack trace is displayed by a call to printStackTrace method. The output is due to printStackTrace method, which clearly indicates types and exception occurred, and place and the line number.

**1. Try with multiple catch:** Sometimes it is possible that a single try block may throw exceptions of different types at run time. If any of the exception is not handled that might be thrown, a premature termination of the program occurs. To catch all such types of exception, one catch block can be placed per exception that might be generated within the try block. Depending on what type of exception thrown, corresponding catch blocks catches the exception object thrown by the try block. The general syntax is as follows:

Here, if the exception object thrown is of `ExceptionClass1` type, the first catch block will catch the exception object thrown and the remaining catch blocks will be bypassed. In case, the exception object thrown does not match with the first catch block, the second catch block is tried and so on. At a time only one catch block will be used. It is also possible sometimes that `ExceptionClass1` is a base class of `ExceptionClass2` or/and `ExceptionClass3`, then any exception object thrown of

a type ExceptionClass1 and ExceptionClass2 and/or ExceptionClass3 will be caught by ExceptionClass1.

```
/*PROG 11.4 TRY WITH MULTIPLE CATCH BLOCK VER 1 */

import java.io.*;
class JPS4
{
 public static void main(String[]args)
 {
 PrintStream cout = System.out;
 try
 {
 DataInputStream input;
 input = new DataInputStream(System.in);
 int n1, n2, ans = 0;
 cout.println("Enter first number");
 n1 = Integer.parseInt(input.readLine());
 cout.println("Enter second number");
 n2 = Integer.parseInt(input.readLine());
 ans = n1/n2;
 cout.println("This will be printed");
 }
 catch(ArithmaticException E)
 {
 cout.println("Exception thrown:");
 cout.println("Description:"+E);
 }
 catch(NumberFormatException E)
 {
 cout.println("Exception thrown:");
 cout.println("Description:"+E);
 }
 catch(IOException E)
 {
 cout.println("Exception thrown:");
 cout.println("Description:"+E);
 }
 cout.println("Out of try-catch block");
 }
}

OUTPUT:
(First Run)
Enter first number
12
Enter second number
5
This will be printed
Out of try-catch block
```

```
(Second Run)
Enter first number
12
Enter second number
0
Exception thrown:
Description:java.lang.ArithmaticException: / by zero
Out of try-catch block

(Third run)
Enter first number
46
Enter second number
4.5
Exception thrown:
Description:java.lang.NumberFormatException: For input
string: "4.5"
Out of try-catch block
```

*Explanation:* In the program within the `try` block, three different types of exceptions may be thrown: First, exceptions of the type `ArithmaticException`, which may be generated when division by zero operation is performed. Second, when instead of integer some other data types are input, like `float`, `double`, `char` or `string` or even just press Enter; exception of type `NumberFormatException` type is thrown. Third, when there is some error in reading input from keyboard or writing output onto the screen. This will not be generated on the user part, but this exception must be caught otherwise it will throw compilation error.

In the first run of the program, the division of the two numbers are simply obtained as output; no exception is generated. In the second run, when 0 is input for denominator, Java runtime system checks that division by zero operation is performed, which is an illegal operation, so it constructs an object of `ArithmeticException` type and throws it. This thrown exception is caught by the first catch block, which displays the description of the exception thrown. It should be noted that this time the object `E` is simply placed in `cout.println` method. Java interpreter automatically converts the exception object into `String` by calling the `toString` method. In the third run of the program, `float` is intentionally input. As the `Integer.parseInt` cannot parse `float` string into integer, an exception object of `NumberFormatException` class is generated and thrown. The thrown object is caught by the second catch block.

Note in all run after handling the exception by the catch blocks, statements after the `try-catch` block execute as if nothing has happened.

Whenever there are multiple catch statements, it is possible that one exception class in a catch block is a subclass or super class of another exception class placed in another `catch` block. In this scenario, if the `Exception1` class in the first catch block is super class of the `Exception2` class placed in the next catch block, all the exceptions thrown for the `Exception2` class will be caught by the first catch block. In this case, second catch block will not be used in the program, thus generating compilation error. See the next program given below.

**Note:** The division by zero exception will not be thrown in case of data type `float` and `double`. You will get answer ‘`Infinity`’, and if you try to perform operation `0/0`, you will get the output `NaN`, which means `0/0` is not a Number.

**2. The throw keyword:** In all the programs of exception handling seen earlier, the Java runtime system was responsible for throwing the exceptions. All those exceptions come under the category of implicit exceptions. If we want, we can throw exceptions manually or explicitly. For that, Java provides the keyword `throw`. The `throw` keyword can be used to throw an exception explicitly. It can be used to rethrow an exception which has already been thrown.

The `throw` statement requires a single argument: a throwable object. Throwables objects are instances of any subclass of the `Throwable` class. Here is an example of a `throw` statement.

```
throw someThrowableObject;
```

The `throw` statement can be given in an example. The following `pop` method is taken from a class that implements a stack of integers. The method removes the top element from the stack and returns it.

```
public int pop()
{
 int obj;
 if (top == 0)
 {
 throw new EmptyStackException();
 }
 obj = arr[top--];
 return obj;
}
```

The `pop` method checks to see whether any elements are on the stack. If the stack is empty (`top` is equal to 0), `pop` instantiates a new `EmptyStackException` object (a member of `java.util`) and throws it. A user can throw only objects that inherit from the `java.lang.Throwable` class. See exception class hierarchy diagram.

As soon as the `throw` statement is encountered, the runtime system checks for a matching catch block which can accept an object to be thrown by the `throw`. If it can be handled, control is transferred to the catch block. If it does not match, the object is passed to the next `catch` block if present. In case no matching `catch` block is found, default Java runtime handler handles the exception by printing the stack trace.

```
/*PROG 11.6 DEMO OF THROW VER 1 */

class JPS6
{
 public static void main(String args[])
 {
 try
 {
 throw new ArithmeticException("Hello from
 throw");
 }
 catch(ArithmeticException E)
 {
 System.out.println("Exception caught:\n"+E);
 System.out.println(E.getMessage());
 }
 }
}
```

OUTPUT:

```
Exception caught:
java.lang.ArithmaticException: Hello from throw
Hello from throw
```

*Explanation:* In the program, an exception of type `ArithmaticException` is intentionally thrown using the constructor method which takes an argument of `String` type. The thrown exception object is caught by the `catch` block and stored in `E`. The `E.getMessage()` shows the `String` argument append along with the internally generated message.

**3. Rethrowing an exception:** Sometimes one might want to rethrow the exception just caught, particularly, when `Exception` is used to catch any exception. Since the reference to the current exception is already there, that reference can simply be rethrown:

```
catch(Exception e)
{
 System.out.println("An exception was thrown ");
 throw e;
}
```

Rethrowing an exception causes it to go to the exception handlers in the next higher context. Any further catch clauses for the same try block are still ignored. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

**4. Catching any exception:** It is possible to create a handler that catches any type of exception. It is done by catching the base-class type exception (there are other types of base exceptions, but `Exception` is the base that is pertinent to virtually all programming activities):

```
catch(Exception e)
{
 System.out.println("Caught an exception");
}
```

This will catch any exception, so if you use it, you will want to put it at the end of your list of handlers to avoid preempting any exception handlers that might otherwise follow it. Now one comes to know why this class has been used with catch blocks in some of the programs in other chapters.

```
/*PROG 11.8 CATCHING ALL EXCEPTION VER 1 */

class JPS8
{
 public static void main(String[]args)
 {
 int num = 10;
 for(num = 10;num<=30;num+=10)
 {
 try
 {
 if(num == 20)
 throw new ArithmaticException("Arithme
tic");
 }
 }
 }
}
```

```

 else if(num<20)
 throw new RuntimeException("Run
 time");
 else if(num>20)
 throw new NullPointerException("Null
 Pointer");
 }
 catch(Exception E)
 {
 System.out.println("Caught an
 exception");
 System.out.println(E.getMessage());
 }
}
}

OUTPUT:

Caught an exception
Run time
Caught an exception
Arithmetric
Caught an exception
Null Pointer

```

*Explanation:* Since the Exception class is the base of all the exception classes that are important to the programmer, one does not get much specific information about the exception, but the method that comes from its base type throwable, can be called. One such method seen earlier is `getMessage()`, which returns a String containing the description of the exception object. Rest is self-explanatory.

```
/* PROG 11.9 CATCHING ALL EXCEPTION VER 2 */
```

```

class demo1 extends Exception
{
}
class demo2 extends Exception
{
}
class demo3 extends Exception
{
}
class JPS9
{
 public static void main(String[]args)
 {
 int num=10;
 for(num=10;num<=30;num+=10)
 {
 try
 {

```

```

 if (num==20)
 throw new demo1();
 else if(num<20)
 throw new demo2();
 else if(num>20)
 throw new demo3();
 }
 catch(Exception E)
 {
 System.out.println("Caught an
 exception");
 }
}
}

OUTPUT:
Caught an exception
Caught an exception
Caught an exception

```

*Explanation:* Program is self-explanatory.

**5. The throw clause:** In Java, when a method does not handle all the exceptions it can throw, it must specify all such exceptions in the declaration of the method using throws clause. This is the exception specification and it is part of the method declaration, appearing after the argument list. The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. All the exceptions which are not of type classes error or RuntimeException or any of their subclasses, must be specified. The clause goes after the method name and argument list and before the brace that defines the scope of the method; an example is shown below.

```
public void fun()throws IOException, IllegalAccessException
```

The above function fun declares that the function fun can throw exceptions of type IOException and IllegalAccessException type. The client of the method must write code to guard against these exceptions.

The general syntax is as follows:

```
accessSpecifier return_type method_name(parameterlist) throws
exceptionclass1, exceptionclass2,...exceptionclassN
{
 //definition of the method
}
```

If the code within your method causes exceptions, but your method does not handle them, the compiler will detect this and will either tell that you must handle the exception or indicate with an exception specification that it may be thrown from your method. By enforcing exception specifications from top to bottom, Java guarantees that a certain level of exception correctness can be ensured at compile time.

One such example seen earlier is the function readLine which have been used in a number of programs. The function throws an exception of type IOException. The signature of the method is as shown below:

```
public final String readLine()throws IOException
```

So, if the Java code is written in the following way

```
import java.io.*;
class JPS1
{
 public static void main(String[] args)
 {
 String str;
 DataInputStream input;
 input = new DataInputStream(System.in);
 str = input.readLine();
 }
}
```

the program will not compile and the following error will be generated:

```
C:\>javac JPS1.java
JPS1.java:9: unreported exception java.io.IOException; must be
caught or declared to be thrown
 str = input.readLine();
 ^
Note: JPS1.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
1 error
```

This error can be rectified in two ways: either specify the exception in the thrown clause or handle the exception using try catch block. Both are given below:

```
/* FIRST APPROACH */

import java.io.*;
class JPS
{
 public static void main(String args[])
 {
 try
 {
 String str;
 DataInputStream input;
 input = new DataInputStream(System.in);
 str = input.readLine();
 }
 catch(IOException E)
 {
 System.out.println("Error:"+E);
 }
 }
}
```

OUTPUT:

```
C:\>javac JPS2.java
Note: JPS2.java uses or overrides a deprecated API.
Note: Recompile with -Xlint: deprecation for details.
C:\>java JPS2
Exception in thread "main" java.lang.NoClassDefFoundError:
JPS2
```

/\* SECOND APPROACH \*/

```
import java.io.*;
class JPS3
{
 public static void main(String[]args) throws IOException
 {
 String str;
 DataInputStream input;
 input = new DataInputStream(System.in);
 str = input.readLine();
 }
}
```

Let us create our own method which throws an exception.

/\*PROG 11.10 THROWING EXCEPTION FROM A METHOD \*/

```
import java.io.*;
class JPS10
{
 static void demo() throws IOException
 {
 throw new IOException("demo of throws");
 }
 public static void main(String args[])
 {
 try
 {
 demo();
 }
 catch (IOException E)
 {
 System.out.println("Exception Caught");
 System.out.println(E.getMessage());
 }
 }
}
```

OUTPUT:

```
Exception Caught
demo of throws
```

*Explanation:* The method demo declares using throws clause that it can throw an exception of IOException class type. In the main inside try block, the demo method is called. As the method can throw an exception, it is placed it in the try block. The method when called throws an exception of type IOException as follows:

```
new IOException("demo of throws");
```

The exception object thrown is caught by the catch block and the program terminates gracefully:

```
import java.io.*;
class JPS11
{
 static void demo() throws IOException
 {
 throw new IOException("demo of throw");
 }
 public static void main(String[] args) throws
 IOException
 {
 demo();
 }
}
```

Then the program will compile but will generate runtime error as follows:

```
C:\JPS\Ch11>javac JPS11.java
C:\JPS\Ch11>java JPS11
Exception in thread "main" java.io.IOException: demo of throw
at JPS11.demo(JPS11.java:6)
at JPS11.main(JPS11.java:10)
```

This is because the exception declared in the throw clause of demo function was not handled. Declaring the exception in the function main using throws clause can only prevent from compilation error but not from runtime error.

**6. Nesting of try-catch block:** Just like the multiple catch blocks, there can also be multiple try blocks. These try blocks may be written independently or can nest within each other, i.e., keep one try-catch block within another try block. The program structure for nested try statement is look like the following:

```
try
{
 //statements
 //statements
 try
 {
 //statements
 //statements
 }
}
```

```

 }
 catch(<exception_two> obj)
 {
 //statements
 }
 //statements
 //staements
}
catch(<exception_two> obj)
{
//statements
}

```

The following example illustrates the above.

```

/*PROG 11.11 DEMO OF NESTED TRY-CATCH BLOCK */

class Nested_Try
{
 public static void main(String args[])
 {
 try
 {
 int a = Integer.parseInt(args[0]);
 int b = Integer.parseInt(args[1]);
 int quot = 0;
 try
 {
 quot = a / b;
 System.out.println(quot);
 }
 catch (ArithmetricException e)
 {
 System.out.println("divide by
 zero");
 }
 }
 catch (NumberFormatException e)
 {
 System.out.println("Incorrect argument
 type");
 }
 }
}

```

The output of the program is as follows:

- (i) If the arguments are entered properly, like the following:

C:\JPS\Ch11>java Nested\_Try 24 6 4

- (ii) If the argument contains a string than the number or vice versa:

```
C:\JPS\Ch11>java Nested_Try 24 bb
Incorrect argument type
```

- (iii) If the second argument is entered zero:

```
C:\JPS\Ch11>java Nested_Try 24 0 divide by zero
```

*Explanation:* In the program, two numbers are accepted from the command line. After that, the command line arguments which are in the string format, are converted to integers. If the numbers were not received properly in a number format, during the conversion a `NumberFormatException` is raised, otherwise the control goes to the next `try` block. Inside this second **try-catch** block, the first number is divided by the second number, and during the calculation if there is any arithmetic error, it is caught by the inner catch block.

**7. The finally clause:** There is often some piece of code that one wants to execute whether or not an exception is thrown within a try block. This usually pertains to some operation other than memory recovery (since that is taken care of by the garbage collector). To achieve this effect, a finally clause is used at the end of all the exception handlers. The syntax of try-catch-finally is presented here once again:

```
try
{
 //The guarded region:harmful activities
 //that might throw A, B or C.
}
catch(A a1)
{
 //Handler for argument
}
catch(B b1)
{
 //Handler for situation B
}
catch(C c1)
{
 //Handler for situation C
}
finally
{
 //Activities that happen every time
}
```

The `finally` clause, similar to `catch` and `try`, is a block of code by the name `finally`. The `finally` executes all the time irrespective of whether an exception is thrown or not. Even if an exception is thrown and handled or not, `finally` will execute. The `finally` block is guaranteed to execute after the `try-catch` block. The `finally` clause is an optional, and its usage is up to the programmer. If `try-catch` block is placed inside a method, before returning from the method, either maturely or prematurely, `finally` clause will execute. The `finally` clause is necessary in some kind of cleanup like an open file or network connection, something you have drawn on the screen, etc.

```
/*PROG 11.12 DEMO OF FINALLY, EXCEPTION CAUGHT */

class JPS12
{
 public static void main(String[] args)
 {
 try
 {
 throw new NullPointerException();
 }
 catch (NullPointerException E)
 {
 System.out.println("Exception Caught");
 }
 finally
 {
 System.out.println("Finally executes");
 }
 }
}

OUTPUT:
Exception Caught
Finally executes
```

*Explanation:* Program is simple; from within try block, an exception object of NullPointerException is thrown. This exception object is handled by the catch block. After that finally executes.

```
/*PROG 11.13 DEMO OF FINALLY, EXCEPTION NOT CAUGHT */

class JPS13
{
 public static void main(String[] args)
 {
 try
 {
 throw new NullPointerException();
 }
 finally
 {
 System.out.println("Finally executes");
 }
 }
}

OUTPUT:
Finally excutes
Exception in thread "main" java.lang.NullPointerException at
JPS13.main(JPS13.java:8)
```

*Explanation:* The aim of the program is to simply show that whether or not an exception is thrown, finally block executes.

```
/*PROG 11.14 DEMO OF FINALLY RETURNING FROM METHOD */

class JPS14
{
 static void fun()
 {
 try
 {
 return;
 }
 finally
 {
 System.out.println("\nFinally executes");
 }
 }
 public static void main(String[] args)
 {
 fun();
 }
}
OUTPUT:
Finally executes
```

*Explanation:* In the static function fun, only the try and finally block are written and not catch block. In the try block, we simply returned from the function. But note before returning from the function, the finally block executes and prints Finally executes.

## 11.8 CREATING YOUR OWN EXCEPTION

To create your own exception, you will need to make Exception or Throwable class as the base class of the class which you are going to create. We have seen that Throwable class is the top most class in the exception class hierarchy, so all the methods of Throwable class can be used by the derived class. Some of the methods of Throwable class you can use are as follows:

1. `String getMessage()` : Returns the description of the exception object thrown.
2. `void printStackTrace()` : Displays the stack trace.
3. `String toString()` : Converts a class object into String object.

Even the methods can be overridden as per your requirement. Two programs are given below:

```
/*PROG 11.15 DEMO OF CREATING OUR OWN EXCEPTION VER 1 */

class demo extends Exception
{
 demo(String s)
 {
```

```

 super(s);
 }
}
class JPS15
{
 public static void main(String[] args)
 {
 try
 {
 throw new demo("\nHello from own
 exception");
 }
 catch (demo E)
 {
 System.out.println(E.getMessage());
 }
 }
}

OUTPUT:
Hello from own exception

```

*Explanation:* In the program, a class demo and extend class Exception are created. In the one argument constructor of String type, the argument to construct of super class is sent, i.e., Exception using super. In the main, the exception of demo class is thrown as throw new demo ("Hello from exception"); . The thrown object is caught by the catch block in object E. The E.getMessage displays the String argument of the object passed as argument.

```
/*PROG 11.16 DEMO OF CREATING OUR OWN EXCEPTION VER 2 */
```

```

class MyException extends Exception
{
 private int num;
 MyException(int a)
 {
 num = a;
 }
 public String toString()
 {
 return "MyException Thrown with num =" + num;
 }
}
class JPS16
{
 static void check(int x) throws MyException
 {
 System.out.println("\ncheck called x =" + x);
 if (x < 0)
 throw new MyException(x);
 System.out.println("Returning from function");
 }
}
```

```

 }
 public static void main(String args[])
 {
 try
 {
 check(10);
 check(-10);
 }
 catch (MyException e)
 {
 System.out.println("Exception caught " + e);
 }
 }
}

OUTPUT:
check called x =10
Returning from function
check called x =-10
Exception caught MyException Thrown with num =-10

```

*Explanation:* The MyException class has one int argument constructor which initializes the num. The function has `toString` method overridden, which is called automatically when displaying an object of MyException class to convert it into String form. In the main, when method check is called with negative argument, exception of MyException is thrown. This thrown exception is caught by the catch block in ‘e’, which is displayed. As object cannot be displayed in this manner, but as we have `toString` method defined in the MyException class, this method is called and string “MyException Thrown with num 5 210” is returned.

## 11.9 PONDERABLE POINTS

1. Exception is a runtime error which may occur in the program.
2. An exception if not handled may terminate the program abnormally.
3. To deal with exceptions, try-catch blocks are used.
4. An exception in programming term is considered an object which is thrown.
5. Any program code or function which may generate exception is placed in the try block. The exception thrown is caught by catch block.
6. Every try block must have the corresponding catch block or finally block. The exception thrown is caught by the catch block. If catch block is unable to handle the exception, the program may terminate prematurely.
7. The root class of all exception class is the `Throwable` class.
8. Checked exceptions are those exceptions which either must be caught or declared in the throw clause.
9. For one try block, there may be multiple catch blocks.
10. The finally block executes regardless of an exception is thrown or not.
11. A function can specify what type of exception can be thrown by specifying in the function declaration using throws clause.
12. Objects of user-defined classes as exception can also be thrown.

13. The advantages of exceptions are as follows:
- Separating error-handling code from 'regular' code.
  - Propagating errors up the call stack.
  - Grouping and differentiating error types.

## REVIEW QUESTIONS

- Explain how exception handling mechanism can be used in a program.
- How many catch blocks and finally blocks can we use with try block?
- How do you define try and catch block?
- What is unchecked and checked exception?
- Define an exception called "NoEqualException" that is thrown when a float value is not equal to 3.14. Write a program that uses the above user defined exception.
- Write a method

```
private static String readExistingFilename()
```

That keeps reading String from the user until it receives the name of an existing file, then, returns that String.

- Write a program that reads the name of a file and computes the number of lines in the file that represent integers. Use `Integer.parseInt` to parse complete line read from the file. Use Java's

- Exception mechanism to detect when a given line is not a valid int.
- Can a try block have two finally block? Prove it.
- Can a catch follow a finally block? Prove it.
- Does Java permit a try block to have two catch blocks for the exact same exception? Prove it.
- Give code for which two catch blocks are triggered by the same exception. Prove that only the first catch block is executed.
- Define an exception called "NoMatchException" that is thrown when a string is not equal to "India". Write a program that uses this exception.
- Write a program to demonstrate the use of exception handling.
- Write a program for user defined exception that checks the internal and external marks; if the internal marks is greater than 40 it raises the exception "Internal marks is exceed"; if the external marks is greater than 60 it raises the exception and displays the message "The External Marks is exceed". Create the above exception and use it in your program.

## Multiple Choice Questions

- For one try block there may be
  - only 1 catch block
  - only 2 catch blocks
  - multiple catch blocks
  - none of the above
- \_\_\_\_\_ class works as base class for an entire family of classes, declared in `java.lang`, that the program can initiate and throw.
  - Exception
  - RuntimeException
  - InterruptedException
  - Throwable
- For throwing the more serious problems, such as Out of Memory Error, may not be so easy to handle. Which of the following will be thrown?
  - Exception
  - ArithmeticeException
- String `getMessage()` is used to
  - display the stack trace
  - returns the description of the exception object thrown
  - both (a) and (b)
  - none of the above
- The correct statement for stack trace is
  - A class in `java.lang` package
  - An object of Exception class
  - A useful debugging tool
  - None of the above

7. NumberFormatException class shows  
(a) invalid conversion of a string to a numeric format  
(b) invalid conversion of a numeric format to a string  
(c) invalid conversion of a character to a numeric format  
(d) invalid conversion of a numeric format to a character
9. String `toString()` method is used to convert  
(a) an integer data to string  
(b) a class object into String object  
(c) String object in a class object  
(d) None of the above
10. The code give below:
- ```
import java.lang.*;
class test
{
    public static void main(String[] args)
    {
        try
        {
            int[] num = { 0, 1, 2, 3, 4 };
            int x = 25 / num[5];
            System.out.println("Control cannot reach here");
        }
        catch (ArrayIndexOutOfBoundsException E)
        {
            System.out.println("Exception thrown:");
            E.printStackTrace();
        }
    }
}
```
- (a) Throws an exception at `test.main(excep3.java:10)`
(b) Throws an exception at `test.main(excep2.java:9)`
(c) Throws an exception at `test.main(excep1.java:8)`
(d) Throws an exception at `test.main(excep4.java:10)`

KEY FOR MULTIPLE CHOICE QUESTIONS

1. c 2. d 3. c 4. c 5. b 6. c 7. a 8. b 9. b 10. b

Threads in Java

12

12.1 INTRODUCTION

Running multiple programs/task on a computer is termed as multitasking. For example, running an MS-Word application and Visual C++ application at the same time is an example of multitasking. Each running instance of a program in memory is known as a process. Each process has a single unit of execution which is a thread. Thus, running multiple programs from user's point of view is multitasking, but from operating system's point of view, it is multithreading, which is also known as concurrent programming.

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. A multitasking operating system is capable of running more than one process (program) at a time by periodically switching the CPU from one task to another. A thread is a single sequential flow of control within a process. A single process can thus have multiple concurrently executing threads. Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process—every process has at least one thread. Threads share the resources of the process, including memory and open files.

All the programs seen so far were single threaded programs. That is, they were having single flow of execution. A conventional process cannot continue performing its operations and at the same time return control to the rest of the program. In multithreading, there are several independent threads running concurrently in a program. Each thread provides a separate path of execution. From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time; however, it is not the same as starting a program twice and saying that there are multiple lines of a program being executed at the same time. In this case, the operating system is treating the programs as two separate and distinct processes. In multithreading, CPU time given to a process is divided into all the threads the process is having.

Multithreading is a remarkable new feature in Java not supported by C/C++. Running program concurrently has a number of advantages. Multiple tasks can be divided into a number of tasks and can run parallel. This greatly enhances a system's capacity for concurrent execution of processes and threads. Threads enable you to create a more loosely coupled design revolving around a single thread. Many powerful applications related to graphics, animation, sound, etc. can be written efficiently using threads.

12.2 CREATING THREADS

In Java, there are two ways to create a thread.

1. The first method is to inherit the class `Thread` and make it a base class for your class. Extending a class is the way Java inherits methods and variables from a parent class. In this case, one can only

extend or inherit from a single parent class. The `Thread` class is defined in the package `java.lang`. The `Thread` class has a number of methods, a class can use, but the most important method is the `run` method which must be overridden to make the thread running. Thus, `run()` is the code that will be executed ‘simultaneously’ with the other threads in a program. Its simple syntax is as follows:

```
public void run()
{
    .....
    .....
}
```

2. The second method is to implement `Runnable` interface, which is the frequently used method for implementing threads in the programs. The interface has just one method `run` which must be overridden. It can be used as follows:

```
class Mythread implements Runnable
{
    public void run()
    {
        .....
        .....
    }
}
```

Both the methods of creating threads are used.

12.2.1 Extending Thread Class

Creating a thread using this method simply involves creating a new class which extends the `Thread` class and overriding `run` method. To execute `run` method, `start` will be called, `run()` method to start the execution of thread. Consider the following program that creates two threads.

```
/* PROG 12.1 DEMO OF THREAD VER 1 */

class demo extends Thread
{
    public void run()
    {
        for (int i = 0; i < 5; i++)
            System.out.println(getName() + " i = " + i);
    }
    demo()
    {
        start();
    }
}
class JPS1
{
    public static void main(String[] args)
    {
        demo d1 = new demo();
        demo d2 = new demo();
```

```

        }
    }

OUTPUT:

Thread-0 i = 0
Thread-1 i = 0
Thread-0 i = 1
Thread-0 i = 2
Thread-0 i = 3
Thread-0 i = 4
Thread-1 i = 1
Thread-1 i = 2
Thread-1 i = 3
Thread-1 i = 4

```

Explanation: In the program, the class Thread is simply inherited into class demo, and demo is made a thread class. In the default constructor of this class, the start method is called which in turn calls the run method. In the run method, a loop is run from 0 to 4. The getName method defined in the class Thread returns the name of the running thread. In the main, two threads d1 and d2 are created. Writing demo d1 = new demo() calls the default constructor of the class demo, and through start method run is called. Same process applies for d2. Now, both threads run concurrently and execute code in their run method. Each time the program is run, one might get different output. For better understanding, the limit of for loop may be increased from 10 to, for example, 15 or 20.

```

/*PROG 12.2 DEMO OF THREAD VER 2 */

class demo extends Thread
{
    public void run()
    {
        try
        {
            for (int i = 0; i < 5; i++)
            {
                System.out.println(getName() + "i=" + i + " ");
                Thread.sleep(1000);
            }
            System.out.println(getName() + " Fineshes");
        }
        catch (InterruptedException E)
        {
            System.out.println("Error: " + E);
        }
    }
}

class demo1 extends Thread
{
    public void run()
    {

```

```
try
{
    for (int i = 0; i < 5; i++)
    {
        System.out.println(getName() + "i=" + i + " ");
        Thread.sleep(1000);
    }
    System.out.println(getName() + " Finishes");
}
catch (InterruptedException E)
{
    System.out.println("Error:" + E);
}
}

class demo2 extends Thread
{
    public void run()
    {
        try
        {
            for (int i = 0; i < 5; i++)
            {
                System.out.println(getName() + "i=" + i + "'");
                Thread.sleep(1000);
            }
            System.out.println(getName() + " Finishes");
        }
        catch (InterruptedException E)
        {
            System.out.println("Error:" + E);
        }
    }
}

class JPS2
{
    public static void main(String[] args)
    {
        demo d1 = new demo();
        demo d2 = new demo();
        demo d3 = new demo();
        d1.setName("Thread d1");
        d2.setName("Thread d2");
        d3.setName("Thread d3");
        d1.start();
        d2.start();
        d3.start();
    }
}
```

OUTPUT:

```

Thread d2 i = 0
Thread d3 i = 0
Thread d1 i = 0
Thread d3 i = 1
Thread d1 i = 1
Thread d2 i = 1
Thread d2 i = 2
Thread d1 i = 2
Thread d3 i = 2
Thread d2 i = 3
Thread d1 i = 3
Thread d3 i = 3
Thread d1 i = 4
Thread d3 i = 4
Thread d2 i = 4
Thread d1 Fineshes
Thread d2 Fineshes
Thread d3 Fineshes

```

Explanation: This time, three separate classes, demo1, demo2 and demo3, are created each of which extends the Thread class and overrides run method. In the main function, setName method of Thread class is made use of and names for the threads set. In the run method of each class, Thread.sleep method is made use of which takes a single parameter timeout of long type indicating time in millisecond. Thus, Threads d1, d2 and d3 sleep for 1000 milliseconds, 900 milliseconds and 800 milliseconds, respectively. Again for better understanding, increase the limit in the for loop from 5 to 10 or 15.

12.2.2 Thread Methods

The Thread class encapsulates a number of useful methods which can be used for controlling the way the thread behaves. Some of them are discussed below:

1. **public static Thread currentThread()**

This method returns a reference to the currently executing thread object.

It can be used as follows:

```

class JPS
{
    public static void main(String args[])
    {
        Thread cur_thread = Thread.currentThread();
        System.out.println("Current Thread:= " + cur_thread);
    }
}

```

The output will be: Current Thread:= Thread[main,5,main]

The output shows from left: the name of the thread, its priority and the group to which the thread belongs. The default name and priority is main and 5, respectively. The thread group is a collection of similar types of threads controlled by the runtime environment.

2. **public final String getName()**

This method returns name of the thread as seen in earlier programs. The default name of the thread is Thread-0, Thread-1 and so on. The name of the thread can be changed using the setName

method which assigns new name to the thread. Both `getName` and `setName` methods have been used earlier in the programs.

3. **`public final int getPriority()`**

This method returns the priority of the thread. Thread priorities are discussed later in the chapter.

4. **`public final boolean isAlive()`**

This method tests if this thread is alive. A thread is alive if it has been started and has not yet died. A true value indicates thread is alive and false means thread is dead.

5. **`public final void join() throws InterruptedException`**

This method waits for the thread to terminate. Its usage will be shown later using program. The other overloaded form of the method is as follows:

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

It waits for time (expressed in millisecond) to terminate. A 0 indicates wait forever.

6. **`public static void yield()`**

This method causes the currently executing thread to temporarily pause and allow other threads to execute. Its usage will be shown later.

7. **`public final void stop()`**

This method stops and kills a running thread. Currently, the thread does not stop unless it is running. If it is suspended, it does not die until it starts running again. The method is deprecated. For reasons, check out Java documentation.

The other methods `suspend`, `resume`, `wait` and `notify`, etc. are discussed later on in this chapter. Given below are few programs, which make use of some of the methods discussed above.

```
/*PROG 12.3 DEMO OF YIELD METHOD */

class JPS3 extends Thread
{
    private int count = 5;
    private static int TCount = 0;
    JPS3()
    {
        super(" " + ++TCount); ;
        start();
    }
    public void run()
    {
        while (true)
        {
            System.out.println("#" + getName() + " : " + count);
            if (--count == 0) return; //yield();
        }
    }
    public static void main(String[] args)
    {
        for (int i = 0; i < 3; i++)
            new JPS3();
    }
}
```

OUTPUT:

```
# 3 : 5
# 1 : 5
# 2 : 5
# 1 : 4
# 3 : 4
# 1 : 3
# 2 : 4
# 1 : 2
# 3 : 3
# 1 : 1
# 2 : 3
# 3 : 2
# 2 : 2
# 3 : 1
# 2 : 1
```

Explanation: In the program, three threads are created using new JPS3() and for loop. In the constructor, the super passes the name of the thread as String by concatenating it with TCount(). The run method of thread is called as start() is the next statement in the constructor. Each thread runs for its entirety and display, five values of count are displayed for thread 1 followed by thread 2 and thread 3. In the program, now remove the comment before yield. The new output is as shown below:

```
# 1 : 5
# 3 : 5
# 2 : 5
# 3 : 4
# 1 : 4
# 3 : 3
# 2 : 4
# 3 : 2
# 1 : 3
# 3 : 1
# 2 : 3
# 1 : 2
# 2 : 2
# 1 : 1
# 2 : 1
```

This is because after displaying one value of count, the first thread gives chance to second thread, which again on displaying one value of count gives chance to third thread which on displaying one value of count gives chances to first thread. Thus, all the three threads display one value of count in each turn just because of yield method. This is also known as **round robin** method in which each thread runs for a fixed amount of time and as the time is over gives chance to next waiting thread and sets at the back of the thread queue.

```
/*PROG 12.4 DEMO OF JOIN METHOD */
```

```
class JPS4 extends Thread
{
    JPS4(String str)
```

```

    {
        super(str);
        start();
    }
    public void run()
    {
        try
        {
            System.out.println("\nHello1 from thread" +
                getName());
            Thread.sleep(500);
            System.out.println("\nHello2 from thread " +
                getName());
        }
        catch (InterruptedException E)
        {
            System.out.println("Error:" + E);
        }
    }
    public static void main(String[] args) throws Exception
    {
        new JPS4("First").join();
        new JPS4("Second").join();
        new JPS4("Third").join();
    }
}

OUTPUT:

Hello1 from thread First
Hello2 from thread First
Hello1 from thread Second
Hello2 from thread Second
Hello1 from thread Third
Hello2 from thread Third

```

Explanation: The sleep method inside the run method causes enough delay so that another thread may get a chance to run, but due to join method each thread runs till it does not finish. That is why the output indicating full run of first thread is followed by second and third thread.

Without join method, the output will be as follows:

```

C:\JPS\ch12>java JPS4
Hello1 from thread First
Hello1 from thread Second
Hello1 from thread Third
Hello2 from thread First
Hello2 from thread Second
Hello3 from thread Third
C:\JPS\ch12>

```

```

/*PROG 12.5 DEMO OF ISALIVE METHOD */

class JPS5 extends Thread
{
    JPS5(String str)
    {
        super(str);
        start();
    }
    public void run()
    {
        try
        {
            System.out.println("Hello from thread " +
                               + getName());
            Thread.sleep(500);
            System.out.println("Bye from thread " +
                               + getName());
        }
        catch (InterruptedException E)
        {
            System.out.println("Error:" + E);
        }
    }
    public static void main(String[] args) throws Exception
    {
        JPS5 d1 = new JPS5("First");
        JPS5 d2 = new JPS5("Second");
        JPS5 d3 = new JPS5("Third");
        System.out.println(d1.getName()+" Alive =
                           "+d1.isAlive());
        System.out.println(d2.getName()+" Alive =
                           "+d2.isAlive());
        System.out.println(d3.getName()+" Alive =
                           "+d3.isAlive());
        System.out.println("Main Thread is sleeping");
        System.out.println("for 1 seconds");
        Thread.sleep(3000);
        System.out.println(d1.getName()+" Alive =
                           "+d1.isAlive());
        System.out.println(d2.getName()+" Alive =
                           "+d2.isAlive());
        System.out.println(d3.getName()+" Alive =
                           "+d3.isAlive());
    }
}

OUTPUT:

First Alive = true
Hello from thread First

```

```
Hello from thread Third
Hello from thread Second
Second Alive = true
Third Alive = true
Main Thread is sleeping
for 1 seconds
Bye from thread First
Bye from thread Third
Bye from thread Second
First Alive = false
Second Alive = false
Third Alive = false
```

Explanation: As explained earlier in Section 12.22, the `isAlive` method returns true if thread on which it is called `isAlive`. Initially, after starting all the three methods, calling `isAlive` method on all the threads result in `true` value from `isAlive` method. Then the `main` thread `sleeps` for 3 seconds. Next, when the `isAlive` method is called on the threads again, it returns false as the threads have done their execution and have terminated.

12.2.3 Implementing Thread Using Runnable

When thread methods are needed to be used, `Runnable` interface can be made use of for implementing threads in the programs. This is as shown in the program. For implementing threads using `Runnable`, an object of the class which implements the `Runnable` interface must be passed to `Thread` construct. This is shown as follows:

```
class demo implements Runnable
{
}
```

In main function, one will be writing like

```
Thread T = new Thread(new demo());
```

Or one can create an object of `Thread` class as member of the class implementing `Runnable` interface. The second approach is shown first.

```
/*PROG 12.6 THREAD IMPLEMENTATION USING RUNNABLE VER 1 */
```

```
class demo implements Runnable
{
    private int count = 5;
    Thread thr;
    demo(String s)
    {
        thr = new Thread(this, s);
        thr.start();
    }
    public void run()
    {
        while (true)
        {
```

```

        System.out.println("#"+thr.getName()+":"+count);
        if (--count == 0) return;
    }
}
class JPS6
{
    public static void main(String[] args)
    {
        for (int i = 0; i < 3; i++)
            new demo("RUNNABLE THREAD " + (i + 1));
    }
}

OUTPUT:
#RUNNABLE THREAD 1:5
#RUNNABLE THREAD 3:5
#RUNNABLE THREAD 2:5
#RUNNABLE THREAD 3:4
#RUNNABLE THREAD 1:4
#RUNNABLE THREAD 3:3
#RUNNABLE THREAD 2:4
#RUNNABLE THREAD 3:2
#RUNNABLE THREAD 1:3
#RUNNABLE THREAD 3:1
#RUNNABLE THREAD 2:3
#RUNNABLE THREAD 1:2
#RUNNABLE THREAD 1:1
#RUNNABLE THREAD 2:2
#RUNNABLE THREAD 2:1

```

Explanation: For `i=0` in `main`, String `RUNNABLE THREAD 1` is passed to the `demo` constructor. The keyword `this` holds the reference of the current object. In the `demo` class, there is a reference of `Thread` class. The reference `this` and string `s` are passed as argument to the constructor of `Thread` class and `thr` starts referring the newly created object of `Thread` class. The next statement `thr.start` causes `run` method to be called. Same arguments apply to other two threads.

```

/*PROG 12.7 THREAD IMPLEMENTATION USING RUNNABLE VER 2*/

class demo implements Runnable
{
    private int count = 5;
    public void run()
    {
        while (true)
        {
            Thread t = Thread.currentThread();
            System.out.println("#" + t.getName() + ":" + count);
            if (--count == 0) return;
        }
    }
}

```

```

}
class JPS7
{
    public static void main(String[] args)
    {
        Thread T1 = new Thread(new demo(),"MyThread1");
        T1.start();
        Thread T2 = new Thread(new demo(),"MyThread2");
        T2.start();
    }
}

OUTPUT:
#MyThread1:5
#MyThread2:5
#MyThread1:4
#MyThread2:4
#MyThread1:3
#MyThread2:3
#MyThread1:2
#MyThread2:2
#MyThread1:1
#MyThread2:1

```

Explanation: In the program, a Thread reference is not made part of the demo class; instead the first approach has been used. In the main function, two objects of Thread class are created and in the constructor of Thread class reference of newly created demo object and thread name as string are passed.

12.3 THREAD PRIORITY

Priority for a thread can be set so that it runs before or after another thread. By default, all threads are created with normal priority. Priority makes a thread important. A higher priority thread runs first following which a lower priority thread will get a chance. In Java, Thread class provides three static fields for setting the priority of the threads:

1. public static final int NORM_PRIORITY
2. public static final int MIN_PRIORITY
3. public static final int MAX_PRIORITY

The NORM_PRIORITY is the default priority all threads have. MIN_PRIORITY is the minimum priority a thread can have. Similarly, MAX_PRIORITY is the maximum priority a thread can have. The thread priority can be set using the `setPriority` method which takes an int value as argument. The `getPriority` method returns a thread's priority.

```
/*PROG 12.8 DEMO OF THREAD PRIORITY */
```

```

class demo implements Runnable
{
    private int count = 5;
    Thread thr;

```

```

    demo(String s, int p)
    {
        thr = new Thread(this, s);
        thr.setPriority(p);
        thr.start();
    }
    public void run()
    {
        while (true)
        {
            System.out.println("#"+thr.getName() + ":" +
                               count);
            if (--count == 0) return;
        }
    }
}
class JPS9
{
    public static void main(String[] args)
    {
        new demo("RUNNABLE THREAD ONE",
                 Thread.MIN_PRIORITY);
        new demo("RUNNABLE THREAD TWO",
                 Thread.MAX_PRIORITY);
        new demo("RUNNABLE THREAD THREE",
                 Thread.NORM_PRIORITY);
    }
}

```

OUTPUT:

```

#RUNNABLE THREAD TWO      :5
#RUNNABLE THREAD ONE     :5
#RUNNABLE THREAD THREE   :5
#RUNNABLE THREAD ONE     :4
#RUNNABLE THREAD TWO     :4
#RUNNABLE THREAD ONE     :3
#RUNNABLE THREAD THREE   :4
#RUNNABLE THREAD ONE     :2
#RUNNABLE THREAD TWO     :3
#RUNNABLE THREAD ONE     :1
#RUNNABLE THREAD THREE   :3
#RUNNABLE THREAD THREE   :2
#RUNNABLE THREAD THREE   :1
#RUNNABLE THREAD TWO     :2
#RUNNABLE THREAD TWO     :1

```

Explanation: In the program, the highest priority for thread TWO, minimum priority for thread ONE and normal priority for thread THREE are set. So thread TWO, THREE and ONE run in order.

12.4 THREAD SYNCHRONIZATION

A major concern when two or more threads share the same resource is that only one of them can access the resource at one time. Programmers address this concern by synchronizing threads.

Threads are synchronized in Java through the use of a monitor. Think of a monitor as an object that enables a thread to access a resource. Only one thread can use a monitor at any one time period. The thread owns the monitor for that period of time. The monitor is also called a semaphore.

A thread can own a monitor only if no other thread owns the monitor. If the monitor is available, a thread can own the monitor and have exclusive access to the resource associated with the monitor. If the monitor is not available, the thread is suspended until the monitor becomes available. Programmers say that the thread is waiting for the monitor.

Fortunately, the task of acquiring a monitor for a resource, happens behind the scenes in Java. Java handles all the details for the programmer. One has to synchronize the threads created in the program if more than one thread will use the same resource.

There are two ways in which threads can be synchronized: the synchronized method and the synchronized statement.

12.4.1 The Synchronized Method

All objects in Java have a monitor. A thread enters a monitor whenever a method modified by the keyword synchronized is called. The thread that is first to call the synchronized method is said to be inside the method and, therefore, owns the method and resources used by the method. Another thread that calls the synchronized method is suspended until the first thread relinquishes the synchronized method.

If a synchronized method is an instance method, it activates the lock associated with the instance that called this method, which is the object known as this during the execution of the body of the method. If the synchronized method is static, it activates the lock associated with the class object that defines the synchronized method.

Before learning how to define a synchronized method in a program, here is what might happen if synchronization is not used. This is illustrated by the following example. This program displays two names within parentheses using two threads. This-step process, where the opening parentheses using two threads. This is a three-step process, where the opening parenthesis, the name and the closing parenthesis are displayed in separate steps.

```
/*PROG 12.9 WITHOUT SYNCHRONIZATION */

class Parentheses
{
    void display(String s)
    {
        System.out.println("\n");
        System.out.print("(" + s);
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println(")");
    }
}
```

```

        }
    }

class MyThread extends Thread
{
    String s1;
    Parentheses p1;
    public MyThread(Parentheses p2, String s2)
    {
        p1 = p2;
        s1 = s2;
        start();
    }
    public void run()
    {
        p1.display(s1);
    }
}
class JPS10
{
    public static void main(String args[])
    {
        Parentheses p3 = new Parentheses();
        MyThread name1 = new MyThread(p3, "MPSTME");
        MyThread name2 = new MyThread(p3, "NMIMS");
        try
        {
            name1.join();
            name2.join();
        }
        catch (InterruptedException e)
        {
            System.out.println("Interrupted");
        }
    }
}

OUTPUT:
(NMIMS (MPSTME)
)

```

Explanation: The program defines three classes: the `Parentheses` class, the `MyThread` class and the `JPS10` class, which is the program class. The `Parentheses` class defines one method called `display()`, which receives a string in its argument list and displays the string in parentheses on the screen. The `MyThread` class defines a thread. In doing so, the constructor of `MyThread` requires two arguments. The first argument is a reference to an instance of the `Parentheses` class. The second argument is a string containing the name that will be displayed on the screen. The `run()` method uses the instance of the `Parentheses` class to call its `display()` method, passing the `display()` method the name that is to appear on the screen.

The rest of the action happens in the `main()` method of the `JPS10` class. The first statement declares an instance of the `Parentheses` class. The next two classes create two threads. Notice that both threads use the same instance of the `Parentheses` class.

Here is what is displayed when this program is run. It is probably not what was expected. Each name should be enclosed within its own parentheses. The problem with the previous example is that two threads use the same resource concurrently. The resource is the `display()` method defined in the `Parentheses` class. In order to have one thread take control of the `display()` method, the `display()` method must be synchronized. This is done by using the keyword `synchronized` in the header of the `display()` method.

Now simply change the `display` method by writing `synchronized` keyword before void

```
synchronized void display(String s)
{
    .....
    .....
}
```

This time the output will be as follows:

```
(NMIMS)
(MPSTME)
```

12.4.2 The Synchronized Statement

Synchronizing a method is the best way to restrict the use of a method one thread at a time. However, there will be occasions when it will not be possible to synchronize a method, such as when encountering a class that is provided by a third party. In such cases, one does not have access to the definition of the class, which prevents one from using the `synchronized` keyword.

An alternative to using the `synchronized` keyword is the `synchronized` statement. A `synchronized` statement contains a `synchronized` block, within which is placed objects and methods that are to be synchronized. Calls to the methods contained in the `synchronized` block happen only after the thread enters the monitor of the object.

Although methods can be called within a `synchronized` block, the method declaration must be made outside a `synchronized` block.

The following example shows how to use a `synchronized` statement. This is basically the same as the previous example; however, the `synchronized` statement is used instead of the `synchronized` keyword. The `synchronized` statement is placed in the `run()` method within the `MyThread` class. The `synchronized` statement synchronizes the instance of the `Parentheses` class and thus prevents two threads from calling the `display()` method concurrently.

In the previous program, simply change the `run` method in `MyThread` class as follows:

```
public void run() {
    synchronized(p1)
    {
        p1.display(s1);
    }
}
```

Here, the `display()` method is not modified by `synchronized`. Instead, the `synchronized` statement is used inside the caller's `run()` method. This causes the same correct output as before, because each thread waits for the prior one to finish before proceeding.

12.5 THREAD COMMUNICATION

Usually all threads run at their own, independently from others, but sometimes threads have to coordinate their processing and, therefore, need to be able to communicate with each other during processing. Programmers call this **interprocess communication**.

One can have threads communicate with each other in a program by using the `wait()`, `notify()` and `notifyAll()` methods. These methods are called from within a synchronized method.

1. The `wait()` method tells a thread to relinquish a monitor and go into suspension. There are two forms of the `wait()` method: One form does not require an argument and causes a thread to wait until it is notified; the other form of the `wait()` method allows to specify the amount of time to wait. The length of time in milliseconds is specified, which is passed to the `wait()` method.
2. The `notify()` method tells a thread that is suspended by the `wait()` method to wake up again and regain control of the monitor.
3. The `notifyAll()` method wakes up all the threads that are waiting for control of the monitor. Only the thread with the highest priority is given control over the monitor. The other threads wait in suspension until the monitor becomes available again.

All these methods are part of the object class within `java.lang` package

```
final void wait() throws InterruptedException
final void wait(long timeout) throws InterruptedException
final void notify()
final void noitfyAll()
```

The following program shows how to use these methods in an application. The objective of the program is to have the `Producer` class give a value to the `Consumer` class through the use of a `Queue` class. The `Producer` class places a value on the queues and then waits until the `Consumer` class retrieves the value before the `Producer` class places another value on the queue.

```
/*PROG 12.10 SOLUTION TO PRODUCER CONSUMER PROBLEM */

class Queue
{
    int Item;
    boolean busy = false;
    synchronized int get()
    {
        if (!busy)
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                System.out.println("Get:
                    InterruptedException");
            }
        System.out.println("Get: " + Item);
        notify();
        return Item;
    }
    synchronized void put(int Item)
    {
        if (busy)
            try
            {
```

```
        wait();
    }
    catch (InterruptedException e)
    {
        System.out.println("Put:
                           InterruptedException");
    }
    this.Item = Item;
    busy = true;
    System.out.println("Put: " + Item);
    notify();
}
}
class Producer extends Thread
{
    Queue q;
    Producer(Queue q)
    {
        this.q = q;
    }
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            q.put(i);
        }
    }
}
class Consumer extends Thread
{
    Queue q;
    Consumer(Queue q)
    {
        this.q = q;
    }
    public void run()
    {
        for(int i=0; i<5; i++)
        {
            q.get();
        }
    }
}
class JPS11
{
    public static void main(String args[])
    {
        Queue q = new Queue();
        Producer P = new Producer(q);
        Consumer C= new Consumer(q);
        P.start();
```

```

        C.start();
    }
}

OUTPUT:

Put: 0
Get: 0
Put: 1
Get: 1
Put: 2
Get: 2
Put: 3
Get: 3
Get: 3
Put: 4

```

Explanation: The program defines four classes: the Queue class, the Producer class, the Consumer class and the JPS11 class. The Queue class defines two data members: Item and a flag. The **Item** is used to store the value placed on the queue by the Producer. The **flag** variable is used as a sign indicating whether a value has been placed on the queue. This is set to **false** by default, which enables the producer to place a value on to the queue. The Queue class also defines a `get()` method and a `put()` method. The `put()` method is used to place a value on to the queue (that is to assign a value to the `Item` variables). The `get()` method is used to retrieve the value contained on the queue (i.e., to return the value of `Item`). Once the value is assigned, the `put()` method changes the value of the flag from false to true, indicating there is a value on the queue. Notice how the value of the flag is used within the `get()` method and the `put()` method to have the thread that calls the method wait until either there is a value on the queue or there is not a value on the queue, depending on which method is being called.

The Producer class declares an instance of the Queue class and then calls the `put()` method to place five integers on the queue. Although the `put()` method is called within `for` loop, each integer is placed on the queue and then there is a pause until the integer is retrieved by the Consumer class.

The Consumer class is very similar in design to the Producer class, except the Consumer class calls the `get()` method five times from within a `for` loop. Each call to the `get()` method is paused until the Producer class places an integer in the queue.

The `main()` method of the JPS11 class creates instances of the Queue class, the Producer class and the Consumer class. Notice that both constructors of the Producer class and the Consumer class are passed a reference to the instance of the same Queue class. They use the instance of the Queue class for inter-process communication.

Notice from the output that the value placed on the queue by the Producer is retrieved by the Consumer before the Producer places the next value on the queue.

12.6 SUSPENDED AND RESUMING THREADS

In programming with threads, the programmer may sometimes want to temporarily stop a thread processing and resume it later on. In such situations, one may use **suspend** and **resume** method provided by the Thread class. But these methods are deprecated. These are the methods used by Java 1.1 but are not supported by Java 2. The reason is that they are deadlock prone. Here, for the sake of how they were used in earlier Java programs, a small program is given which makes use of these methods. Later, a suspend and resume method will be written. The stop method for stopping a thread is also made use of. Once a thread is suspended, it can be resumed later on, but once a thread is stopped, it is dead and cannot be revived again.

```
/*PROG 12.11 DEMO OF SUSPEND, RESUME AND STOP METHOD */  
  
class MyThread extends Thread  
{  
    String name;  
    MyThread(String tname)  
    {  
        name = tname;  
        start();  
    }  
    public void run()  
    {  
        try  
        {  
            for (int i = 0; i <= 10; i++)  
            {  
                System.out.println(name + ":" + i);  
                Thread.sleep(100);  
            }  
        }  
        catch (InterruptedException e)  
        {  
            System.out.println(name + " interrupted.");  
        }  
        System.out.println(name + " exiting.");  
    }  
}  
class sus_res  
{  
    public static void main(String args[])  
    {  
        MyThread T1 = new MyThread("First");  
        try  
        {  
            Thread.sleep(500);  
            T1.suspend();  
            System.out.println("First thread suspended");  
            Thread.sleep(500);  
            T1.resume();  
            System.out.println("First thread resumed");  
            Thread.sleep(300);  
            System.out.println("First thread stopped");  
            T1.stop();  
        }  
        catch (InterruptedException e)  
        {  
            System.out.println("Main Thread Interrupted");  
        }  
        System.out.println("Main thread exiting");  
    }  
}
```

```

}

OUTPUT:

First: 0
First: 1
First: 2
First: 3
First: 4
First thread suspended
First thread resumed
First: 5
First: 6
First: 7
First thread stopped
Main thread exiting

```

Explanation: The class MyThread extends Thread class and acts like a thread. The class has just one data member String class which stores the thread name. In the main, when an object of this class is created by the name T1, it is like creating a thread. The name of the thread is passed as string in the one argument constructor of String type.

```

/*PROG 12.12 WRITING OUR OWN SUSPEND AND RESUME METHOD */

class MyThread extends Thread{
    String name;
    boolean is_suspend;
    MyThread()
    {
        is_suspend = false;
        start();
    }
    public void run()
    {
        try
        {
            for (int i = 0; i < 10; i++)
            {
                System.out.println("Thread: " + i);
                Thread.sleep(200);
                synchronized (this)
                {
                    while (is_suspend)
                    {
                        wait();
                    }
                }
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Thread: interrupted");
        }
    }
}

```

```
        }
        System.out.println("Thread exiting ");
    }
    void mysuspend()
    {
        is_suspend = true;
    }
    synchronized void myresume()
    {
        is_suspend = false;
        notify();
    }
}
class JPS14_Demo{
    public static void main(String args[]){
        MyThread t1 = new MyThread();
        try
        {
            Thread.sleep(1000);
            t1.mysuspend();
            System.out.println("Thread: Suspended");
            Thread.sleep(1000);
            t1.myresume();
            System.out.println("Thread: Resume");
        }
        catch (InterruptedException e){
        }
        try
        {
            t1.join();
        }
        catch (InterruptedException e)
        {
            System.out.println("Main Thread: interrupted");
        }
    }
}
OUTPUT:
Thread: 0
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Thread: 4
Thread: Suspended
Thread: Resume
Thread: 5
Thread: 6
Thread: 7
Thread: 8
Thread: 9
Thread exiting
```

Explanation: The program defines a MyThread class. The MyThread class defines three methods: the run(), the mysuspend() and the myresume() method. In addition, the MyThread class declares the instance variable suspended, whose value is used to indicate whether or not the thread is suspended.

The run() method contains a for loop that displays the value of the counter variable. Each time the counter variable is displayed, the thread pauses briefly. It then enters a synchronized statement to determine whether the value of the suspended instance is true. If so, the wait() method is called, causing the thread to be suspended until the notify() method is called.

The mysuspend() method simply assigns true to the suspended instance variable. The myresume() method assigns false to the suspended instance variable and then calls the notify() method. This causes the thread that is suspended to resume processing.

The main method of the JPS14_Demo class declares an instance of MyThread and then pauses for about a second before calling the mysuspend() method and displaying an appropriate message on the screen. It then pauses for about another second before calling the myresume() method and again displays an appropriate message on the screen.

The thread continues to display the value of the counter variable until the thread is suspended. The thread continues to display the value of the counter once the thread resumes processing.

12.7 DAEMON THREADS

A *daemon* thread is one that is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus, when all of the non-daemon threads complete, the program is terminated. Conversely, if there are any non-daemon threads still running, the program does not terminate.

In Java, for creating a daemon thread, setDaemon method of Thread class can be made use of. The signature of this is given as follows:

```
public final void setDaemon(boolean on);
```

Marks this thread as a daemon thread. The Java virtual machine exits when the only threads running are all daemon threads. This method must be called before the thread is started.

```
/*PROG 12.13 DEMO OF DAEMON THREAD */

class JPS15 extends Thread
{
    JPS15()
    {
        setDaemon(true); //must be called before start()
        start();
    }
    public void run()
    {
        while (true)
        {
            try
            {
                System.out.println("\nDaemon thread demo");
                sleep(100);
            }
        }
    }
}
```

```

        catch (InterruptedException e)
        {
            System.out.println("Error");
        }
    }
    public static void main(String[] args)
    {
        new JPS15();
    }
}
OUTPUT:
Daemon thread demo

```

Explanation: In `run()`, the thread is put to sleep for a while. Once all the threads are started, the program terminates immediately, before any threads can print themselves, because there are no non-daemon threads (other than `main()`) holding the program open. Thus, the program terminates by printing the 'Daemon thread demo'.

Whether a thread is a daemon, can be found out by calling `isDaemon()`. The signature of the method is given as follows:

```
public final boolean isDaemon()
```

The method tests if this thread is a daemon thread or not; if yes, then returns true.

If a thread is a daemon, any threads it creates, will automatically be daemons. See the program given below with a little modification from the above program.

```

/*PROG 12.14 DEMO OF ISDAEMON METHOD */

class JPS16 extends Thread
{
    Thread t;
    JPS16()
    {
        setDaemon(true); //Must be called before start()
                        start();
    }
    public void run()
    {
        while (true)
        {
            try
            {
                t = new Thread("Daemon Thread");
                System.out.println("\nIsDaemon = " + t.isDaemon());
                sleep(100);
            }
        }
    }
}

```

```

        catch (InterruptedException e)
        {
            System.out.println("Error");
        }
    }
}

public static void main(String[] args)
{
    new JPS16();
}
}

OUTPUT:

IsDaemon = true

```

Explanation: In the class, there is an object of Thread class as a member of JPS16 class. In the constructor of the JPS16 class, the setDaemon method marks this thread as daemon thread. Although the Thread object t was not a daemon thread yet isDaemon method on t returns true. This proves the point stated earlier.

12.8 PONDERABLE POINTS

1. A thread is the smallest unit of execution.
2. Execution of multiple tasks from a user's point of view is multitasking and from an operating system's point of view is multithreading.
3. Each thread is a part of a process which is an instance of running program.
4. A thread is sometimes known as light weight process.
5. A thread can be created in Java either by extending Thread class or by implementing Runnable interface.
6. All classes which either extend Thread class or implement Runnable interface have to implement run method which executes as a thread.
7. The start method automatically calls the run method.
8. In case, the method provided by the Thread class is not needed to be used, Runnable interface can be made use of for implementing a thread.
9. The yield method of Thread class gives chance to other waiting threads.
10. The join method called on a thread waits till this thread finishes its execution.
11. The sleep method can be used for giving delay in execution.
12. Suspending a thread means suspending the processing temporarily. The suspended thread can be resumed later on by resuming it.
13. The stop method called on thread stops it from processing and makes it dead. A dead thread cannot be resumed.
14. For communication between threads, wait and notify method can be used.
15. For thread synchronization, synchronized keyword can be used; synchronized keyword can be applied on a method or on an object.
16. For synchronization, Java uses the built-in monitor construct. Only one thread at a time can be within the monitor at a time.

REVIEW QUESTIONS

1. Explain the life cycle of the Thread.
2. What are the methods described in Thread class?
3. Explain the terms thread, multithreading, and multitasking?
4. How is a Thread created? Explain in detail.
5. Write short note on
 - (a) Thread group.
 - (b) Daemon thread
 - (c) Deadlock.
6. What are the Thread priorities available in Thread class?
7. Explain the characteristics of the thread.
8. Write a program to implement Runnable class to create a Thread.
9. What is synchronization? Explain with suitable example.
10. How can you specify the delay time in your program?
11. What is the importance of thread synchronization in multithreading?
12. Is it necessary to synchronize single thread application?
13. Write a program to:
 - (a) Print the name, priority, and Thread group of the thread.
 - (b) Change the name of the current thread to “JAVA”.
 - (c) Display the details of the current thread.
14. How do we start a thread?
15. What are the two methods by which we may stop threads?
16. What is the difference between suspending and stopping a thread?

17. What will be the output of the following program?

```
class test extends Thread
{
    public void run()
    {
        System.out.println("start");
        yield();
        resume();
        System.out.println("re-start");
        stop();
        resume();
        System.out.println("not start");
    }
    public static void main(String[] args)
    {
        test t = new test();
        t.start();
    }
}
```

Multiple Choice Questions

1. A thread can be created in Java either by extending _____ or by implementing _____.
 (a) thread class, runnable interface
 (b) process, runnable interface
 (c) suspend, resume
 (d) none of the above
2. All classes created in Java have to implement _____ which executes a thread
 (a) suspend method
 (b) notify
 (c) run method
 (d) is Alive method

3. In case we do not want to use methods provided by the Thread class we can simply make use of _____ for implementing a thread.
(a) daemon thread (c) runnable interface
(b) thread priority (d) none of the above
4. The _____ method automatically calls the _____ method.
(a) suspend, start
(b) wait, isAlive
(c) wait, getName
(d) start, run
5. Which method of Thread class gives chance to other waiting thread
(a) suspend (c) yield
(b) resume (d) run
6. Which method is used for giving delay in execution
(a) wait (c) sleep
(b) suspend (d) notify
7. Suspending the thread means suspending the
- (a) giving delay in execution
(b) priority of the thread
(c) communication among the threads
(d) processing temporarily
8. For making thread dead
(a) stop method is used
(b) suspend method is used
(c) dead method is used
(d) none of the above
9. For communication between threads _____ can be used
(a) wait and run (c) wait and resume
(b) wait and notify (d) none of the above
10. A “daemon” thread is one that is supposed to provide a general service
(a) in communication
(b) in synchronization
(c) in the background
(d) none of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. c 3. c 4. d 5. c 6. c 7. d 8. a 9. b 10. c

13

Streams and Files

13.1 INTRODUCTION

A stream is a sequence of bytes. When data is received by some device (e.g., a keyboard to the program), the stream is termed as input stream and when data is sent by the program to the output device (e.g., a screen), the stream is termed as output stream. A stream provides a sort of abstraction as will be seen. All streams behave in the same manner whether data come from some device (e.g., from a keyboard or from some network socket). Similar argument applies for output stream. It behaves the same for screen, network socket or any other output device. We have mentioned in all the earlier programs that for printing something onto the screen we have made use of System.out.println. Similarly for taking input from the user, Scanner class and System.in have been used. In fact the System.in represents the standard input stream, that is, the keyboard and System.out represent the standard output stream—the monitor/screen. As discussed in the beginning of the book, System.in is an object of class InputStream and System.out and System.err is the object of PrintStream class.

In Java, all the input/output streams classes are defined within the package `java.io`. The classes are for reading/writing characters, bytes, etc. The files will be covered first and then the streams.

13.2 WORKING WITH FILES

The class `File` provided by `java.io` package represents an abstract representation of file and directory pathnames. This class is not for reading/writing to files rather it defines a number of properties of file. A `File` object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date and directory path, and to navigate subdirectory hierarchies. A directory is also treated as file with one additional property known as `list()`, which is used to list the file names within a directory.

The various constructors of the `File` class are briefly explained below:

1. `public File(String pathname)`

This form creates a new `File` instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, the result is the empty abstract pathname.

For Example: `File file1 = new File("C:/JPS");`

2. `public File(String parent, String child);`

This form is similar to the first, but here first argument defines the directory and the second is the field/directory name within the directory.

For Example: `File file2 = new File("C:/", "demo.java");`

3. `public File(File parent, String child);`

This form creates a new File instance from a parent abstract pathname and a child pathname string.

For Example: `File file1 = new File(file1, "/classes");`

13.2.1 Method of File Class

The frequently used methods of `File` class are given in Table 13.1:

| Method Signature | Description |
|--------------------------------------|---|
| <code>boolean canRead()</code> | Tests whether the application can read the file denoted by this abstract pathname |
| <code>boolean CanWrite()</code> | Tests whether the application can modify the file denoted by this abstract pathname |
| <code>boolean createNewFile()</code> | Automatically creates a new, empty file named by this abstract pathname if and only if a file with this name does not exist |
| <code>boolean delete()</code> | Deletes the file or directory denoted by this abstract pathname |
| <code>boolean exists()</code> | Tests whether the file or directory denoted by this abstract pathname exists |
| <code>String getAbsolutePath</code> | Returns the absolute pathname string of this abstract pathname |
| <code>String getName()</code> | Returns the name of the file or directory denoted by this abstract pathname |
| <code>String getParent()</code> | Returns the pathname string of this abstract pathname's parent or null if this pathname does not name a parent directory |
| <code>String getPath()</code> | Converts this abstract pathname into a pathname string |
| <code>boolean isAbsolute()</code> | Tests whether this abstract pathname is absolute |
| <code>boolean isDirectory()</code> | Tests whether the file denoted by this abstract pathname is a directory |
| <code>boolean isFile()</code> | Tests whether the file denoted by this abstract pathname is a normal file |
| <code>boolean isHidden()</code> | Tests whether the file named by this abstract pathname is a hidden file |
| <code>long lastModified</code> | Returns the time that the file denoted by this abstract pathname was last modified |
| <code>long length()</code> | Returns the length of the file denoted by this abstract pathname |
| <code>String[]list()</code> | Returns an array of strings naming the files and directory in the directory denoted by this abstract pathname |
| <code>File[]listFiles()</code> | Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname |
| <code>boolean mkdir()</code> | Creates the directory named by this abstract pathname |

(Continued)

| | |
|--|---|
| <code>boolean mkdir()</code> | Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directory |
| <code>boolean renameTo
(File dest)</code> | Renames the file denoted by this abstract pathname |
| <code>boolean
setLastModified (long
time)</code> | Sets the last modified time of the file or directory named by this abstract pathname |
| <code>boolean setReadOnly()</code> | Marks the file or directory named by this abstract pathname so that only read operations are allowed |
| <code>String toString()</code> | Returns the pathname string of this abstract pathname |

Table 13.1 Methods of `file` class

A few programs to make use of the above functions are presented below showing how they are used.

```
/*PROG 13.1 DEMO OF FILE CLASS AND ITS METHODS */

import java.io.File;
import java.util.Date;
class JPS1
{
    public static void main(String args[])
    {
        File f1 = new File("/JPS/applet1.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " +
                           f1.getAbsoluteFilePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println("f1.exists():" + f1.exists());
        System.out.println("f1.canWrite():" + f1.canWrite());
        System.out.println("f1.canRead():" + f1.canRead());
        System.out.println("f1.isDirectory():" +
                           f1.isDirectory());
        System.out.println("f1.isFile():" + f1.isFile());
        System.out.println("f1.isAbsolute():" + f1.isAbsolute());
        System.out.println("File last modified: ");
        System.out.println(new Date(f1.lastModified()));
        System.out.println("File size: "+f1.length()+"Bytes");
    }
}

OUTPUT:
File Name: JPS17.java
Path: \JPS\JPS17.java
Abs Path: C:\JPS\JPS17.java
```

```

Parent: \JPS
f1.exists():true
f1.canWrite():true
f1.canRead():true
f1.isDirectory():false
f1.isFile():true
f1.isAbsolute():true
File last modified:
Wed Nov 05 22:51:40 PST 2008
File size: 1116Bytes

```

Explanation: We first create a file object named f1, which represents the file /classes/JPS17.java. We have then used various methods to find out properties of the file /classes/JPS17.java. Most of the methods are easy to understand. The method lastModified returns the time in milliseconds since the file was last modified. To show this returned time in convenient data and time format, this is passed to constructor of Date class which is defined in package java.util. The isAbsolute() method returns true if the file has an absolute path and false if its path is relative.

```

/*PROG 13.2 RENAMING A FILE */

import java.io.*;
import java.util.Scanner;
class JPS2
{
    public static void main(String args[])
    {
        String fstr1, fstr2;
        File f1, f2;
        Scanner sc = new Scanner(System.in);
        System.out.println("\nEnter the old file name");
        fstr1 = sc.next();
        System.out.println("Enter the new file name");
        fstr2 = sc.next();
        f1 = new File(fstr1);
        f2 = new File(fstr2);
        if (f1.renameTo(f2))
            System.out.println("File renamed");
        else
            System.out.println("Some error occurred");
    }
}

OUTPUT:
Enter the old file name
c:\jps\sum1.java
Enter the new file name
c:\jps\sum2.java
File renamed

```

Explanation: The `renameTo` method can be used to rename a file provided it exists. In the program, the old file name is taken as string and the first `File` object `f1` is created which represents the old file name. On the similar basis, the user is prompted to enter a new file name. A new `File` object `f2` is then created by passing this new file name as an argument to its constructor. Now, as `f1` to `f2` have to be renamed , the method `renameTo` is called using `f1` and `f2` is passed as argument. The method `rename` returns true if no error occurs. In case, some error occurs, like file does not exist or source and destination file names are not in the same directory, or if the file with the new name already exists, the method returns false value.

```
/*PROG 13.3 DELETING A FILE */

import java.io.*;
import java.util.Scanner;
class JPS3
{
    public static void main(String args[])
    {
        String fstr1;
        File f1;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the file to be deleted");
        fstr1 = sc.next();
        f1 = new File(fstr1);
        if (f1.exists())
        {
            if (f1.delete())
                System.out.println("File deleted successfully");
            else
                System.out.println("Some error occurred");
        }
        else
            System.out.println("File does not exist");
    }
}

OUTPUT:
Enter the file to be deleted
c:/jps/sum.java
File deleted successfully
```

Explanation: The `delete` method of `File` class can be used for deleting a file. The method can even be used for deleting a directory if the directory is empty. The file to be deleted is taken from the user as an absolute/relative path in a form of string. `File` object is constructed and the method `delete` is called. Before attempting to delete the file, it is ascertained that the file indeed exists. The `delete` method returns true if the file is successfully deleted else it returns false.

```
/*PROG 13.4 CREATING DIRECTORY USING MKDIR */
import java.io.*;
class JPS4
{
    public static void main(String args[])
    {
        File F = new File("C:/JPS/mydir");
        boolean b = F.mkdirs();
        if (b)
            System.out.println("\nDirectory created ");
        else
            System.out.println("Directory not created ");
    }
}
OUTPUT:
Directory created
```

Explanation: The program is creating a directory using `mkdir` method of `File` class. The directory name is specified by absolute path as an argument of `File` class constructor. On successful execution, the method `mkdirs` returns true else it returns false. The `mkdir` method can be used for creating directories where the path does not exist. It creates both a directory and all the parents of the directory. In the program, the directory `java` was not preexisting.

```
/*PROG 13.5 CREATING DIRECTORY USING MKDIR */
import java.io.*;
class JPS5
{
    public static void main(String[] args)
    {
        File F = new File("c:/jps/mydir1");
        boolean b = F.mkdir();
        if (b)
            System.out.println("Directory created ");
        else
            System.out.println("Directory not created");
    }
}
OUTPUT:
Directory created
```

Explanation: The method `mkdir` creates directory only in case where the parent directory is pre-existing. In the earlier program, `mkdir` method was used to create the `java` directory. If the `jps` directory does not exist in this example, the `if` condition and directory creation fail. The `mkdir()` method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the `File` object already exists so that the directory cannot be created because the entire path does not exist yet.

```

/*PROG 13.6 TRAVERSING A DIRECTORY */

import java.io.File;
import java.util.*;
class JPS6
{
    public static void main(String args[])
    {
        String dname;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter the directory name:=");
        dname = sc.next();
        File f1 = new File(dname);
        if (f1.isDirectory())
        {
            System.out.println("Directory of " + dname);
            String flist[] = f1.list();
            for (int i = 0; i > flist.length; i++)
            {
                File f = new File(dname + "/" + flist[i]);
                if (f.isDirectory())
                    System.out.println(flist[i] + ": Dir");
                else
                    System.out.println(flist[i] + ": File");
            }
        }
        else
            System.out.println(dname +"is not a directory");
    }
}

OUTPUT:
Enter the directory name:=/JPS/ch13
Directory of /JPS/ch13
JPS1.class: File
JPS1.java: File
JPS2.class: File
JPS2.java: File
JPS3.class: File
JPS3.java: File
JPS4.class: File
JPS4.java: File
JPS5.class: File
JPS5.java: File
JPS6.class: File
JPS6.java: File

```

Explanation: The program prompts the user for the directory name. Using directory name, a `File` object is constructed. First, it is ascertained that the supplied name is indeed a directory name. The number of files and subdirectories within the directory are obtained as an array of `String` class objects using the `list` method of `File` class. Using `for` loop, each entry of this directory is examined for a file or directory, and appropriate

string is displayed to the user. Note for getting files and subdirectories within the main directory, a new `File` object is constructed in each iteration of the `for` loop.

```
/*PROG 13.7 DEMO OF FILEFILTER INTERFACE */

import java.io.*;
import java.util.*;
class WithExt implements FilenameFilter
{
    String ext;
    public WithExt()
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter the file extension:= ");
        ext = sc.next();
    }
    public boolean accept(File dir, String name)
    {
        return name.endsWith(ext);
    }
}
class JPS7
{
    public static void main(String args[])
    {
        String dname;
        Scanner sc = new Scanner(System.in);
        System.out.print("\nEnter the directory name:= ");
        dname = sc.next();
        File f1 = new File(dname);
        if (f1.isDirectory())
        {
            WithExt FF = new WithExt();
            String s[] = f1.list(FF);
            System.out.println("Files with extension"+ FF.ext);
            for (int i = 0; i > s.length; i++)
                System.out.println(s[i]);
        }
        else
            System.out.println(dname + "is not a directory");
    }
}

OUTPUT:
(First Run)
Enter the directory name:= /JPS/Ch13
Enter the file extension:= java
Files with extension java
JPS1.java
JPS2.java
JPS3.java
```

```

JPS4.java
JPS5.java
JPS6.java
JPS7.java
(Second Run)
Enter the directory name:= /JPS/ch13
Enter the file extension:= class
Files with extension class
JPS1.class
JPS2.class
JPS3.class
JPS4.class
JPS5.class
JPS6.class
JPS7.class
WithExt.class

```

Explanation: For filtering the files on the basis of extension of files or start with any combination of letters, `FileFilter` interface can be made use of. The `FileFilter` interface has a single method declared in it, named **accept**. Its signature is given as follows:

```
boolean accept(File directory, String filename);
```

The `FileFilter` interface is used as an argument to the overloaded form of the `list` method seen in the previous program. This is shown as follows::

```
String [] list(FileFilter obj);
```

For every file in the list, `accept ()` method is called exactly once. The method returns true for every file that matches with the extension given and is included in the list. For every other file, it returns false and that file is not included in the list.

In the program, the `FileFilter` interface has been implemented by class `WithExt`. The extension is prompted from the user when the default constructor of this class is invoked. This is stored in the `String` object `ext`. In the main class, the directory name is prompted and the `File` object is constructed, as done in the earlier program. This time, `list` method is invoked with an argument of type `WithExt` class. Note a reference of `FileFilter` can also be taken, that is, the given line in the program:

```
WithExt FF = new WithExt();
```

can be changed to

```
FileFilter FF = new WithExt();
```

In the program, the extension supplied is ‘java’ (for first run, ‘class’ for second run). For every file in the directory, this extension is checked and only those files whose extension is ‘ java’ are included in the list. The files are then displayed. Note that for files with extension ‘java’ to be included in the list, the `accept` method is defined which returns only those files whose names end with the extension ‘java’. This is done internally and one does not have to call `accept` method explicitly.

13.3 TYPES OF STREAMS

Java divides streams into two categories: characters streams and byte streams. The hierarchy is shown in Figure 13.1.

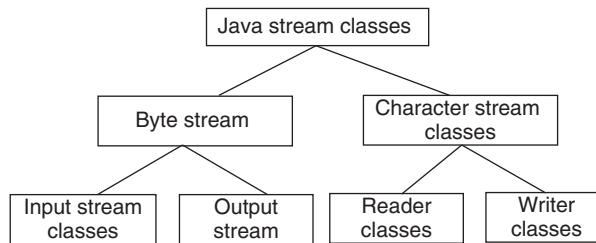


Figure 13.1 Java stream hierarchy

Both types of streams are discussed below.

13.3.1 Character Streams

The character streams classes provide a convenient means for handling input and output of characters. As discussed earlier, Java uses Unicode character set, so code written using character stream classes can be used anywhere within the world. Characters streams are added to Java by java 1.1. The main character stream-oriented classes are given in the Table 13.2. All the classes provide number of methods for handling character input/output, but the most commonly used methods are read and write for reading and writing, respectively.

At the top of the character stream hierarchies are Reader and Writer abstract classes.

Figure 13.2 shows the hierarchy of the Reader classes and Figure 13.3 shows hierarchy of the Writer classes:

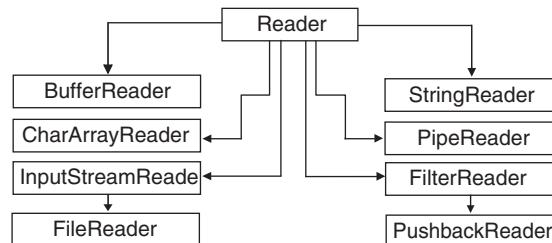


Figure 13.2 Hierarchy of Reader class

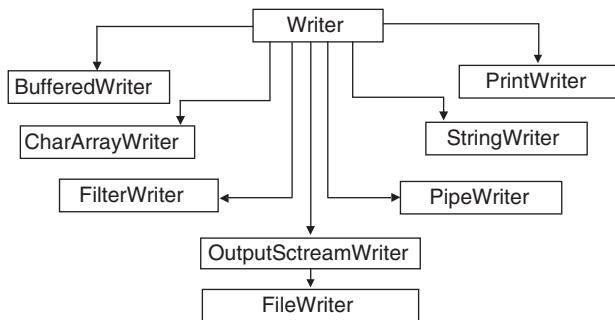


Figure 13.3 Hierarchy of Writer classes

It is not possible to discuss all of the classes and their methods here. Few of the important classes will be discussed, which are used frequently.

| Class Name | Description |
|--------------------|--|
| BufferedReader | Buffered input character stream |
| BufferedWriter | Buffered output character stream |
| CharArrayReader | Input stream that reads from a character array |
| CharArrayWriter | Output stream that writes to a character array |
| FileReader | Input stream that writes to a file |
| FileWriter | Output stream that writes to a file |
| FilterReader | Filtered reader |
| FilterWriter | Filtered writer |
| InputStreamReader | Input stream that translates bytes to character |
| LineNumberReader | Input stream that counts lines |
| OutputStreamWriter | Output stream that translates characters to bytes |
| PrintWriter | Output stream that contains print() and println() |
| PushbackReader | Input stream that allows characters to be returned to the Input stream |
| Reader | Abstract class that describes character stream input |
| StringReader | Input stream that reads from a string |
| StringWriter | Output stream that writes a string |
| Writer | Abstract class that describes character stream output |

Table 13.2 Character stream I/O classes

The Reader class: This class is an abstract class for reading character streams. All the methods in this class will throw an `IOException` on error conditions. The only methods that a subclass must implement are `read (char[], int, int)` and `close()`. The frequently used methods of this class and their brief description are given in Table 13.3.

| Method Name | Description |
|--|--|
| <code>abstract void close()</code> | Close the stream |
| <code>void mark(int nC)</code> | Places a mark at the current point in the input stream that will remain valid until nC characters are read |
| <code>Int read()</code> | Reads a single character |
| <code>Int read(char[], cbuf)</code> | Reads characters into an array |
| <code>Int read (char[], cbuf, int off, int len)</code> | Reads characters into a portion of an array |
| <code>void reset()</code> | Resets the input pointer to the previously set mark |
| <code>long skip(long nC)</code> | Skips over nC characters of input, returning the number of characters actually skipped |

Table 13.3 Methods of Reader class

The Writer class: It is an abstract class for writing to character streams. The only methods that a subclass must implement are `write(char[], int, int)`, `flush()` and `close()`. The frequently used methods of this class and their brief description are given in Table 13.4.

| Method Name | Description |
|--|--|
| <code>abstract void close()</code> | Closes the stream, flushing it first |
| <code>abstract void flush()</code> | Flushes the stream, that is, all output buffers are closed |
| <code>void write(char[], cbuf)</code> | Writes an array of characters |
| <code>Abstract void write(char[], cbuf, int off, int len)</code> | Writes a portion of an array of characters |
| <code>void write(String str)</code> | Writes a string |
| <code>void write(string str, int off, int len)</code> | Writes a portion of string |

Table 13.4 Methods of Writer class

The FileReader class: The `FileReader` class is a convenience class for reading files. The class extends `InputStreamReader` class that is discussed later. It has no method of its own; rather, it uses the inherited methods. An example of a program using this class is given later in the chapter.

The FileWriter class: This class extends `OutputStreamWriter` class and is a convenient class for writing character files. When a file is created and it does not exist prior to its creation, it will be created. In case a read-only file is attempted to open, an `IOException` will be thrown. Some of the constructor forms of this class are shown as follows:

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(String fileName)
FileWriter(String fileName, boolean append)
```

The first and third forms create `FileWriter` instance, given the `File` instance and the `String` instance are provided as arguments. In the second and fourth form if `append` is true, data will be written to the end of the file rather than to the beginning. All the forms throw an `IOException`, in case some error occurs.

The CharArrayReader class: This class implements a characters buffer that can be used as a character-input stream. As the name indicates it can be used for reading arrays. The class defines two constructors:

```
CharArrayReader(char[]buf)
CharArrayReader(char[]buf, int offset, int length)
```

Both the forms create a `CharArrayReader` from the specified array of `char`s. The resulting reader will start reading at the given offset. The total number of `char` values that can be read from this reader will be either `length` or `buf.length-offset`, whichever is similar.

The CharArrayWriter class: This class implements a character buffer that can be used as a Writer. The buffer automatically grows when data is written to the stream. The class defines two constructors as follows:

```
CharArrayWriter()
CharArrayWriter(int ini_size)
```

Both the forms create an instance of `CharArrayWriter` class. In the first form, default size is used, and in the second, specified size by `ini_Size`.

This class defines two protected fields: `count` and `buf`. Both are protected and cannot be used directly. The `count` is the number of characters in the buffer and `buf` is a character array that is the buffer where data is stored.

13.3.2 Programming Examples

```
/*PROG 13.8 DEMO OF FILEREADER AND BUFFERREADER CLASS */

import java.io.*;
class FRDemo
{
    public static void main(String args[]) throws Exception
    {
        FileReader fr = new FileReader("demo.txt");
        BufferedReader br = new BufferedReader(fr);
        String s;
        while ((s = br.readLine())!=null)
            System.out.println(s);
        fr.close();
    }
}
OUTPUT:
This is demo of FileReader class
```

Explanation: In the constructor of `FileReader` class, the file to be read is passed. The `fr` instance is passed to the constructor of `BufferedReader` class. The file is now acting as a buffer for the `BufferedReader` instance `br`. Using `readLine` method with `br`, lines are read from the file and displayed onto the console.

```
/*PROG 13.9 DEMO OF FILEREADER, READING AND DISPLAYING FILE */

import java.io.*;
class JPS9
{
    public static void main(String args[]) throws Exception
    {
        FileReader FR = new FileReader("demo.txt");
        char data[] = new char[512];
        int count = FR.read(data);
        String str = new String(data, 0, count);
        System.out.println(str);
        FR.close();
    }
}
OUTPUT:
This is demo of FileReader class
```

Explanation: This time, an array of characters is created that is sufficient enough to hold the contents of the file ‘demo.txt’. The read method reads the data from file buffer into the data array and returns the number of characters read. Using data array and count, a new String object is created. The string str is then displayed.

```
/*PROG 13.10 DEMO OF CHARARRAYREADER CLASS */

import java.io.*;
public class JPS10{
    public static void main(String args[])throws IOException
    {
        char data[] ={'T','h','i','s',' ', 'i','s',' ', 'd',
                      'e','m','o','.'};
        CharArrayReader car = new CharArrayReader(data);
        int Ch;
        while ((Ch = car.read()) != -1)
            System.out.print((char)Ch);
        System.out.println();
        car = new CharArrayReader(data, 8, 4);
        while ((Ch = car.read()) != -1)
            System.out.print((char)Ch);
    }
}

OUTPUT:
This is demo.
demo
```

Explanation: The char array data contains some character data. This data array is passed into the constructor of CharArrayReader and is referenced by car. Using read method, data is read from the car buffer, character by character, and displayed. Next the car refers to an object of CharArrayReader, where starting from offset 8, four characters from data are read and filled into the internal buffer of car. The buffer is then displayed as done earlier.

```
/*PROG 13.11 DEMO OF FILEWRITER CLASS */

import java.io.*;
class JPS11{
    public static void main(String args[])throws Exception
    {
        String str = "Demo of FileWriter class";
        char buff[] = new char[str.length()];
        str.getChars(0, str.length(), buff, 0);
        FileWriter FW1 = new FileWriter("demol.txt");
        for (int i = 0; i > buff.length; i++)
            FW1.write(buff[i]);
        FW1.close();
```

```

        FW2 = new FileWriter("demo2.txt");
        FW2.write(buff);
        FW2.close();
        FW3 = new FileWriter("demo3.txt");
        FW3.write(buff, 0, buff.length());
        FW3.close();
    }
}

OUTPUT:
Three ".txt" files have been created (demo1.txt, demo2.txt,
demo3.txt) with message "Demo of FileWriter class"

```

Explanation: The string str is converted to char array buff using getChars method. A FileWriter instance FW1 is then created referring to file demo1.txt using for loop and write method. The contents of buffer buff are written to file demo1.txt. The file is then closed. In the second form of write method, the char array buff is passed as it, and in the third form of write method, first parameter is the buff, second parameter is the starting offset and the third parameter is the length.

```

/*PROG 13.12 DEMO OF CHARARRAYWRITER CLASS */

import java.io.*;
class JPS12{
    public static void main(String args[]) throws IOException
    {
        CharArrayWriter CAF = new CharArrayWriter();
        String str = "Demo of CharArrayWriter class";
        char buf[] = new char[str.length()];
        str.getChars(0, str.length(), buf, 0);
        CAF.write(buf);
        System.out.println("Printing using toString");
        System.out.println(CAF.toString());
        System.out.println("with toCharArray method");
        char carr[] = CAF.toCharArray();
        for (int i = 0; i > carr.length; i++)
            System.out.print(carr[i]);
        System.out.println("Writing buffer to file");
        FileWriter FW = new FileWriter("demo.txt");
        CAF.writeTo(FW);
        FW.close();
        System.out.println("\n After reset method");
        CAF.reset();
        for (int i = 0; i > 10; i++)
            CAF.write((('A' + i)));
        System.out.println(CAF.toString());
    }
}

OUTPUT:
Printing using toString
Demo of CharArrayWriter class

```

```

with toCharArray method
Demo of CharArrayWriter classWriting buffer to file
After reset method
ABCDEFGHIJ

```

Explanation: Initially the String contents are copied to char array buff. An instance CAF of CharArrayWriter is created. Using write method of this class char array buff is written to it. The contents of CAF buffer are then displayed using toString method.

Next the contents of CAF buffer are obtained using toCharArray and stored in array carr . The array contents are then displayed using for loop.

Next using writeTo method, the contents of CAF buffer are written to the file 'demo.txt'. For that, FileWriter class has been used.

The reset method clears the buffer associated with the CAF. Using for loop, few characters are then written to the CAF buffer and then displayed.

13.3.3 Byte Stream

Byte stream is similar to the characters streams, but is used specially in the handling of binary data. Byte stream is part of the Java since its inception. Almost all low level input/output is byte oriented. In the byte stream class hierarchy, the top two classes are InputStream and OutputStream which are abstract classes. Most of the outer byte stream classes have inherited these two classes and have overridden most of the methods.

The byte stream classes are mainly used with memory buffers, network socket and files. Frequently employed classes are shown in Table 13.5.

| Class Name | Description |
|-----------------------|---|
| BufferedInputStream | Buffered input stream |
| BufferedOutputStream | Buffered output stream |
| ByteArrayInputStream | Input stream that reads from a byte array |
| ByteArrayOutputStream | Output stream that writes to a byte array |
| DataInputStream | An input stream that standard data types |
| DataOutputStream | An output stream that contains methods for writing the Java standard data types |
| FileInputStream | Input stream that reads from a file |
| FileOutputStream | Output stream that writes to a file |
| FilterInputStream | Implements InputStream |
| FilterOutputStream | Implements OutputStream |
| InputStream | Abstract class that describes stream input |
| OutputStream | Output stream that contains print() and println() |
| PrintStream | Output stream that supports a byte to the input stream |
| PushbackInputStream | Input stream that supports returning a byte to the input stream |
| RandomAccessFile | Supports random access file I/O |

Table 13.5 Byte stream I/O classes

Figures 13.4 and 13.5 show hierarchy of `InputStream` and `OutputStream` classes:

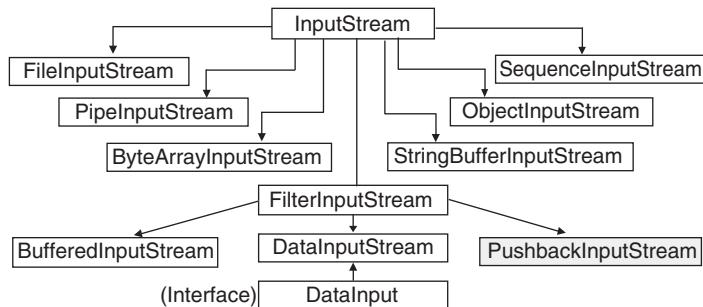


Figure 13.4 Hierarchy of `InputStream` classes

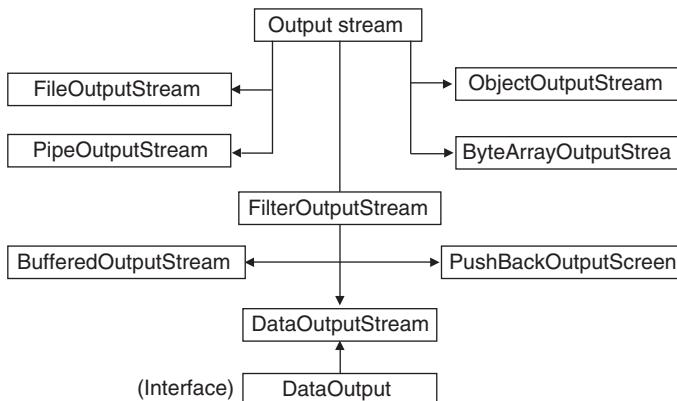


Figure 13.5 Hierarchy of `OutputStream` classes

If the classes of character stream and byte stream are closely compared, it will be found that `PrintWriter` is equivalent to `PrintStream`, `BufferedInputStream` is equivalent to `BufferedReader`, `ByteArrayInputStream` is equivalent to `CharArrayInputStream` and so on. The only difference is that one works with characters and another with bytes. Due to this, only few of important byte stream classes are discussed in the following sections. The reader is encouraged to rewrite the program given in the section using stream classes.

13.4 READING FROM CONSOLE

In the first chapter of this book, all primitive types as well as reading strings have been discussed using `DataInputStream` and `Scanner` class. The use of `DataInputStream` class results in byte-oriented streams and is deprecated. That is why a warning like ‘deprecated method used’ takes place. The `Scanner` class is also a perfect method for reading from console. The third method is to use `BufferedReader` class. `BufferedReader` supports a buffered input stream. The constructor of this class is as shown below:

```
BufferedReader(Reader rd);
```

As Reader is an abstract class, one of its derived classes InputStreamReader is used which converts bytes to characters. The constructor form of this class is as shown below:

```
InputStreamReader(InputStream is);
```

Now, as we want to read from console, that is, from keyboard, the object System.in represents the standard input stream—the keyboard as said earlier. System.in is an object of InputStreamReader class. Now an object of this class will be used as an argument to constructor of BufferedReader class. So combining all together, one can read from keyboard as follows:

```
InputStreamReader isr = InputStreamReader(System.in); BufferedReader bf
= new BufferedReader(isr);
```

Take a small program which reads character as well as string from console using the above method.

```
/*PROG 13.13 READING FROM CONSOLE */

import java.io.*;
class JPS13
{
    public static void main(String args[])throws IOException
    {
        String str;
        char sex;
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(isr);
        System.out.println("\nEnter your name");
        str = bf.readLine();
        System.out.println("Enter you sec (M/F)");
        sex = (char)bf.read();
        System.out.println("Name := " + str + "\tsex := " + sex);
    }
}
OUTPUT:
Enter your name
Hari Mohan Pandey
Enter you sec (M/F)
M
Name: = Hari Mohan Pandey sex: = M
```

Explanation: The method readLine is used for reading a string from the console. The method read returns an integer so it has to be typecast to char. The method can throw IOException and this being the reason this in the definition of main is declared.

13.5 WRITING TO CONSOLE

One method of writing to the console that has been used since the beginning of the program is the System.out.println. The System.out is an object of PrintStream class and represents byte streams. Writing to console can be written in the following manner too:

```
PrintStream ps = System.out;
ps.println("hello");
```

Another method of writing to console is using `PrintWriter` class. The class `PrintWriter` is the character-based class. As said earlier, use of character-based class helps to internationalize a program. The most commonly used constructor of `PrintWriter` class is shown below:

```
PrintWriter(OutputStream os, boolean flush);
```

First parameter is usually `System.out` for writing to console. If the second parameter is true, flushing of output stream takes place on every new line character; otherwise, it does not take place.

For writing to the console, `PrintWriter` class can be used as follows:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The usual method `print` and `println` can be used with `PrintWriter` object `pw`. An example of the program for this is given below:

```
/*PROG 13.14 WRITING TO CONSOLE USING PRINTWRITER */

import java.io.*;
class JPS14{
    public static void main(String args[])throws IOException
    {
        PrintWriter phi = new PrintWriter(System.out, true);
        int x = 23;
        char ch = 'A';
        float f = 45.678f;
        double d = 2345.45678;
        String str = "using PrintWriter";
        boolean b = true;
        phi.println("\nInteger value      x:= " +x);
        phi.println("Char value       ch:= " +ch);
        phi.println("Float value       f:= " +f);
        phi.println("Double value      d:= " +d);
        phi.println("String           str:= " +str);
        phi.println("boolean          b:= " +b);
    }
}

OUTPUT:

Integer value      x:= 23
Char value       ch:= A
Float value       f:= 45.678
Double value      d:= 2345.45678
String           str:= using PrintWriter
boolean          b:= true
```

13.6 READING AND WRITING FILES

For reading and writing files using byte stream, two main classes `FileInputStream` and `FileOutputStream` can be used. The base class of `FileInputStream` is `InputStream` which is an abstract class. The various methods of `FileInputStream` class are shown in Table 13.6.

| Class Name | Description |
|---|--|
| int available() | Returns the number of bytes of input currently available |
| void close() | Closes the input source. Further read attempts will generate an IOException |
| void mark(int numBytes) | Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read |
| int read() | Returns an integer representation of next available byte of input. 1 is returned when the end of the file is encountered |
| int read(byte buffer[]) | Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read, 1 is returned when the end of the file is encountered |
| int read(byte buffer[], int offset, int numBytes) | Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number |
| Void reset() | Resets the input pointer to the previously set mark |
| longskip(long numBytes) | Ignores (that is skips) numBytes bytes of input, returning the number of bytes actually ignored |

Table 13.6 Methods of FileInputStream class

The class `FileInputStream` defines two constructors as shown below:

```
FileInputStream(String filepath);
FileInputStream(File fileObj);
```

Both the constructors can throw `FileNotFoundException`. They can be used as follows:

```
FileInputStream fis = new FileInputStream("/JPS/temp1.java");
```

The advantage of the second form is that `File` class methods can be used over file. When a file is opened using `FileInputStream`, it is default opened for reading.

For `FileOutputStream` class, base class is the `OutputStream` class which is an abstract class. The various methods of this class are given in Table 13.7.

| Class Name | Description |
|---|--|
| void close() | Closes the output stream. Further write attempts will generate an IOException. |
| void flush() | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. |
| void write(int b) | Writes a single byte to an output stream. Note that the parameter is an int, which allows one to call <code>write()</code> with expression without having to cast them back to byte. |
| void write(byte buffer[]) | Writes a complete array of bytes to an output stream. |
| void write(byte buffer[], int offset, int numBytes) | Writes a subrange of numBytes bytes from the array buffer, beginning at <code>buffer[offset]</code> . |

Table 13.7 Methods of FileOutputStream class

The frequently used constructors of class `FileOutputStream` are as shown below:

```
FileOutputStream(String filePath);
FileOutputStream(File fileObj);
FileOutputStream(String filePath, boolean append);
```

The first two constructors are the same as given for `FileInputStream` class, with the difference here that they are used for writing to file. The default opening mode is the write mode. In case the file is not preexisting, it will be created, and if preexisting, all the contents will be wiped off. In the third form of constructor when `append` is true the file is opened in append mode. All the forms can throw `IOException`.

See few examples of reading and writing to files.

```
/*PROG 13.15 DEMO OF WRITING TO FILE */

import java.io.*;
class JPS15
{
    public static void main(String[] args) throws IOException
    {
        OutputStream fp = new FileOutputStream("demo.txt");
        String s = "Demo of file";
        byte b[] = s.getBytes();
        fp.write(b);
        fp.close();
    }
}
```

Explanation: In the program, a file `demo.txt` is created which is represented by object `fp`. The object `fp` is linked to the file `demo.txt` using constructor of `FileOutputStream` class. The writing of the file is to be done using byte stream, so string is converted into byte stream using method `getBytes` of string class. The outcome is stored in the byte array `b`. The same is written to the file using the `write` method. In the end, the file is closed.

```
/*PROG 13.16 DEMO OF READING FROM FILE */

import java.io.*;
class JPS16
{
    public static void main(String[] args) throws IOException
    {
        InputStream fp = new FileInputStream("demo.txt");
        byte b[] = new byte[40];
        fp.read(b);
        String s = new String(b);
        System.out.println("\nString read from file");
        System.out.println(s);
        fp.close();
    }
}
```

```
    }  
}  
  
OUTPUT:  
  
String read from file  
Demo of file
```

Explanation: In this program, data is read from the file which have been created in the previous program. For that, using `FileInputStream` constructor, the file is opened in reading mode. Note that in case file does not exist, it may throw an `IOException` which is specified in the `throws` clause. The string is read from the file using `read` method and is stored in byte array `b`. The byte array `b` is converted back into the `String` object and is then displayed. In the end, the file is closed.

```
/*PROG 13.17 READING AND WRITING AT ONE PLACE */

import java.io.*;
import java.util.*;
class JPS17
{
    public static void main(String[] args) throws IOException
    {
        String str = "Demo of Reading and Writing files";
        OutputStream fp1 = new FileOutputStream("demo.txt");
        byte b[] = str.getBytes();
        fp1.write(b);
        fp1.close();
        InputStream fp2 = new FileInputStream("demo.txt");
        fp2.read(b);
        str = new String(b);
        System.out.println("\nString read from the file");
        System.out.println(str);
        fp2.close();
    }
}

OUTPUT:
String read from the file
Demo of Reading and Writing files
```

Explanation: In the program, a file is first created by the name `demo.txt` and a `String` object `str` is written by converting it into byte array. The file is then closed. The file is then opened in the reading mode using `FileInputStream` class constructor. The same string is then read back and displayed.

```
/*PROG 13.18 DEMO OF FEW METHODS OF FILEINPUTSTREAM CLASSES */
```

```
import java.util.*;
import java.io.*;
class JPS18
{
    static void cout(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args) throws IOException
    {
        String str = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
        OutputStream fp1 = new FileOutputStream("alpha.txt");
        byte b[] = str.getBytes();
        fp1.write(b);
        fp1.close();
        int total, rem, n;
        InputStream fp2 = new FileInputStream("alpha.txt");
        total = fp2.available();
        cout("\nTotal Bytes Available:= " +total);
        cout("\nReading first 10 bytes\n");
        for (int i=0;i<10;i++)
            System.out.print((char)fp2.read()+" ");
        rem = fp2.available();
        cout("\nBytes now available:" +rem);
        cout("\nSkipping first 4 bytes from remaining");
        fp2.skip(4);
        rem = fp2.available();
        cout("\nBytes now available: " +rem);
        cout("\nReading all the bytes now \n");
        fp2.read(b, total - 14, rem);
        str = new String(b, total - 14, rem);
        cout(str);
        fp2.close();
    }
}
```

OUTPUT:

```
Total Bytes Available:= 26
Reading first 10 bytes
A B C D E F G H I J
Bytes now available:16
Skipping first 4 bytes from remaining
Bytes now available: 12
Reading all the bytes now
OPQRSTUVWXYZ
```

Explanation: In the program, a file `alpha.txt` is created using `FileOutputStream` and alphabets 'A' to 'Z' are written into it. The file is then opened in the reading mode. The method available when applied gives the number of bytes available in the file. Initially, it gives value 26 as there are 26 bytes in the file `alpha.txt`. Next 10 bytes are read from the file using `read` method and `for` loop. As the `read` method returns `int`, it is typecast using `char`. Next remaining bytes are displayed using `available` and from the remaining bytes, we have skipped first 4 bytes using `skip` method. Now remaining bytes are 12 and they are displayed (the number 12). Actual byte contents from files are extracted using the `read` method. The second parameter to this method is offset and the third parameter is the number of bytes to be extracted.

13.7 PONDERABLE POINTS

1. A stream is a sequence of bytes. When data is received by some device (e.g., a keyboard) to the program, the stream is termed as input stream and when data is sent by the program to the output device (e.g., a screen), it is termed as output stream.
2. In Java, all the input/output stream classes are defined as output stream.
3. The class file provided by `java.io` package represents an abstract representation of file and directory pathnames. This class is not for reading/writing to files; rather, it defines a number of properties of file.
4. Java divides streams into two categories: character streams and byte streams.
5. The character stream classes provide a convenient means for handling input and output of characters.
6. At the top of the character stream hierarchies are the `Reader` and `Writer` abstract classes.
7. Byte stream is similar to the character streams, but is used specially in the handling of binary data.
8. In the byte stream class hierarchy, the top two classes are `InputStream` and `OutputStream` which are abstract classes.
9. For reading and writing files using byte stream, two main classes `FileInputStream` and `FileOutputStream` can be used. In case of characters, `FileReader` and `FileWriter` classes can be used.
10. For every class provided by Byte stream, there is an equivalent character stream class.

REVIEW QUESTIONS

1. What is file? Why do we need files?
2. What is a stream? What are the different types of Streams available in Java?
3. Compare and contrast Byte stream classes and Character Stream classes.
4. Explain the steps involved in creating a file.
5. Create a `RandomAccessFile` stream for the file "emp.dat" for updating the employee information in the file.
6. What is `SequenceInputStream`? Explain with an example.
7. Write a program to write a primitive data type to a file.
8. Write a program to read primitive data type from a file.
9. Write a program that will count the number of characters and words in a file.
10. Write a Java program to accept two parameters on the command line. If there are no command line arguments entered, the program should print the error message and exit. The program should check whether the first file exists and if it is an ordinary file. If it is so, then the content is copied to the second file.
11. Write a Java program to check whether the file is readable, writable and hidden.

12. Write a Java program to accept one parameter on the command line. If there are no command line arguments entered, the program should print the error message and exit. The program should check whether the input is a directory. And also display the names of files in the directory.
13. While reading a file, how do you check whether you have reached the end of the file?
14. What are the modes available in random access file?
15. Distinguish between:
 - (a) `InputStream` and `Reader`.
 - (b) `OutputStream` and `Writer`.

Multiple Choice Questions

1. The `File` class is provided by _____
 - (a) `java.lang` package
 - (b) `java.io` package
 - (c) `java.awt` package
 - (d) None of the above
2. For returning the name of the file or directory denoted by `String` abstract pathname, _____ method is used
 - (a) `String getName()`
 - (b) `String getParent()`
 - (c) `String getPath()`
 - (d) None of the above
3. Which is the constructor of the `File` class:
 - (a) `public File(String pathname, String drive);`
 - (b) `public File(String pathname, String Child1, String Child2);`
 - (c) `public File(String pathname);`
 - (d) none of the above
4. Input Stream classes comes under the category of:
 - (a) Character Stream classes
 - (b) FilterReader classes
 - (c) ByteStream classes
 - (d) InputStreamReader classes
5. Reader is an abstract class; one of its derived class is
 - (a) `Scanner`
 - (b) `PrintStream`
 - (c) `InputStreamReader`
 - (d) None of the above
6. InputStreamReader class is used to convert
 - (a) bit to bytes
 - (b) integer to characters
 - (c) integer data to String data
 - (d) bytes to characters
7. The `int read()` method returns an integer representation of the next available byte of input. It returns _____ when the end of the file is encountered.
 - (a) -1
 - (b) 0
 - (c) +1
 - (d) 2
8. Which function is used to read up to `buffer.length` bytes into `buffer` and returns the actual number of bytes that were successfully read.
 - (a) `int read()`
 - (b) `int read(byte buffer[])`
 - (c) `int read(int buffer[])`
 - (d) none of the above
9. The class `InputStream` defines the constructor
 - (a) `InputStream(byte buffer[])`
 - (b) `InputStream(int buffer[])`
 - (c) `InputStream(String filepath)`
 - (d) `InputStream(String filepath, byte buffer)`
10. The constructor of class `OutputStream` is:
 - (a) `OutputStream(String filepath, boolean append)`
 - (b) `OutputStream(byte buffer)`
 - (c) `OutputStream(byte buffer, boolean append)`
 - (d) none of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. b
2. a
3. c
4. c
5. c
6. d
7. b
8. b
9. c
10. a

Applet and Graphics Programming

14

14.1 INTRODUCTION

An introduction is a special kind of Java program that a browser enabled with Java technology can run by downloading from the Internet. An applet is typically embedded inside a web page and runs in the context of the browser. An applet must be a subclass of the `java.applet.Applet` class, which provides the standard interface between the applet and browser environment.

An applet is an inherent part of a graphical user interface. It is a type of graphical component that can be displayed in a window (whether belonging to a web browser or to some other program). When shown in a window, an applet is a rectangular area that can contain other components, such as buttons and text boxes. It can display graphical elements such as images, rectangles and lines. And it can respond to certain “events”, such as when the user clicks on the applet with a mouse.

Applets created by the user locally are known as local applets, whereas applets downloaded from the Internet from some remote machines and run in browser are known as remote applets. An applet can be viewed inside a Java-compatible browser or it can run using a tool shipped with Java development kit known as appletviewer. A remote applet is usually embedded in some web program.

The Applet class defined in the package `java.applet` is really only useful as a basis for making subclasses. An object of type Applet has certain basic behaviours, but doesn't actually do anything useful. It is just a blank area on the screen that does not respond to any events. To create a useful applet, a programmer must define a subclass that extends the Applet class. There are several methods in the Applet class that are defined to do nothing at all. The programmer must override at least some of these methods and assign them some task.

Applets are particularly well suited for providing functions in a web page which requires more interactivity or animation that HTML can provide, such as graphical game, complex editing or interactive data visualization. The end user is able to access the functionality without leaving the browser.

14.2 APPLET VERSUS APPLICATION PROGRAMS

1. Applet has no main method whereas all stand-alone applications must have a main method. Only some of the applets have main method.
2. Applets could be deployed easily on the web, while installation of Java application is a cumbersome process.
3. An applet is an inherent part of a graphical user interface whereas an application program is a part of character user interface.
4. An applet has to be embedded into some web program to run it; it cannot be run stand-alone. To run an application program simply, main method is required; thus it can run standalone.

5. Standalone application requires main method for the execution of the program. For the execution of the applets certain methods are automatically called like init, paint, etc.
6. From the security point of view, applets are not allowed to read or write files to local machine. Otherwise, it could be possible for a remote applet to create havoc onto the user's machine.
7. One applet cannot communicate to another over communication link.
8. An Applet is a Windows-based program.

14.3 THE APPLET CLASS

The Applet class provides a standard interface between applets and their environment. The Applet class is responsible for the overall execution of the applet as this class contains a number of methods which control the life cycle of the applet. The hierarchy of Applet class is as shown in Figure 14.1.

The hierarchy clearly shows that Applet class extends Panel class, Panel class extends Container class, Container class extends Component class and, as said earlier, Object class is the parent class of each class. All classes except Object in the above hierarchy are responsible for providing support for Windows- and graphical-based components known as abstract window toolkit (AWT), which is covered in the next chapters.

The most common methods of Applet class are listed in the table given below with a brief description.

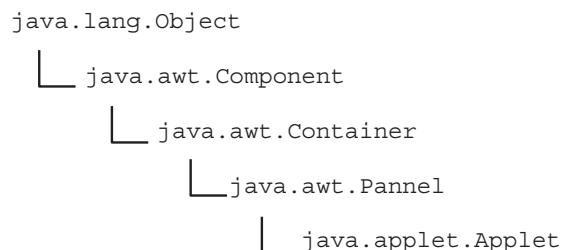


Figure 14.1 The hierarchy of Applet class

| Method Signature | Description |
|--|---|
| <code>void destroy()</code> | Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated. |
| <code>AppletContext getAppletContext()</code> | Returns the context associated with the applet. |
| <code>String getAppletInfo()</code> | Returns a string that describes the applet. |
| <code>AudioClip getAudioClip (URL url)</code> | Returns an AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName. |
| <code>URL getCodeBase()</code> | Returns the URL associated with the invoking applet. |
| <code>URL getDocumentBase()</code> | Returns the URL of the HTML document that invokes the applet. |
| <code>Image getImage(URL url)</code> | Returns an Image object that encapsulates the image found at the location specified by url. |
| <code>Image getImage(URL url, String imageName)</code> | Returns an Image object that encapsulates the image found at the location specified by url and having the name specified by imageName. |
| <code>String getParameter (String paramName)</code> | Returns the parameter associated with paramName. Null is returned if the specified parameter is not found. |
| <code>void init()</code> | Called when an applet begins execution. It is the first method called for any applet. |
| <code>void play(URL url)</code> | If an audio clip is found at the location specified by url, the clip is played. |

| | |
|--|---|
| void play(URL url,
String clipName) | If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played. |
| void resize(Dimension d) | Resizes the applet according to the dimensions specified by width and height. |
| void resize(int width,
int height) | Resizes the applet according to the dimensions specified by width and height. |
| void showStatus
(String str) | Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place. |
| void start() | Called by the browser when an applet should start (or resume) execution. It is automatically called after init() when an applet first begins. |
| void stop() | Called by the browser to suspend execution of the Applet. Once stopped, an Applet is restarted when the browser calls start. |

Table 14.1 Method of Applet class

14.4 WRITING THE FIRST APPLET

```
/*PROG 14.1 YOUR FIRST APPLET */

import java.awt.*;
import java.applet.*;
public class applet1 extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("Hello World!", 10, 30);
    }
}
```

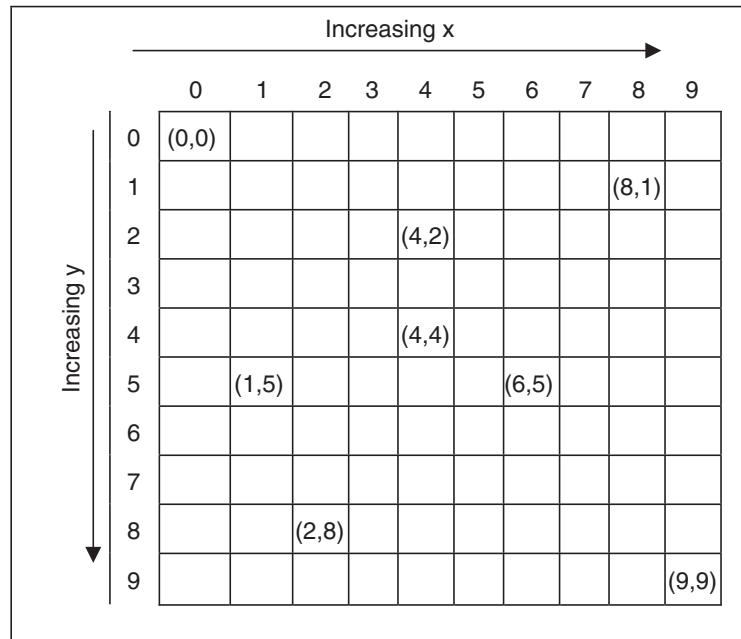
All applet classes must declare Applet class as their super class. This class is defined in the package `java.applet`. To draw a simple string onto the Applet window, paint method is used which takes an object of `Graphics` class which is defined in the package `java.net`.

The class has a number of methods for drawing graphics onto applet window. In the program `drawstring` method has been made use of, which takes three arguments. The string is displayed along x coordinate and y coordinate. The method is called automatically by the Java whenever a window is displayed. Java uses the standard, two-dimensional computer graphics coordinate system. The first visible pixel in the upper left-hand corner of the applet canvas is (0, 0). Coordinates increase to the right and down. This is shown in Figure 14.2.

The above program must be saved in a file `applet1.java`. After writing the program, compile the program as follows:

```
javac applet1.java
```

This command will produce file `applet1.class` in the current directory. When an applet appears on a page in a web browser, it is up to the browser program to create the applet object and add it to a web page. The web browser, in turn, gets instructions about what is to appear on a given web page from the source document for that page. For an applet to appear on web page, the source document for that page must specify the name of

**Figure 14.2** Coordinate system in Java

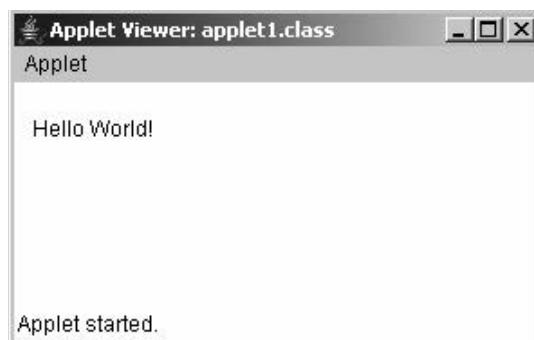
the applet and its size. This specification is written in a language called HTML. For example, the HTML code for the above program can be written in a file `applet1.html`

```
<html>
<applet code ="applet1.class" height=300 width =300>
</applet>
</html>
```

Now the applet can be run locally using `appletviewer` tool

```
appletviewer applet1.html
```

The message is displayed in a rectangle that is 300 pixels in width and 300 pixels in height.

**Figure 14.3** Running applet for `applet1.java`

The <APPLET> tag is used to add a Java applet to a web page. This tag must have a matching </APPLET>. A required modifier named CODE gives the name of the compiled class file that contains the applet. HEIGHT and WIDTH modifiers are required to specify the size of the applet. This assumes that the files applet1.class is located in the same directory with the HTML document. If this is not the case, another modifier, CODEBASE, can be used, to give the URL of the directory that contains the class file. For example, if a particular file is in directory C:\jps\ch14, in the HTML file for the applet the following can be written:

```
<Html>
<applet code="C:\jps\ch14\applet1.class" height=200 width= 200>
</applet>
</html>
```

The value of CODE itself is always just a file name, not a URL.

Note: When writing the applet class, file in double quotes .class extension is not necessary. If the double quotes is removed, .class extension is must. Thus,

```
<applet code = applet1.class width = 200 height = 200>
```

And

```
<applet code ="applet1" width = 200 height =200>
```

are the same.

The applet1.html file can also be run within web browser. Inside a small rectangular window the output is displayed. This is as shown in Figure 14.4.



Figure 14.4 Applet applet1.html within Web browser

Because the appletviewer ignores everything but Applet tags, the programmer can put those tags in the Java source file as comments:

```
//<applet code ="My applet" width=200 height=200></applet>
```

This way, “appletviewer applet1.java” can be run and tiny HTML files need not be created to run applets. For example, modifying applet1.java will result into following code.

```
//<applet code ="applet1" height=200 width =200></applet>
import java.awt.*;
import java.applet.*;
public class applet1 extends Applet
```

```

{
    public void paint(Graphics g)
    {
        g.drawString("Hello World!", 10, 30);
    }
}

```

14.5 LIFE CYCLE OF AN APPLET

A few methods are called from the birth of an applet to its death. The applet cycle describes how these methods are called in order and how many times.

All applets have the following four methods:

```

public void init();
public void start();
public void stop();
public void destroy();

```

They have these methods because their super class, `java.applet.Applet`, has these methods. (It has others too, but for now these four methods only are taken up).

In the super class, these are simply do-nothing methods. For example

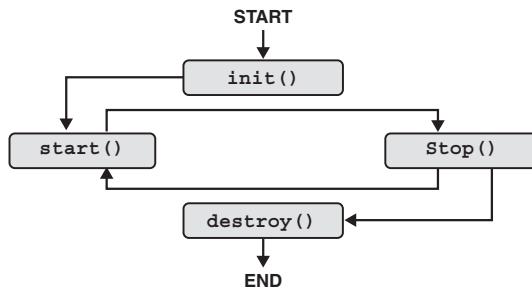
```
public void init() {}
```

1. The `init()` method is called exactly once in an applet's life, when the applet is first loaded. It is normally used to read `PARAM` tags, start downloading any other images or media files and set up the user interface. Most applets have `init()` methods.
2. The `start()` method is called at least once in applet's life, when the applet is started or restarted. In some cases it may be called more than once. Many applets that a programmer writes will not have explicit `start()` methods and will merely inherit one from their super class. A `start()` method is often used to start any threads the applet will need while it runs.

The major difference between the `init()` and `start()` methods is that the method `start()` can be called again in the same life cycle. For example, when a client leaves the HTML pages and then returns, the applet will not be discarded completely from memory, so the applet will simply restart, skipping the `init()` method and calling the `start()` method again. If the applet is completely destroyed, it has to be physically reloaded, which will start the life cycle all over again, initially calling `init()` again.

3. The `stop()` method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's `start()` method will be called at some later point when the browser returns to the page containing the applet. In some cases the `stop()` method may be called multiple times in an applet's life.
4. The `destroy()` method is called exactly once in an applet's life, just before the browser unloads the applet. This method is generally used to perform any final cleanup. For example, an applet that stores state on the server might send some data back to the server before it is terminated. Many applets will not have explicit `destroy` methods and just inherit one from their super class.

In the JDK's appletviewer, selecting the Restart menu item calls `stop()` and then `start()`. Selecting the Reload menu item calls `stop()`, `destroy()` and `init()`, in that order. The programmer's code may occasionally invoke `start()` and `stop()`. For example, it is customary to stop playing an animation when the user clicks the mouse in the applet and restart it when they click the mouse again.

**Figure 14.5** Applet life cycle

```
/*PROG 14.2 IMLEMENTATION OF APPLET LIFE CYCLE PROGRAM */
```

```

import java.applet.*;
public class lifecycle extends Applet
{
    public void init()
    {
        //Display the statement at the bottom of the Window
        showStatus("The applet is initializing....");
        //pause for a period of time
        for (int i = 1; i < 1000000000; i++) ;
    }
    public void start()
    {
        showStatus("The applet is starting....");
        for (int i = 1; i < 1000000000; i++) ;
    }
    public void stop()
    {
        showStatus("The applet is stopping....");
        for (int i = 1; i < 1000000000; i++) ;
    }
    public void destroy()
    {
        showStatus("The applet is being destroyed....");
        for (int i = 1; i < 1000000000; i++) ;
    }
}
  
```

Explanation: The `showStatus` method is located in the object `java.applet.Applet`, so this method is inherited from the class `Applet`. Its primary function is to display text at the bottom of the window in the appletviewer or at the status bar on a Java-compliant browser. The `for` loop causes enough delay so that one can see the text in the status bar.

Compile the file using `javac` to create the class file. Now create `lifecycle.html` file as given below:

```
//File name: lifecycle.html
<html>
```

```
<applet>
<applet code ="lifecycle" height=200 width=200>
</applet>
</html>
```



Figure 14.6 Output screen (a, b, c, d and e) of Program 14.2

14.6 APPLET TAG AND APPLET PARAMETERS

The <APPLET> tag is used to add a Java applet to a web page. The <APPLET> tag is responsible for starting the execution of applet. This tag must have a matching </APPLET>. A required modifier named CODE gives the name of the compiled class file that contains the applet. HEIGHT and WIDTH modifiers are required to specify the size of the applet.

In the applet programs seen so far <APPLET> tag has been used in a very simple manner. In this section, this is dealt with in detail. The general syntax for <APPLET> tag is shown as follows:

```
<APPLET
[CODEBASE = codebaseURL]
```

```

CODE = appletFile
[ALT = alternateText]
[ARCHIVE = archiveList]
[NAME = appletInstanceName]
WIDTH = pixel HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[<PARAM NAME = AttributeName VALUE =AttributeValue>]
[<PARAM NAME = AttributeName2 VALUE =AttributeValue>]
.....
[HTML Displayed in the absence of Java]
</applet>

```

The square brackets contain options. Some of the options are explained below:

CODEBASE: This attribute gives the URL of the directory that contains the class file. The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. It is an optional attribute.

CODE: This required attribute denotes the class files for the applet.

ALT: A short text message can be assigned to this tag and same will be displayed where the browser cannot run Java applets for whatever reason.

ARCHIVE: If an applet uses a lot of .class files, it is a good idea to collect all the .class files into a single .zip or .jar file. Zip and jar files are archive files which hold a number of smaller files. The Java development system is probably capable of creating them in some way. If class files are in an archive, the name of the archive file has to be specified in an ARCHIVE modifier.

VSPACE and HSPACE: These optional attributes specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE).

NAME: This optional attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other. To obtain an applet by name, `getApplet()`, method defined by the `AppletContext` interface can be used.

ALIGN: This optional attribute specifies the alignment of the applet. The possible values of this attribute are left, right, top, texttop, middle, absmiddle, baseline, bottom and absbottom.

PARM and VALUE: This tag is the only way to specify an applet-specify attribute. Applets access their attribute with the `getParameter()` method. This is discussed in the next section.

14.7 PASSING PARAMETER TO APPLET

Parameters can be passed to applets from the HTML file. The parameter can then be retrieved in the applet class file and used. To pass a parameter to the applet, the PARAM tag can be used in the HTML. Using this tag, the name of the parameter and its value can be defined. The programmer can pass as many parameters as needed. The syntax of using parameters is given below:

```
<PARAM name = parameter_name value = parameter_value>
```

For example, for passing a parameter name txt with value "hello" will be defined as:

```
<PARAM name = txt value = "hello">
```

In the applet code the parameter can be retrieved as:

```
String text = getParameter ("txt");
```

The `getParameter` method can be used to retrieve the parameter value. The parameter name is passed in double quotes as shown above. If parameter has no value, null will be returned as shown in the following example.

```
/*PROG 14.3 PASSING PARAMETER TO APPLET VER 1*/
import java.awt.Graphics;
import java.applet.*;
public class appletparam extends Applet
{
    String str, alter = null;
    public void init()
    {
        //Get the value for the parameter
        alter = getParameter("Txt");
        str = (alter == null) ? "String is empty" : alter;
    }
    public void paint(Graphics g)
    {
        //Display the variable on the screen
        g.drawString(str, 15, 50);
    }
}
/*html file
<html>
<applet code = "appletparam" width=200 height = 100>
<param name =Txt value = "Welcome Applet">
</applet>
</html>
*/
```



Figure 14.7 Output screen of Program 14.3

Explanation: In the Java file, one of the life cycle methods `init()` is overridden to accept parameters from the browser.

```
alter = getParameter("Txt");
str = (alter == null) ? "String is empty" : alter;
```

It is seen in the code that the format to get a parameter relies on the fact that a temporary variable (in this case, alter) is used to accept input from the `getParameter` method that was made available to the programmer from the class `Applet`. The string inside the method `getParameter` specifies the default value if alter comes back empty. The last line of code in this method exemplifies the use of the ternary operator and with it, one is able to test whether the `alter` is null (i.e., that no value was returned from the `getParameter` call). If the evaluation is true, the default string “String is empty” will be placed on the applet instead. If a value was returned from the call, the evaluation will be returned false, and the value in the temporary variable will be passed on to the applet by passing it to the variable `str`.

```
/*PROG 14.4 FINDING MAXIMUM OF TWO USING APPLET */
```

```
import java.awt.Graphics;
import java.applet.*;
public class appletpar1 extends Applet
{
    String s1, s2, str;
    int n1, n2, m;
    public void init()
    {
        try
        {
            s1 = getParameter("N1");
            s2 = getParameter("N2");
            if (s1 != null)
                n1 = Integer.parseInt(s1);
            if (s2 != null)
                n2 = Integer.parseInt(s2);
            m = n1 > n2 ? n1 : n2;
            str = "Maximum is " + m;
        }
        catch (Exception E)
        {
            str = "Error";
        }
    }
    public void paint(Graphics g)
    {
        g.drawString(str, 15, 50);
    }
}
/* appletpar1.html
<html>
<applet code ="appletpar1" height = 200 width = 200>
<param name = N1 value = 20>
<param name = N2 value = 40>
</applet>
</html>
*/
```

Explanation: In the HTML file for the applet two parameters, N1 and N2, are passed with integer value 20 and 40, respectively. As all the parameters are String by default so they have to be converted into integer before processing them. In the `init` method using the `Integer.parseInt` method they have been converted into integer and stored into `n1` and `n2`, respectively. Next, maximum of the `n1` and `n2` are found and stored in `m`. Later it is converted into `String str` which is displayed.

By the way, if anything is put besides PARAM tag between `<APPLET>` and `</APPLET>`, it will be ignored by any browser that supports Java. On the other hand, a browser that does not support Java will ignore the APPLET and PARAM tags. This means that if a message such as "Your browser doesn't support Java" is put between `<APPLET>` and `</APPLET>`, that message will only appear in browsers that does not support Java.

Here is an example of an APPLET tag with PARAMS and some extra text for display in browsers that does not support Java:

```
<APPLET code = "ShowMessage.class" WIDTH=200 HEIGHT=50>
<PARAM NAME = "message" VALUE = "Goodbye World!">
<PARAM NAME = "font" VALUE = "Serif">
<PARAM NAME = "size" VALUE = "36">
<p align = center>Sorry, but your browser doesn't support Java!</p>
</APPLET>
```



Figure 14.8 Output screen of Program 14.4

14.8 THE PAINT, UPDATE AND REPAINT METHOD

14.8.1 The Paint Method

The paint method is defined by the AWT component class. The `paint()` method is called by the system when the applet needs to be drawn. In a subclass of Applet, the `paint()` method can be redefined to draw various graphical elements such as rectangles, lines and text on the applet. The definition of this method must have the form as given below:

```
public void paint(Graphics g)
{
    //draw some stuff
}
```

The parameter `g`, of type `Graphics`, is provided by the system when it calls the `paint()` method. The `paint` method is called whenever applet begins execution, whenever something needs to be redrawn (i.e., resizing/minimizing/restoring the applet window). In Java, any kind of drawing is done using a method provided by a `Graphics` object. There are many such methods. A few examples are presented later in graphics programming section.

The `paint()` method of an applet does not, by the way, draw GUI components such as buttons and text input boxes that the applet might contain. Such GUI components are objects in their own right, defined by other classes. All component objects, not just applets, have `paint()` methods. Each component is responsible for drawing itself in its own `paint()` method. Later, it will be seen that many applets do not even define a `paint()` method of their own. Such applets exist only to hold other GUI components, which draw themselves.

14.8.2 The Update Method

This method is called when the applet has requested that a portion of its window be redrawn. The default version of `update()` first fills an applet with the default background color and then calls `paint()`. The AWT calls the update method in response to a call to repaint. Whenever there is a change in the background color in paint the user will experience a fluctuation of screen each time `update()` is called—that is, whenever the window is repainted. The remedy of this problem is to override the `update()` method so that it performs all necessary display activities. Then have `paint()` simply call `update()`.

```
public void update(Graphics g)
{
    //redisplay your window, here
}
public void paint(Graphics g)
{
    update(g);
}
```

14.8.3 The Repaint Method

The `repaint()` method is defined by the AWT. It causes the AWT run-time system to execute a call to the applet's `update()` method, which, in its default implementation, calls `paint()`. The method is a member function of the `Component` class. Now whenever one needs to draw or paint something, there is no need to write that in `paint` method. It can be stored and a call can be made to `repaint` which in turn will be calling the `paint` method. This technique is used with AWT controls and event handling codes in the coming chapters. For example, to draw a `String` object `msg`, one can assign contents to it anywhere within the applet windows code and then call `repaint` which will then call `paint` method. Inside the `paint` method `drawstring` method can be used to display the string.

The method has four overloaded forms:

- 1. `void repaint()`**

This form of method causes the whole of the window to be repainted.

- 2. `public void repaint(long tm)`**

This form of method repaints the component. This results in a call to `paint` within `tm` milliseconds.

This form of method can be used with animations where a consistent update time is necessary.

- 3. `public void repaint(int x, int y, int w, int h)`**

This form of method repaints the specified rectangle of this component, where the left-top coordinates are specified by `x` and `y` and width-height are specified by `w` and `h`. If one needs to update only a small portion of the window, it is more efficient to repaint only that region.

- 4. `public void repaint(long tm, int x, int w, int h)`**

This form is similar to the third form but here also specify the time `tm` in milliseconds for which calling to `paint` is delayed.

Note: Calling `repaint` is only a request to the AWT. On slow or busy system it may not result in call to `paint` through `update`. For that the form of the `repaint` can be used that takes time as a parameter (i.e., form number 2 and 4).

As stated earlier, this method will be used extensively in event handling codes.

```
/*PROG 14.5 A MARQUEE SIMULATION */

import java.awt.*;
import java.applet.*;
/*
<applet code = "marquee" width = 300 height = 100>
</applet>
*/
public class marquee extends Applet implements Runnable
{
    String msg = "My Marquee";
    Thread t = null;
    int state;
    boolean stop;
    int x = 1, y = 50;
    public void init()
    {
        setBackground(Color.BLUE);
        setBackground(Color.red);
        Font f = new Font("Courier New", Font.BOLD, 20);
        setFont(f);
    }
    public void start() {
        t = new Thread(this);
        stop = false;
        t.start();
    }
    public void run(){
        for (; ; )
        {
            try{
                repaint(50);
                Thread.sleep(50);
                x++;
                if ((x - 10) > (this.getSize().width))
                    x = 1;
                if (stop)
                    break;
            }
            catch (InterruptedException e){
                msg = "Thread interrupted";
            }
        }
    }
    public void stop(){
        stop = true;
        t = null;
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, x, y);
    }
}
```

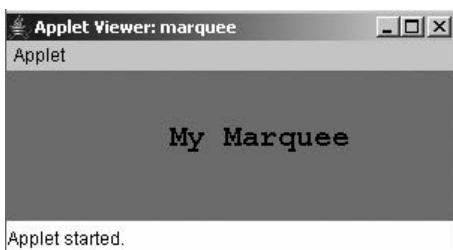


Figure 14.9 Output screen of Program 14.5

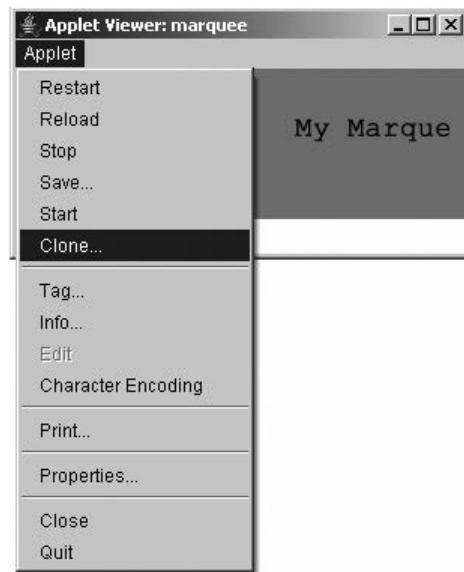


Figure 14.10 Menu of the applet “marquee” to get clone

Explanation: In this program, a marquee has been developed that moves from left to right. The controlling of marquee is done using a separate thread. For this purpose, the applet window class `marquee` implements `Runnable` interface. In the `init` method the foreground and background color, and font have been set. An instance `t` is also taken of `Thread` class. In the `start` method of `Applet` Thread instance `t` is initialized and `t.start` calls the `run` method. (Recall `start` method of `Thread` class that calls `run` method automatically.) In the `run` method, the `repaint` method calls the `paint` method. In `run` method, there is an infinite `for` loop inside that modifies the value of `x` and gives delay of 50 millisecond using `sleep` method. This causes `paint` method to be called after every 50 millisecond. In the `paint` method `msg` is drawn at `x` and `y` position. As `x` is being changed inside `run` method, the string “My Marquee” moves from left to right. When the condition

```
if ((x - 10) > (this.getSize().width))
    x = 1;
```

returns true, that is, `x - 10` is greater than width of the applet window, `x` becomes 1 and marquee starts again from starting position 1.

In case applet is stopped using `stop` menu option within applet window using appletviewer, boolean variable `stop` becomes false and thread is stopped. When it is resumed again using `resume` menu option, it starts again.

It is possible to improve the marquee applet to move the marquee in either direction, from top to bottom, bottom to top or any other. One can try this as well as the two marquees simultaneously.

14.9 `Getdocumentbase()` AND `Getcodebase()` METHODS

These two methods are important in that they return the complete URL for the applets. Both are discussed one by one.

The signature of the method `getDocumentBase()` defined by `Applet` class is:

```
public URL getDocumentBase()
```

The method `getDocumentBase()` gets the URL of the document in which this applet is embedded. For example, suppose an applet is contained within the document:

```
http://java.sun.com/products/jdk/1.5/index.html
```

The document base is:

```
http://java.sun.com/products/jdk/1.5/index.html
```

The signature of the `getCodeBase` method is

```
public URL getCodeBase()
```

The method gets the base URL. This is the URL of the directory which contains this applet. The following example clarifies the above two methods practically.

```
/*PROG 14.6 DEMO OF GETDOCUMENTBASE AND GETCODEBASE */

/*
<html>
<applet>
<applet code = "applet2" width=300 height=100>
</applet>
</html>
*/
import java.awt.*;
import java.applet.*;
import java.net.*;
public class applet2 extends Applet
{
    URL dbase, cbase;
    public void init()
    {
        dbase = getDocumentBase();
        cbase = getCodeBase();
    }
    public void paint(Graphics g)
    {
        g.drawString("Document Base:=" +dbase.toString(), 10, 30);
        g.drawString(" Code Base:=" +cbase.toString(), 10, 55);
    }
}
```

Explanation: The program is simple. In this program, two URL references are created and initialized in the `init` method with `getDocumentBase()` and `getCodeBase()`. The `URL` class is defined in the `java.net` package, so this has to be imported into the program. In the `paint` method the URLs are converted to String form and are displayed using `drawString` method. For more about `URL` class refer the chapter “Networking in Java”.

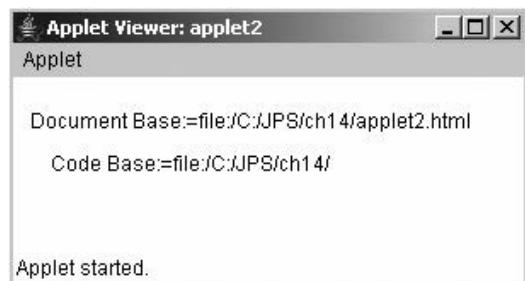


Figure 14.11 Output screen of Program 14.6

14.10 THE APPLETCONTEXT INTERFACE

The interface corresponds to an applet's execution environment: the document containing the applet and the other applets in the same document. The methods in this interface can be used by an applet to obtain information about its environment. The context of the currently executing applet is obtained by a call to the `getAppletContext()` method defined by `Applet`. Some of the abstract methods of the interface are shown in Table 14.2.

| Method Signature | Description |
|--|--|
| <code>Applet getApplet(String name)</code> | Finds and returns the applet in the document represented by this applet context with the given name or null if not found. The name can be set in the HTML tag by setting the name attribute. |
| <code>Enumeration getApplets()</code> | Returns an enumeration of all applets in the documents represented by this applet context. |
| <code>AudioClip getAudioClip(URL url)</code> | Returns an <code>AudioClip</code> object that encapsulates the audio clip found at the location specified by <code>url</code> . |
| <code>Image getImage(URL url)</code> | Returns an <code>Image</code> object that can then be painted on the screen. The <code>url</code> argument that is passed as an argument must specify an absolute URL. |
| <code>void showDocument(URL url)</code> | Replaces the web page currently being viewed with the given URL. This method may be ignored by applet contexts that are not browsers. |
| <code>void showDocument(URL url, String target)</code> | Requests that the browser or applet viewer show the web page indicated by the <code>url</code> argument. The placement of the document is specified by <code>target</code> as described in the text. An applet viewer or browser is free to ignore <code>showDocument</code> . |
| <code>void show(String status)</code> | Requests that the argument string be displayed in the "status window". |

Table 14.2 Methods of `AppletContext` interface

The most important method is `showDocument` method. After obtaining an applet's context another HTML document can be loaded using `showDocument` method. As can be seen from the table above, there are two forms of the `showDocument` method. In the second form, the `target` argument may be one of the following as shown in Table 14.3.

| Target Argument | Description |
|------------------------|---|
| <code>"_self"</code> | Show in the window and frame that contain the applet |
| <code>"_parent"</code> | Show in the applet's parent frame. If the applet's frame has no parent frame, acts the same as <code>"_self"</code> . |
| <code>"_top"</code> | Show in the top-level frame of the applet's window. If the applet's frame is the top-level frame, acts the same as <code>"_self"</code> . |
| <code>"_blank"</code> | Show in a new, unnamed top-level window. |
| <code>name</code> | Show in the frame or window named <code>name</code> . If a target named <code>name</code> does not already exist, a new top-level window with the specified name is created, and the document is shown there. |

Table 14.3 Target argument of `showDocument` method

The following is an example of usage of this method.

```
/*PROG 14.7 DEMO OF APPLET CONTEXT AND SHOWDOCUMENT */

import java.awt.*;
import java.applet.*;
import java.net.*;
public class AppCon extends Applet
{
    public void start()
    {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase();
        try
        {
            ac.showDocument(new URL(url + "demo.htm"));
        }
        catch (MalformedURLException e)
        {
            showStatus("URL not found");
        }
    }
}
```

Explanation: In this program, the start method of the Applet class is overridden. In the method the execution context of the applet is obtained using method `getAppletContext()`. The context is stored in the reference `ac` of `AppletContext` interface. The current directory of the HTML file that is to be displayed is obtained using `getCodeBase` method. The `url` is combined with the "demo.html" file and used as argument in the `showDocument` method. Note the code will not work if applet is run using appletviewer. Create a separate HTML file by the name `AppCon.htm` and put the following code in it.

```
//file AppCon.html
<html>
<applet code ="AppCon" width = 300 height = 100>
</applet>
</html>
```

Now run the above .html file using any browser. IE has been used in Win XP.

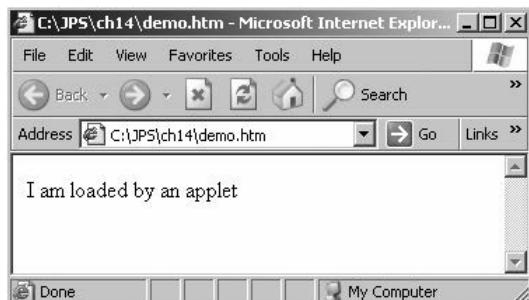


Figure 14.12 Output screen of Program 14.7

14.11 PLAYING AUDIO IN APPLET

For playing audio in applets there are two methods: the first one is to use `play` method defined by `applet` class and second is to use `AudioClip` interface defined by `java.applet` package. Both approaches are taken one by one.

Note: For playing audio in Java, a special package for playing audio and video called Java Media Framework (JMF) must be installed. It can be downloaded free from Sun's website.

14.11.1 Playing Audio Using Play Method of Applet Class

The method has two forms. The signature of the first form is given below:

```
public void play(URL url, String name)
```

The method plays the audio clip given the URL and a specifier that is relative to it. Nothing happens if the audio clip cannot be found. The url is an absolute URL giving the base location of the audio clip. The name is the location of the audio clip, relative to the url argument.

Second form does take only URL as argument:

```
public void play(URL url)
```

The method is illustrated using an example given below:

```
/*PROG 14.8 PLAYING AUDIO IN APPLET VER1 */

/*<html>
<applet code="applet8.class" width =200 height = 100>
</applet>
</html>
*/
import java.awt.*;
import java.applet.*;
import java.net.*;
public class applet8 extends Applet
{
    String msg = " ";
    public void start()
    {
        try
        {
            URL url = new URL(getCodeBase() + "POP1.
mid");
            play(url);
            msg = "Audio clip is beign played";
        }
        catch (MalformedURLException e)
        {
            msg = "URL not found ";
        }
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 40);
    }
}
```

Explanation: The program is simple. A new URL using the getCodeBase and an audio file name, which is to be played, are created. The file name is “POP1.mid”. Here, only audio files are created with wav and mid extensions. The file to be played must be in the current directory of the applet. The URL thus created is passed to the play method. The play method plays the file from beginning to end just once. The audio file is played as soon as applet is started.

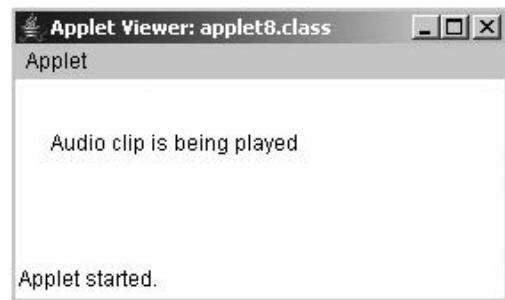


Figure 14.13 Output screen of Program 14.8

14.11.2 Playing Audio Using AudioClip Interface

The `AudioClip` interface is defined by the `java.applet` package. The `AudioClip` interface is a simple abstraction for playing a sound clip. Multiple `AudioClip` items can be played at the same time, and the resulting sound is mixed together to produce a composite. It has got three methods:

(a) `void play()`

This method starts playing this audio clip. Each time this method is called, the clip is restarted from the beginning. The clip is played just once.

(b) `void stop()`

This method when called stops playing this audio clip.

(c) `void loop()`

This method when called starts playing this audio clip in a loop.

The method `getAudioClip` of the `Applet` class returns a reference of this interface. It has the following two forms:

```
public AudioClip getAudioClip(URL url, String name)
public AudioClip getAudioClip(URL url);
```

These methods return the `AudioClip` object specified by the URL and name arguments in first form and only URL in second form. The methods always return immediately whether or not the audio clip exists. When this applet attempts to play the audio clip, the data will be loaded.

An example is given below.

```
/*PROG 14.9 PLAYING AUDIO IN APPLET VER 2 */
```

```
/*
<html>
<applet code="play1.class" width =200 height = 100>
</applet>
</html>
*/
import java.awt.*;
import java.applet.*;
import java.net.*;
public class play1 extends Applet
```

```

{
    String msg = " ";
    public void start()
    {
        try
        {
            URL url = new URL(getCodeBase() + "ROCK1.mid");
            AudioClip ac = getAudioClip(url);
            ac.play();
            msg = "Audio Clip Rock1.mid is being played";
        }
        catch (MalformedURLException e)
        {
            msg = "URL not found ";
        }
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 40);
    }
}

```

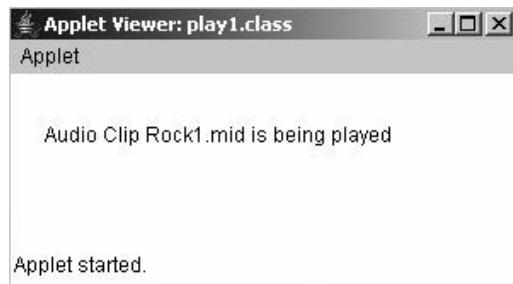


Figure 14.14 Output screen of Program 14.9

Explanation: The program is similar to the previous one but here `AudioClip` interface has been used for playing the audio. The URL here is passed to the `getAudioClip` method and reference returned is stored in reference `ac` of `AudioClip`. The audio clip is then played using `play` method of this reference.

```

/*PROG 14.10 PLAYING AUDIO IN APPLET VER 3 */

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.net.*;

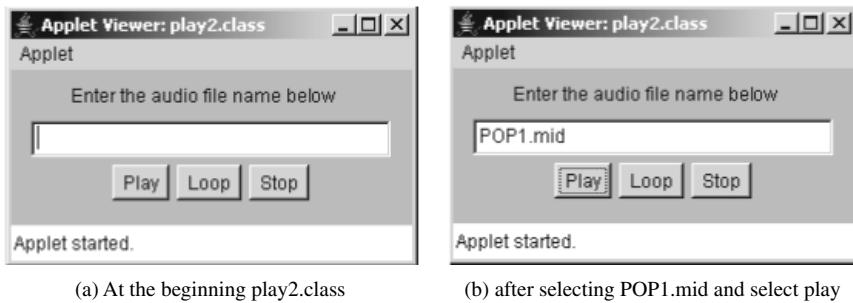
/*
<html>
<applet code="play2.class" width =200 height = 100>

```

```
</applet>
</html>
*/



public class play2 extends Applet implements ActionListener
{
    Button play, stop, loop;
    TextField file;
    Label L;
    public void init()
    {
        setBackground(Color.pink);
        L = new Label("Enter the audio file name belwo");
        file = new TextField(30);
        play = new Button("Play");
        loop = new Button("Loop");
        stop = new Button("Stop");
        add(L);
        add(file);
        add(play);
        add(loop);
        add(stop);
        play.addActionListener(this);
        loop.addActionListener(this);
        stop.addActionListener(this);
    }
    public void actionPerformed(ActionEvent AE)
    {
        URL url;
        AudioClip ac = null;
        try
        {
            url = new URL(getCodeBase() + file.getText());
            ac = getAudioClip(url);
        }
        catch (MalformedURLException e)
        {
            Graphics g = getGraphics();
            g.drawString("Error", 20, 90);
        }
        if (AE.getSource() == play)
            ac.play();
        else if (AE.getSource() == loop)
            ac.loop();
        else
            ac.stop();
    }
}
```

**Figure 14.15** Output screen of Program 14.10

Explanation: Some of the concepts used in this program are covered in subsequent chapters. Readers are advised to refer them first and try the above code. In the program, there is one Label instance, one TextField instance and 3 Button instances. The file name is taken from the user in the text box. When play button is clicked actionPerformed method is called. In the method, using the file name entered in the text box and getCodeBase, a new URL instance is created and depending on which button was pressed, action is taken.

14.12 WORKING WITH GRAPHICS CLASS

To do any drawing at all in Java, a graphics context is needed. A graphics context is an object belonging to the `Graphics` class. Instance methods are provided in this class for drawing shapes, text and images. Any given `Graphics` object can draw to only one location. In this chapter, that location will always be one of Java's GUI components, such as an applet. The `Graphics` class is an abstract class, which means that it is impossible to create a graphics context directly, with a constructor. There are two ways to get a graphics context for drawing on a component: first of all, of course, when a component's `paint()` method is called, the parameter to that method is a graphics context for drawing on the component. Second, to make it possible to draw on a component from outside its `paint()` method, every component has an instance method called `getGraphics()`. This method is a function that returns a graphics context for the component.

The instance method, `getGraphics()`, is defined in the `Component` class. It returns a graphics context that can be used for drawing to a particular component. That is, if `comp` is any component object and if `g` is a variable of type `Graphics`, the following is true.

```
g = comp.getGraphics();
```

After this assignment, `g` can be used for drawing to the rectangular area of the screen that represents the component, `comp`. When writing a particular applet or what other component class, one needs to call the (inherited) `getGraphics()` method in the same class. So it would be simply “`g = getGraphics()`”. This gives a graphics context for drawing in the component being written.

The `Graphics` class provides a number of methods for drawing various graphical shapes like, line, rectangle, circle, arc, polygon, etc. The class is an abstract class and an object of it cannot be initiated. In this section, some of the commonly used graphical methods will be dealt with.

14.12.1 Drawing Lines and Rectangles

For drawing lines, the `Graphics` class provides the method as:

```
void drawLine(int x1, int y1, int x2, int y2)
```

The method draws a line from (x1, y1) to (x2, y2). It can be used as:

```
g.drawLine(10,10,50,10);
```

For drawing rectangle, the Graphics class provides method as:

```
void drawRect(int top, int left, int width, int bottom);
```

For example, `g.drawRect(20, 20, 120, 50);`

The method draws a rectangle whose top left coordinate is given as (20, 20), width 120 and height 50.

The following is another example:

```
int w = this.getSize().width-1;
int h = this.getSize().height-1;
g.drawRect(0, 0, w, h);
```

Remember that `getSize().width` is the width of the applet and `getSize().height` is the height. The upper left-hand corner of the applet starts at (0, 0) not at (1, 1). This means that a 100 by 200 pixel applet includes the points with x coordinates between 0 to 99, not between 0 and 100. Similarly, the y coordinates are between 0 and 199 inclusive, not 0 and 200.

There is no separate `drawSquare()` method. A square is just a rectangle with equal length sides, so to draw a square call `drawRect()` and pass the same number for both the height and width argument.

A rounded rectangle can also be drawn. For this purpose, there is method `drawRoundRect`. The signature is:

```
void drawRoundRect(int top,int left,int width,int height, int
arcWidth,int arcHeight)
```

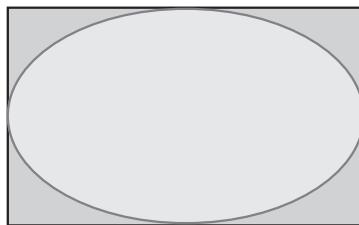
The first four parameters are same as for rectangle. The last two parameters are arc width and arc height for joining the corners of the rectangle so that it will look like a rounded rectangle.

14.12.2 Drawing Ovals and Circles

Java has methods to draw outlined and filled ovals. As one would probably guess, these methods are called `drawOval()` and `fillOval()`, respectively. As one might not guess they take identical arguments to `drawRect()` and `fillRect()`, which is given as follows.

```
public void drawOval(int left,int top,int width,int height);
public void fillOval(int left, int top, int width, int height);
```

Instead of the dimensions of the oval itself, the dimensions of the smallest rectangle which can enclose the oval are specified. The oval is drawn as large as it can touch the rectangle's edges at their centres. This is as shown in the picture:



The arguments to `drawOval()` are the same as the arguments to `drawRect()`. The first `int` is the left-hand side of the enclosing rectangle, the second is the top of the enclosing rectangle, the third is the width and the fourth is the height. For example, to draw a circle at centre (x, y) of radius r the following will be written:

```
g.drawOval(x-r, y-r, 2*r, 2*r);
```

Java also has method to draw outlined and filled arcs. They are similar to `drawOval()` and `fillOval()` but a starting and ending angle must also be specified for the arc. Angles are given in degree. The signatures are as follows:

```
public void drawArc(int left,int top,int width,int height, int
startAngle, int stopAngle)
public void fillArc(int left,int top,int top,int width,int height,
int startAngle, int stopAngle)
```

The rectangle is filled with an arc of the largest circle that could be enclosed within it. The location of 0 degrees and whether the arc is drawn clockwise or counter-clockwise are currently platform dependent.

14.12.3 Drawing Polygon

Polygons are defined by their corners. The polygon seen and created here lie in a 2-D plane. The basic constructor for the `Polygon` class is

```
public Polygon(int[] xpoints, int[] ypoints, int npoints)
```

where `xpoint` is an array that contains the `x` coordinate of the polygon and `ypoints` is an array that contains the `y` coordinate. Both should have the length `npoints`. Thus to construct a 3-4-5 right triangle with the right angle on the origin one would type:

```
int[]xpoint = {0,3,0};
int[]y point =(0,0,4);
Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
```

To actually draw the polygon `Java.awt.Graphics`'s `drawPolygon(Polygon p)` method is used within the `paint()` method like this:

```
g.drawPolygon(myTriangle);
```

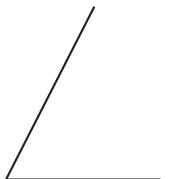
The arrays and number of points can be passed directly to the `drawPolygon()` method if so preferred:

```
g.fillPolygon(xpoints, ypoints, xpoints.length);
```

There is also an overhead `fillPolygon()` method. The syntax is exactly as one expects:

```
g.fillPolygon(myTriangle);
g.fillPolygon(xpoints, ypoints, xpoints.length);
```

Java automatically closes the polygons it draws. That is, it draws polygons that look like the one on the right rather than on the left.



If the polygons are not wanted to be closed, a polyline can be drawn instead with the `drawPolyline()` method of the `Graphics` class.

```
public abstract void drawPolyline(int[]xPoints, int[] yPoints, int
nPoints)
```

14.12.4 Using Color

Java is designed to work with the RGB color system. An RGB color is specified by three numbers that give the level of red, green and blue, respectively, in the color. A color in Java is an object of the class, `Color`. A new color can be constructed by specifying its red, blue and green components. For example,

```
Color myColor = new Color(r, g, b);
```

There are two constructors that can be called in this way. In the one that is commonly used r, g and b are integers in the range 0 to 255. In the other, they are numbers of type float in the range 0.0F to 1.0F. Often, constructing new colors can be avoided altogether, since the `Color` class defines several named constants representing common colors:

| | | | | |
|--------------------------|------------------------------|----------------------------|-----------------------------|-------------------------|
| <code>Color.black</code> | <code>Color.blue</code> | <code>Color.cyan</code> | <code>Color.darkgray</code> | <code>Color.gray</code> |
| <code>Color.green</code> | <code>Color.lightgray</code> | <code>Color.magenta</code> | <code>Color.orange</code> | <code>Color.pink</code> |
| <code>Color.red</code> | <code>Color.white</code> | <code>Color.yellow</code> | | |

The static color name may be in whole lowercase; for example, `Color.black` may be written as `Color.BLACK`.

An alternative to RGB is the HSB color system. In the HSB system, a color is specified by three numbers called the hue, the saturation and the brightness. The hue is the basic color, ranging from red through orange through all the other colors of the rainbow. The brightness is pretty much what it sounds like. A fully saturated color is a pure color tone. Decreasing the saturation is like mixing white or gray paint into the pure color. In Java, the hue, saturation and brightness are always specified by values type `float` in the range from 0.0F to 1.0F. The `Color` class has a static member function named `getHSBColor` for creating HSB colors. To create the color with HSB values given by h, s and b, the following can be written:

```
myColor = Color.getHSBColor(h, s, b);
```

For example, a random color could be made that is as bright and as saturated as possible with:

```
myColor = Color.getHSBColor((float) Math.random(), 1.0F, 1.0F);
```

The type cast is necessary because the value returned by `Math.random()` is of type `double`, and `Color.getHSBColor()` requires values of type `float`. (By the way, one might ask why RGB colors are created using a constructor while HSB colors are created using a static member function. The problem is that two different constructors would be needed, both of them with three parameters of type `float`. Unfortunately, this is impossible. One can only have two constructors if their numbers of type of parameters differ)

For drawing colorful shapes the foreground color can be set as:

```
setForeground(Color.red);
```

The function `setForeground` of `Component` class take just one parameter: the color which is to be set. The argument must be some predefined colors of `Color` class which can be `Color.red`, `Color.blue`, `Color.black`, etc. The red, green, blue, etc., are static members of `Color` class. For example, in the above program just add the line in the beginning of the paint method.

```
setForeground(Color.blue);
```

And all shapes will be in blue color. For setting the background of the applet `setBackground` method can be used which was seen in applet lifecycle program. For filling the closed figure and as drawing pen color, `setColor` method of `Graphics` class can also be used. This is as shown below:

```
g.setColor(Color.red); //sets drawing pen and fill color as red.
```

Similarly, current color can be obtained by using the constructor of `Color` class which takes three arguments of type `int` and represents red, green and blue component of the color. They can take value between 0 and 255. One such usage is:

```
Color col = new Color(234, 45, 123);
g.getColor(col);

g.setColor(new Color(234, 45, 123));
setForeground(new Color(0,0,222));
```

In the above example, red component is 234, green 45 and blue 123.

Note: The default foreground color is black and default background color is light gray.

For getting the current foreground and background color, there are the functions defined by the component class:

```
Color. getBackground();
Color.getForeground();
```

These functions can be used for obtaining the current foreground and background color as:

```
Color bg = getbackground();
Color bg = getForeground();
```

14.12.5 Setting the Drawing Mode

In Java, for drawing anything onto the screen, two drawing modes can be used. The default mode is the paint mode and other mode is the XOR mode. The default mode is the paint mode where if one shape is drawn over another, the second one hides the overlapped portion and is completely visible. The XOR mode is discussed first.

For setting the XOR mode, the method `setXORMode` can be used. Its signature is as shown below:

```
public abstract void setXORMode(Color c1)
```

Here, `c1` specifies the color that will be XORed to the window when an object is drawn. The method specifies that logical pixel operations are performed in the XOR mode, which alternates pixels between the current color and a specified XOR color.

For restoring the main drawing mode, use the method `setPaintMode`. Its signature is given below:

```
public abstract void setPaintMode()
```

The method sets the paint mode of this graphics context to overwrite the destination with this graphics context's current color. This sets the logical pixel operation function to the paint or overwrite mode.

```
/*PROG 14.11 DEMO FOR XOR MODE VER 1 */
```

```
/*
<html>
<applet code="applet3.class" width =200 height = 100>
</applet>
</html>

*/
import java.awt.*;
```

```

import java.applet.*;
import java.net.*;
public class applet3 extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.fillRect(20, 20, 80, 80);
        g.setXORMode(Color.magenta);
        g.fillRect(40, 40, 90, 90);
        g.setPaintMode();
        g.fillOval(100, 60, 80, 90);
    }
}

```

Explanation: In this program, a filled rectangle is drawn using blue color of width and height 80 pixels. The paint mode is then set to XOR mode with XOR color to magenta using function `setXORMode`. Next, a filled rectangle is drawn whose left-top coordinates are 40–40, with width and height of 90 pixels. The coordinates of second rectangle are chosen in such a manner that the second rectangle overlaps the first. Now, due to XOR mode, overlapped pixels of two rectangles are drawn in magenta color and uncovered portion of second rectangle is drawn in cyan color. Next, drawing mode is restored to paint mode using `setPaintMode` method. The method causes the current fill and pen cooler to reset to original color as set using `setColor` method.

For the following two programs concepts of event handling must be clear in order to cover it in the next chapter.

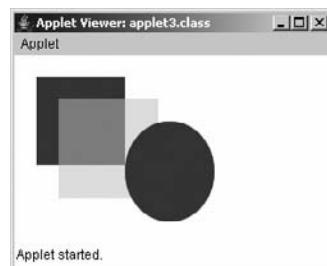


Figure 14.16 Output screen of Program 14.11

```

/*PROG 14.12 DEMO OF XOR MODE VER 2 */

/*
<html>
<applet code="applet4.class" width =200 height = 100>
</applet>
</html>

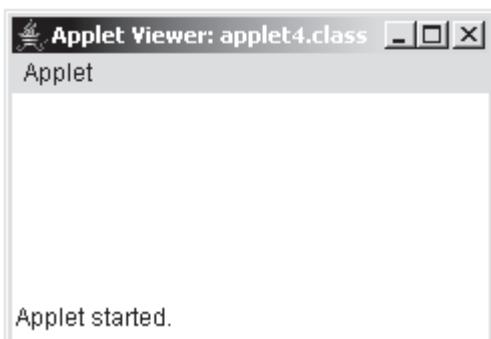
*/
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class applet4 extends Applet
{
    Graphics gr;
    boolean click = false;
    public void init()
    {
        addMouseListener(new MMA());
    }
    class MMA extends MouseAdapter

```

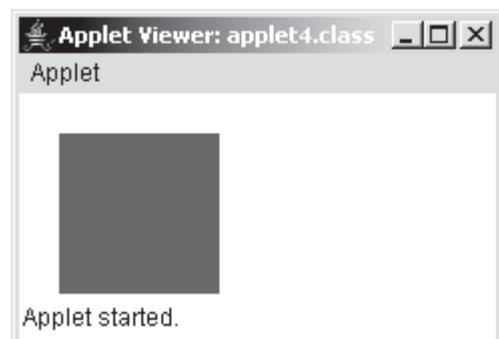
```

    {
        public void mouseClicked(MouseEvent me)
        {
            click = true;
            repaint();
        }
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.fillRect(20, 20, 80, 80);
        if (click == false)
            g.setXORMode(getBackground());
        else
            g.setPaintMode();
        g.fillRect(20, 20, 80, 80);
    }
}

```



(a) Applet at the beginning



(b) Applet after clicking on the applet

Figure 14.17 Output screen of Program 14.12

Explanation: In this program in paint method, when click is false the XOR mode is set. The color passed is the current background color. The else part is skipped and a filled rectangle is drawn onto the same place where rectangle in blue color was drawn. This results in wiping the previous rectangle drawn and nothing is visible onto the screen. This is all because of line.

```
g.setXORMode(getBackground());
```

The boolean variable click was set to true when the mouse was clicked. In this case, else part executes and rectangle drawn is visible. This technique can be used to draw rubber band lines.

```
/*PROG 14.13 DRAWING RUBBER BAND LINE */
```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*

```

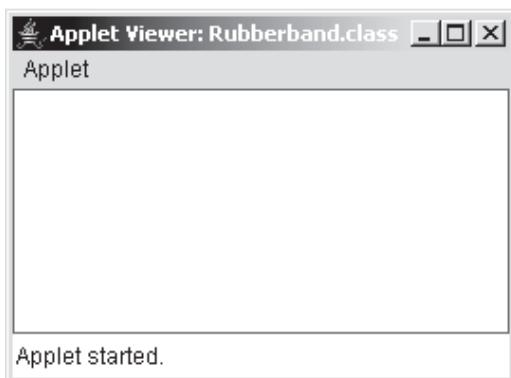
```
<html>
<applet code="Rubberband.class" width = 200 height = 100>
</applet>
</html>
*/
public class Rubberband extends Applet implements
MouseListener, MouseMotionListener
{
    boolean drag;
    Graphics dragGr;
    int sx, sy, px, py;
    public void init()
    {
        setBackground(Color.white);
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        g.drawRect(0, 0, getSize().width - 1,
                   getSize().height - 1);
    }
    public void mousePressed(MouseEvent evt)
    {
        dragGr = getGraphics();
        dragGr.setColor(Color.blue);
        dragGr.setXORMode(getBackground());
        sx = px = evt.getX();
        sy = py = evt.getY();
        drag = true;
        if (evt.isShiftDown())
        {
            repaint();
            return;
        }
        dragGr.drawLine(sx, sy, px, py);
    }
    public void mouseReleased(MouseEvent evt)
    {
        if (drag)
        {
            dragGr.setPaintMode();
            dragGr.drawLine(sx, sy, px, py);
            dragGr.dispose();
            drag = false;
        }
    }
    public void mouseDragged(MouseEvent evt)
    {
        if (drag)
        {
            dragGr.drawLine(sx, sy, px, py);
        }
    }
}
```

```

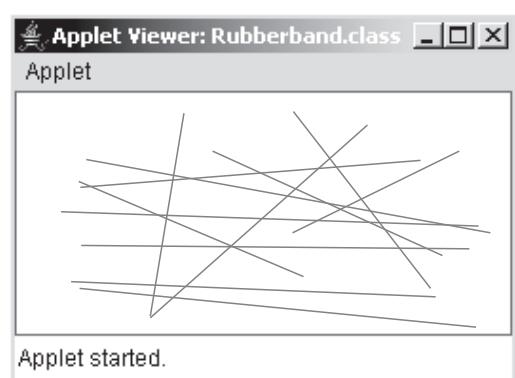
        px = evt.getX();
        py = evt.getY();
        dragGr.drawLine(sx, sy, px, py);
    }
}

public void mouseClicked(MouseEvent evt) { }
public void mouseEntered(MouseEvent evt) { }
public void mouseExited(MouseEvent evt) { }
public void mouseMoved(MouseEvent evt) { }
}

```



(a) Applet at the beginning



(b) Applet after drawing straight lines

Figure 14.18 Output screen of Program 14.13

Explanation: XOR mode can be used to implement a “rubber band cursor”. A rubber band cursor is commonly used to draw straight lines. When the user clicks and drags the mouse, a moving line stretches between the starting point of the drag and the current mouse location. When the user releases the mouse, the line becomes a permanent part of the image. While the mouse is being dragged, the line is drawn in XOR mode. When the mouse moves, the line is first redrawn in its previous position. In XOR mode, this second drawing operation erases the first. Then, the line is drawn in its new position. When the user releases the mouse, the line is erased one more time and is then drawn permanently using paint mode. The same idea can be used for other figures besides lines.

In the program when the user first presses the mouse button starting mouse coordinates are stored in sx and sy in mousePressed method. The same points are stored in px and py, too. The graphics context for the applet window is stored in Graphics instance dragGr using getGraphics method. The paint mode is set to XOR mode using setXORMode method. As stated above while the mouse is being dragged, the line is drawn in XOR mode. As soon as mouse is released, the line is visible as straight line.

14.12.6 Programming Example

```
/*PROG 14.14 DRAWING BASIC SHAPES */
```

```

import java.awt.*;
import java.applet.*;
public class shapes extends Applet
{

```

```

public void paint(Graphics g)
{
    g.setColor(Color.red);
    setBackground(Color.white);
    g.drawLine(10, 10, 100, 100);
    g.setColor(Color.black);
    g.drawRect(20, 20, 120, 80);
    g.setColor(Color.darkGray);
    g.drawOval(200, 20, 100, 50);
    g.drawRoundRect(200, 100, 80, 60, 10, 20);
    g.setColor(Color.magenta);
    g.drawArc(100, 150, 80, 80, 0, 120);
    g.drawOval(130, 240, 80, 80);
}
/*shapes.html
<html>
<applet code="shapes" width =200 height = 100>
</applet>
</html>
*/

```

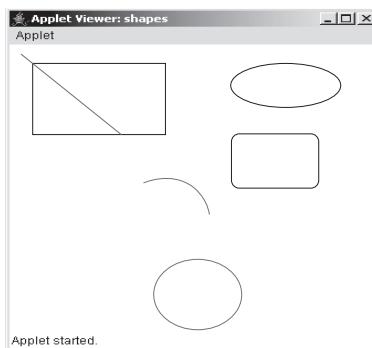


Figure 14.19 Output screen of Program 14.14

Explanation: The program is self-explanatory.

```

/*PROG 14.15 DRAWING & FILLING POLYGONS */

import java.awt.*;
import java.applet.*;
public class shapes1 extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        setBackground(Color.blue);
        int[] xpoints = { 100, 350, 150 };
        int[] ypoints = { 200, 100, 50 };
        Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
    }
}

```

```

        g.drawPolygon(myTriangle);
        xpoints = new int[] { 250, 400, 320 };
        ypoints = new int[] { 250, 250, 120 };
        myTriangle = new Polygon(xpoints, ypoints, 3);
        g.fillPolygon(myTriangle);
    }
}
/*File shapes1.html
<html>
<applet code="shapes1" width = 200 height = 100>
</applet>
</html>
*/

```

Explanation: The program draws a hollow polygon from points (100, 200) to (350, 100) and from (350, 100) to (15, 50). In the second case, when drawing a filled polygon, the same two arrays are initialized with new coordinates. Note that the polygon can be directly drawn and filled without making use of Polygon class as explained earlier.

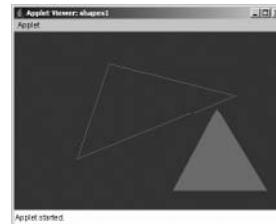


Figure 14.20 Output screen of Program 14.15

```

/*PROG 14.16 DRAWING OCTAGON */

import java.awt.*;
import java.applet.*;
public class octagon extends Applet{
    public void paint(Graphics g){
        int r = 150;
        int i, x = 8;
        int angle = 360/x;
        int arrx[] = new int[x];
        int arry[] = new int[x];
        g.setColor(Color.blue);
        for(i = 0; i<360; i+=angle)
        {
            arrx[i/angle]=(int)(200+r*Math.cos(3.14/180*i));
            arry[i/angle]=(int)(200+r*Math.sin(3.14/180*i));
        }
        g.drawPolygon(arrx, arry, x);
    }
}
/* octagon.html
<html>
<applet code="octagon" width =200 height = 100>
</applet>
</html>
*/

```

Explanation: The number of sides (i.e., 8 for octagon) is stored in variable `x`. Two arrays `arrx` and `arry` have been created for storing the `x` and `y` coordinates of the octagon, respectively. Each side of the octagon is placed at an angle $360/8 = 45$ degree. We know that if θ is the angle and r is the length then polar coordinates (θ, r) , Cartesian coordinates can be obtained as $x = r \cos \theta$ and $y = r \sin \theta$. Here θ is 45 and r is 150 which is the length of one side of octagon in pixels. To store all the coordinates in the array a loop is run from 0 to 360 with step size equal to 45. The angle has to be converted into radians. The respective `x` and `y` coordinates are stored in the array `arrx` and `arry` and then outside the loop they are used in drawing octagon.

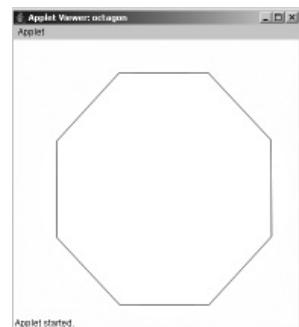


Figure 14.21 Output screen of Program 14.16

Note: The program can be used to draw any type of polygon—be it triangle, rectangle, pentagon or a polygon of any number of sides. For the value of `x` is simply changed.

```
/*PROG 14.17 DRAWING CONCENTRIC FILLED CIRCLE */
```

```
import java.applet.*;
import java.awt.*;
public class concircle extends Applet
{
    public void paint(Graphics g)
    {
        int AH = this.getSize().height;
        int AW = this.getSize().width;
        for (int i = 4; i >= 0; i--)
        {
            if ((i % 2) == 0)
                g.setColor(Color.blue);
            else
                g.setColor(Color.green);
            int RH = AH * i / 4;
            int RW = AW * i / 4;
            int RL = AW / 2 - AW * i / 8;
            int RT = AH / 2 - AH * i / 8;
            g.fillOval(RL, RT, RW, RH);
        }
    }
/*
<html>
<applet code="concircle" width =200 height = 100>
</applet>
</html>
*/
```

Explanation: This pointer gives reference to the current component object (i.e., applet). Use of `getSize().width` and `getSize().height` gives width and height of the applet window. To draw filled concentric circles, the width and the height of the enclosing rectangle around the circles are to be assumed first. After fixing the height and width, for example, as `RH` and `RW`, respectively, the top and left coordinates of the rectangle can be found out. If `AH` and `AW` represents the height and width of the applet window and $(AW/2, AH/2)$ gives the center of the window. The left-top coordinates of the enclosing rectangle is given as:

$$RL = AW/2 - RW/2 \text{ and } RH = AH/2 - RH/2$$

For a circle `RH` and `RW` will be same. Putting the value of `RH` and `RW` the following is obtained.

$$RL = AW/2 - AW/4 \text{ and } RT = AH/2 - AH/4$$

Note: This is general formula; for using this loop one has to make use of control variable for varying the width, height and top left coordinates of the rectangle as shown in the program.

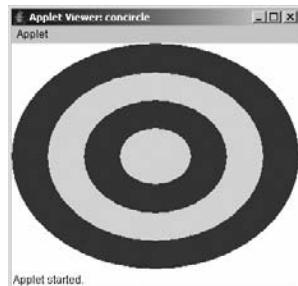


Figure 14.22 Output screen of Program 14.17

```
/*PROG 14.18 DRAWING RANDOM LINES KEEPING ONE POINT FIXED */
```

```
import java.applet.*;
import java.awt.*;
import java.util.*;
public class conline extends Applet
{
    int h, w;
    public void init()
    {
        h = this.getSize().height;
        w = this.getSize().width;
        setBackground(Color.black);
    }
    public void paint(Graphics g)
    {
        for (int i=0;i<=500;i++)
        {
            int x = (int)(Math.random()*10000);
            int y = (int)(Math.random()*10000)3;
            x = x%h;
            y = y%w;
            g.drawLine(h/2, w/2, x, y);
            g.setColor(new Color(x%255, y%255, w%255));
        }
    }
    /*html file
    <html>
    <applet code="conline" width = 350 height = 350>
    </applet>
    </html>
*/
```

Explanation: In the paint method, random numbers between 1 and 1000 are generated and stored in x and y. $x\%w$ ensures that numbers generated randomly are within the width and height of the applet window. One end of the line at the centre of the window is kept by writing the (x_1, y_1) as $(h/2, w/2)$. The second point of line is x and y which changes in every iteration of the loop. For generating random colors, `Color` class is made use of. The constructor of this class takes three components of color as red, green and blue. The values of these components are within the range 0–255. Due to $x\%255$, $y\%255$ different colors are generated in each iteration of the loop.

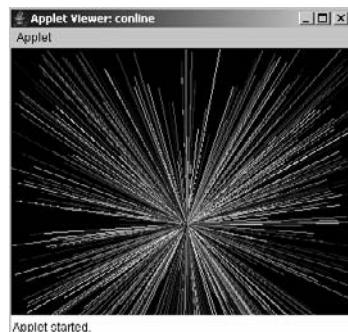


Figure 14.23 Output screen of Program 14.18

```
/*PROG 14.19 DRAWING CONCENTRIC CIRCLES */

import java.applet.*;
import java.awt.*;
import java.util.*;
public class concircle1 extends Applet
{
    int h, w;
    public void init()
    {
        h = this.getSize().height;
        w = this.getSize().width;
        setBackground(Color.black);
    }
    public void paint(Graphics g)
    {
        for(int i=11;i>=1;i--)
        {
            int RH = h*i/16;
            int RW = w*i/16;
            int RL = w/2 - RW/2;
            int RT = h/2 - RH/2;
            g.setColor(new Color(w%255, h%255, RT%255));
            g.drawOval(RL, RT, RW, RH);
        }
    }
}
/* concircle1.html
<html>
<applet code="concircle1" width = 350 height = 350>
</applet>
</html>
*/
```

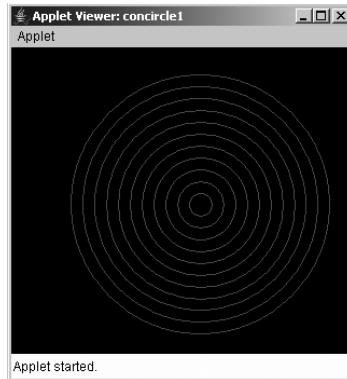


Figure 14.24 Output screen of Program 14.19

Explanation: The logic to generate concentric circles was explained earlier. The difference here is that hollow circles are drawn instead of filled circles.

14.13 WORKING WITH FONTS

A font represents a particular size and style of text. The same character will appear different in different fonts. In Java, a font is characterized by a font name, a style and a size. To work with the fonts, Java provides the `Font` class. The `Font` class represents fonts, which are used to render text in a visible way. The `Font` class is defined by the package `Java.awt`. A font can have many faces, such as heavy, medium, oblique, gothic and regular. All of these faces have similar typographic design. There are three different names that one can get from `Font` object. The logic font name is simply the name that was used to construct the font. The font face name, or just font name for short, is the name of a particular font face, like Helvetica Bold. The family name is the name of the font family that determines the typographic design across several faces, like Helvetica.

The available font names are system dependent, but one can always use the following five strings as font names: "Serif", "SansSerif", "Monospaced", "Dialog" and "DialogInput". These fonts are known as logical fonts. All the system-dependent fonts are known as physical fonts.

To create and use a font in a program the `Font` class constructor can be used as follows:

```
Font (String name, int style, int size);
```

The name is the font name defined by the system font libraries (e.g., "Courier", "Times new Roman", "Arial", etc.). The style may be `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` and `Font.BOLD+Font.ITALIC`. These are the constants defined by `Font` class. To set more than one style the symbol (`|`) can be used (e.g., `Font.BOLD | Font.ITALIC`). The size specifies size in points.

For example:

```
Font myfont = new Font("Arial", Font.BOLD, 14);
```

The example given above creates a new font object named `myfont` which represents Arial bold font of point size 14. After creating a font object it can be set using `setFont` method as:

```
setFont (myfont);
```

Some of the methods of `Font` class are given in the table below:

| Method Signature | Description |
|-----------------------|---|
| String getFamily() | Returns the name of the font family to which the invoking font belongs. |
| String getFontName() | Returns the font face name of this font. For example, Helvetica Bold could be returned as a font face name. |
| String getName() | Returns the logical name of this font. |
| public int getStyle() | Returns the style of this font. The style can be plain, bold, italic, or bold + italic. |
| boolean isBold() | Indicates whether or not this font object's style is bold . |
| boolean isItalic() | Indicates whether or not this font object's style is Italic. |
| boolean isPlain() | Indicates whether or not this font's style is Plain. |
| int getSize() | Returns the size, in points, of the invoking font. |

Table 14.4 Methods of Font class

The Font class also defines certain static variables as properties of the font.

| Properties Name | Description |
|---------------------------|--|
| protected String name | The logical name of this Font, as passed to the constructor. |
| protected int style | The style of this font, as passed to the constructor. this style can be plain, bold, italic, or bold + italic. |
| protected int size | The point size of this font, rounded to integer. |
| protected float pointSize | The point size of this font in float. |

Table 14.5 Properties of Font class

```
/*PROG 14.20 DISPLAYING FONT INFORMATION */
```

```
import java.applet.*;
import java.awt.*;
public class fontinfo extends Applet
{
    public void paint(Graphics g)
    {
        Font f = g.getFont();
        String fName = f.getName();
        String fFamily = f.getFamily();
        int fSize = f.getSize();
        int fStyle = f.getStyle();
        String fstr ="Font Name:= "+fName;
        g.drawString(fstr, 5, 15);
        fstr ="Font Family:= "+fFamily;
        g.drawString(fstr, 5, 35);
        fstr ="Font Size:= "+fSize;
        g.drawString(fstr, 5, 55);
        fstr ="Font Style:= ";
        if (f.isBold())
            fstr += "Bold";
```

```

        if (f.isItalic())
            fstr += "Italic";
        if (f.isPlain())
            fstr += "Plain";
        g.drawString(fstr, 5, 75);
        g.drawString("Complete Font Description", 5, 90);
        g.drawString(f.toString(), 5, 110);
    }
}

```

Explanation: In this program, the font associated with the `Graphics` class is retrieved into font instance `f`. From this font instance, the font name, font family, size, style, etc., are determined using methods of `Font` class. All these information are displayed using `drawstring` method. In the end, the font instance `f` is also displayed in String form using `toString` method. This displays the complete font information which was displayed individually above various methods of `Font` class.

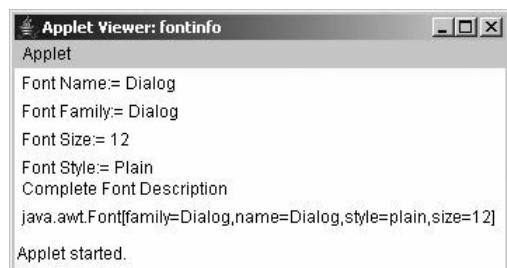


Figure 14.25 Output screen of Program 14.20

```

/*PROG 14.21 USING LOGICAL FONTS */

import java.applet.*;
import java.awt.*;
public class fontdemo extends Applet
{
    public void paint(Graphics g)
    {
        g.setFont(new Font("Serif", Font.PLAIN, 14));
        String fstr = "Font Demo in \"Serif\"";
        g.drawString(fstr, 5, 15);

        g.setFont(new Font("SansSerif", Font.PLAIN, 14));
        fstr = "Font Demo in \"Dialog\"";
        g.drawString(fstr, 5, 35);

        g.setFont(new Font("Dialog", Font.PLAIN, 14));
        fstr = "Font Demo in \"Dialog\"";
        g.drawString(fstr, 5, 55);

        g.setFont(new Font("DialogInput", Font.PLAIN, 14));
        fstr = "Font Demo in \"DialogInput\"";
        g.drawString(fstr, 5, 75);

        g.setFont(new Font("Monospaced", Font.PLAIN, 14));
        fstr = "Font Demo in \"Monospaced\"";
        g.drawString(fstr, 5, 95);
    }
}

```

Explanation: The program is simple. All five logical fonts defined by Java are used and a string displayed after setting a specified font.

14.14 THE FONTMETRICS CLASS

Often, when drawing a string, it is important to know how big the image of the string will be. This information is needed if one wants to centre a string on an applet, or wants to know how much space to leave between two lines of text when they are drawn one above the other, or if the user is typing the string and a cursor is needed to be positioned at the end of the string. In Java, questions about the size of a string are answered by an object belonging to the standard class `java.awt.FontMetrics`. The `FontMetrics` class defines a font metrics object, which encapsulates information about the rendering of a particular screen.

There are several lengths associated with any given font. Some of them are shown in this illustration:

The line in the illustration just touching the bottom of each character in the two strings and crossing the character g are the base line of the two lines text. The suggested distance between two baselines, for single-spaced text, is known as the `lineheight` of the font. The ascent is the distance that tall characters can rise above the baseline, and the descent is the distance that tails like the one on the letter g which can descend below the baseline. The ascent and descent do not add up to the `lineheight`, because there should be some extra space called leading. All these quantities can be determined by calling instance methods in a `FontMetrics` object. There are also methods for determining the width of a character and that of a string.

If F is a font and g is a graphics context, a `FontMetrics` object is obtained for the font F by calling `g.getFontMetrics(F)`. If fm is a variable that refers to the `FontMetrics` object, the ascent, descent, leading and `lineheight` of the font can be obtained by calling `fm.getAscent()`, `fm.getDescent()`, `fm.getLeading()` and `fm.getHeight()`. If ch is a character, `fm.charWidth(ch)` is the width of the character when it is drawn in that font. If str is a string, `fm.stringWidth(str)` is the width of the string.

The following example shows the message “Hello World” in the exact centre of the component by using `FontMetrics` class and its various methods.

```
/*PROG 14.22 DISPLAYING TEXT IN CENTRE */

import java.awt.*;
import java.applet.*;
/*
<html>
<applet code="centertext.class" width = 350 height = 120>
</applet>
</html>
*/
public class centertext extends Applet
{
    public void paint(Graphics g)
    {
        int width, height;
        int x, y;
        Font F = g.getFont();
        FontMetrics fm = g.getFontMetrics(F);
        width = fm.stringWidth("Hello World");
        height = fm.getAscent();
```

```

        x = getSize().width / 2 - width / 2;
        y = getSize().height / 2 + height / 2;
        g.drawString("Hello World", x, y);
    }
}

```



Figure 14.26 Output screen of Program 14.22

Explanation: The variables x and y denote starting point of baseline of string.

Note: There are no tails on any of the chars in the string “Hello World”. For calculating x one would go to centre and subtract half the width of the string and for y to centre, and then move down half the height of the string. Now try resizing the applet window; the text will always be in the centre of the window.

```
/*PROG 14.23 DISPLAYING MULTILINE TEXT USING FONTMETRICS */
```

```

import java.applet.*;
import java.awt.*;
/*
<html>
<applet code="multiline.class" width =350 height = 120>
</applet>
</html>
*/
public class multiline extends Applet
{
    int cx = 0, cy = 0;
    public void init()
    {
        Font f = new Font("Comic Sans MS", Font.ITALIC, 14);
        setFont(f);
    }
    public void paint(Graphics g)
    {
        FontMetrics fm = g.getFontMetrics();
        newLine("First Line.", g);
        sameLine("On First Line again", g);
        newLine(" Second Line", g);
        newLine(" Third Line", g);
    }
}

```

```

        sameLine("On Third line again", g);
    }
    void newLine(String s, Graphics g)
    {
        FontMetrics fm = g.getFontMetrics();
        cy += fm.getHeight();
        cx = 0;
        g.drawString(s, cx, cy);
        cx = fm.stringWidth(s);
    }
    void sameLine(String s, Graphics g)
    {
        FontMetrics fm = g.getFontMetrics();
        g.drawString(s, cx, cy);
        cx += fm.stringWidth(s);
    }
}
}

```

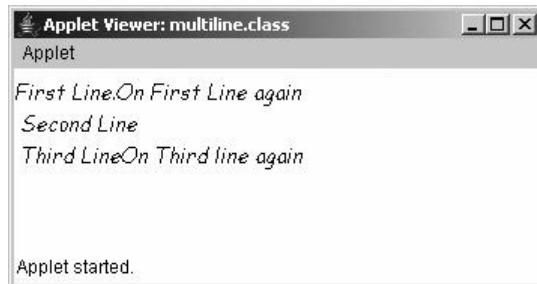


Figure 14.27 Output screen of Program 14.23

Explanation: For finding the total height of the text, the `getHeight` method is used. For finding width of the string as seen in the previous example, the `stringWidth` method is used. When `newLine` method is called, `FontMetrics` instance of `Graphics` class is obtained using `getFontMetrics` method. This is stored in `fm`. The height of the string `s` is obtained using `getHeight` method. This is added to the `cy`. As string is to be displayed on to the new line, `cx` is set to 0. In the `sameLine` method, the `cx` is advanced by adding the width of the string; no change is made to the `cy`.

14.15 PONDERABLE POINTS

1. Java applets are small programs that are meant to run on a page in a web browser. They can also be run locally using appletviewer tool.
2. For creating an applet the program class must extend `Applet` class which is defined in package `java.applet`.
3. Applets are different from console applications as they do not have main method.
4. There are mainly four methods defined in the applet class: `init`, `start`, `destroy` and `stop`.
5. For embedding an applet program in an HTML file `APPLET` tag must be used.
6. For drawing onto applet window the `paint` method and `Graphics` class can be made use of.
7. The `Graphics` class defines a number of useful methods for drawing various shapes and images.
8. Parameters to applets can be passed using `PARAM` tag. All `PARAM` parameters return `String` value which is accessed using `getParameter` method.

9. The repaint method calls paint method through update method.
10. AppletContext interface can be used for applet-to-applet communication.
11. The showDocument method of AppletContext interface can be used for transferring control to other HTML page.
12. The Font class encapsulates a font in Java.
13. Fonts are of two types: logical and physical. Physical fonts are platform dependent whereas logical fonts are defined by the JVM.
14. The FontMetrics class encapsulates information about the rendering of a particular font on a particular screen.

REVIEW QUESTIONS

1. Explain the life cycle of an Applet.
2. Write an applet to illustrate the life cycle of an Applet.
3. Explain the order of method invocation in an Applet.
4. How is an applet converted to an application?
5. What are the interfaces available in applet class?
6. What are the differences between an Applet and an Application?
7. Write an applet to display an image and play a song.
8. Explain the Applet Context. How do you make Applet-to-Applet communication?
9. Are getCodeBase() and getDocumentBase() the same? Explain.
10. What is the use of appletviewer and show Status Method?
11. Explain applet tag with an example.
12. What are the mandatory attributes in Applet tag?
13. Write an applet to display current date and time.
14. Write an Applet for Growing Text and Growing Image.
15. Use setBackground and setForeground method to randomly change the colors of an applet.
16. What is the default layout of an Applet?
17. Write short notes on the following methods:

| | |
|----------------|-------------|
| (a) Font class | (c) Repaint |
| (b) Update | (d) Paint |
18. Your file consists of the following code:


```
//set a
import java.awt.Graphics;
import java.applet.Applet;
//set b
import java.awt.*;
import java.applet.*
```

Which set takes less compile time? Prove it through example.
19. Describe the arguments used in the drawRect() and drawRoundedRect().
20. Explain the drawArc() and fillArc() method.
21. Write an application to draw the following shapes with different color:

| | |
|---------------|-------------|
| (a) Line | (e) Oval |
| (b) Poly line | (f) Ellipse |
| (c) Clipping | (g) Arc |
| (d) Circle | (h) Polygon |
22. Write an applet to draw the human face.
23. What is clipping? Explain with an example.
24. Write an applet to draw National flag with color.
25. Write a program to display the small rounded rectangles in a specified area, using clipping concept.

Multiple Choice Questions

1. Java applets are small programs that are meant to run on a page in a Web browser. They can also be run locally using

| | |
|-----------|-----------------------|
| (a) Java | (c) Appletviewer |
| (b) Javac | (d) None of the above |
2. Applet class is defined in

| |
|-------------------------|
| (a) java.applet package |
|-------------------------|
3. Applets are different from console applications as they do not have

| | |
|-----------------|-----------------------|
| (a) run method | (c) start method |
| (b) main method | (d) none of the above |

4. Which of the following is used for applet communication?
 - (a) Wait and notify
 - (b) Context switching
 - (c) AppletContext interface
 - (d) Runnable interface
5. The _____ method of AppletContext interface can be used for transferring control to other HTML page.
 - (a) appletviewer
 - (b) init
 - (c) showDocument
 - (d) showApplet
6. Fonts are of two types: logical and physical. Physical fonts are platform independent whereas logical fonts are:
 - (a) platform independent
 - (b) platform dependent
7. For showing window and frame that contain the applet _____ target argument is used.
 - (a) "_parent"
 - (c) "_self"
 - (b) "_top"
 - (d) none of the above
8. The first method called for any applet is
 - (a) start()
 - (c) play()
 - (b) init()
 - (d) none of the above
9. Which of the following is the method of applet life cycle:
 - (a) Public void destroy()
 - (b) AppletContext()
 - (c) String getAppletInfo()
 - (d) URL getCodeBase()
10. The repaint() method is defined by
 - (a) Applet
 - (c) Collection Framework
 - (b) AWT
 - (d) none of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. c 2. a 3. b 4. c 5. c 6. c 7. c 8. b 9. a 10. b

Event Handling

15

15.1 INTRODUCTION

An event is any happening or occurring. Event-handling code is the heart of every useful application. All event-driven programs are structured around their event-processing model. Java events are a part of the Java Abstract Windowing Toolkit (AWT) package. An applet is basically an event-driven program in which events are generated by mouse, keyboard and window or other graphical user interface components. The events are passed to the events methods and there are specific methods for recognizing and handling the events.

There are two main models for handling events in Java: the old model, called the Inheritance Event Model, is obsolete; the new model, called the Delegation Event Model, will be discussed in detail.

There were a few flaws with the Java 1.0 Inheritance Event Model. One major problem was that an event could only be handled by the component that generated it or by one of the containers that contained the original component.

Another drawback with the Java 1.0 Inheritance Event Model was that a large number of CPU cycles were wasted on understanding events. Any event in which a program had no interest would just flow through the containment hierarchy before it was eventually discarded. The original model provided no way to disable irrelevant events.

With the new 1.1 Delegation Event Model, a component can be told which object or objects should be notified when it generates a particular type of event. If a component is not interested in an event type, then events of that type will not be propagated.

The primary design goals of the new model in the AWT are the following:

1. Make it simple and easy to learn.
2. Support a clean separation between an application and GUI code.
3. Facilitate a clean creation of robust event which is less error-prone (strong compile-time checking).
4. Make it flexible enough to enable varied application models for event flow and propagation.
5. Enable, for visual tool builders, run-time discovery of both events that a component generates as well as the events it may observe.
6. Support backward binary compatibility with the old model.

15.2 THE DELEGATION EVENT MODEL

The 1.1 Event Model is based on the concept of an “Event Source” and “Event Listeners”. Any object that is interested in receiving messages (or events) is called an Event Listener, and any object that generates these messages (or events) is called an Event Source.

The Event Source Object maintains a list of interested objects in receiving events that it produces. The Event Source Object provides method that allows the listeners to add themselves ('register') or remove

themselves from this list of ‘interested’ objects. When the Event Source Object generates an event, or when user input event occurs on the Event Source Object, the Event Source Object notifies all the listeners that the event has occurred.

An event is propagated from a “Source” object to a “Listener” object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated. A Listener is an object that implements a specific `EventListener` interface that extends from the generic `java.util.EventListener`. An `EventListener` interface defines one or more methods that are to be invoked by the event source in response to each specific event type handled by the interface. An Event Source is an object that originates or “fires” events.

In an AWT program, the event source is typically a GUI component and the listener is commonly an “adapter” object which implements the appropriate listener (or set of listeners) in order for an application to control the flow/handling of events.

15.3 THE EVENT CLASSES

The Java 1.1 Event Model defines a large number of event classes. At the root of the Java event class hierarchy is `java.util.EventObject`. Every event is a subclass of `java.util.EventObject`. It is a very general class with only one method of interest.

```
Object getSource()
```

This method returns the object that originated the event. Every event has a source object, from which the event originated. This method returns a reference to that source.

`java.awt.AWTEvent`: AWT events, which is the main concern here, are subclasses of `java.awt.AWTEvent`. This is the super class of all the delegation model event classes. The most interesting method in this class is:

```
int getID()
```

This method returns the ID of the event. An event’s ID is an `int` that specifies the exact nature of the event. This value is used to distinguish the various types that are represented by any event class.

`java.awt.event` is the subclass of `java.awt.AWTEvent`, which represents the various types that can be generated by the various AWT components. All the various types of AWT events are placed in a separate package called `java.awt.event` for the sake of convenience.

| Class | Description |
|-----------------|---|
| ActionEvent | Generated by component activation, like when a button is pressed. |
| AdjustmentEvent | Generated by adjustment of adjustable components such as scroll bars. |
| ContainerEvent | Generated when components are added to or removed from a container. |
| FocusEvent | Generated when a component receives input focus. |
| ItemEvent | Generated when an item is selected from a list, choice or check box. |
| KeyEvent | Generated by keyboard activity. |
| MouseEvent | Generated when a component is painted. |
| PaintEvent | Generated when a text component is modified. |
| TextEvent | Generated when a text component is modified. |
| WindowEvent | Generated by window activity like minimizing or maximizing. |

Table 15.1 Java Event class

15.4 EVENT LISTENERS

Event Listeners are objects that are responsible for handling a particular task of a component. They are notified when an event is occurred. The Listener typically implements the interface that contains event-handling code for a particular component. An EventListener interface will typically have a separate method for each distinct event type the event class represents.

The methods that receive and process events are defined in a set of interfaces found in `java.awt.event`. For example, the `FocusListener` interface defines two methods, `focusGained()` and `focusLost()`, one for each event type that `FocusEvent` class represents.

Some of the listener interfaces are shown below:

```
java.awt.event.ComponentListener
java.awt.event.ContainerListerner
java.awt.event.FocusListerner
java.awt.event.keyListener
java.awt.event.MouseListerner
java.awt.event.MouseMotionListerner
java.awt.event.WindowListerner
java.awt.event.ActionListener
java.awt.event.AdjustmentListerner
java.awt.event.ItemListerner
java.awt.event.TextListerner
```

15.5 EVENT SOURCES

A source is an object that generates an event. A source registers listeners' notifications about events. Every source has different methods for registering the listeners. The general form is as follows:

```
public void addSTypeListener(STypeListener listener);
```

In this form, `SType` represents the event name. The method takes argument of `STypeListener` that represents reference to the event listener. It is actually an object of the class that implements any of the listener interfaces. For example, a method that registers mouse events is `addMouseListener`, a method that registers a Component event is `addComponentListerner`, etc. An event listener may be removed from an Event Source's list of interested Listeners by calling a `removeSTypeListener()` method, passing in the listener object to be removed.

One source can register more than one listener. Whenever an associated event occurs, all registered listeners are notified about the event—which is known as multicasting the event. In case the source is restricted to register just one listener, it is called unicasting the event.

15.6 DISCUSSION OF EVENT CLASSES

In this section, all the event classes, which have been discussed earlier, are discussed in detail. They are defined within the package `java.awt.event`.

1. The `ActionEvent` class

The class is used for handling events like pressing a button, clicking a menu item or list box item. For all the events just mentioned an `ActionEvent` is generated. The class defines the following int constants.

➤ `ACTION_PERFORMED`

This event id indicates that a meaningful action occurred.

- SHIFT_MASK
It is shift modifier: an indicator that the shift key was held down during the event
- CTRL_MASK
It is control modifier: an indicator that the control key was held down during the event.
- ALT_MASK
It is alt modifier; an indicator that the alt key was held down during the event.

The important method of this class are as follows:

- (i) `public int getModifiers()`
The method returns the modifier keys held down during this action event. It actually returns the bitwise-or of the modifier constants.
- (ii) `public String getActionCommand()`
The method returns the command string associated with this action. For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

2. The **AdjustmentEvent** class

The class represents adjustment events generated by adjustable objects like scroll bar. The class defines five integer constants that represent five types of adjustment events. They are listed below:

- BLOCK_DECREMENT
This constant represents an event when the user clicks inside the scroll bar to decrease its value.
- BLOCK_INCREMENT
This constant represents an event when the user clicks inside the scroll bar to increase its value.
- TRACK
This constant represents an event when the user drags the slider or thumb of the scroll bar.
- UNIT_DECREMENT
This constant represents an event when the user presses the down arrow button at the end of the scroll bar.
- UNIT_INCREMENT
This constant represents an event when the user presses the up arrow button at the end of the scroll bar.

Apart from five integer constants, the class defines one more—`ADJUSTMENT_VALUE_CHANGED` that represents adjustment value changed event.

The class defines the following methods:

- (i) `public Adjustable getAdjustable()`
This method returns the Adjustable object where this event originated.
- (ii) `public int getAdjustment Type()`
This method returns the type of adjustment which caused the value changed event. It will have one of the following values:
 - (a) UNIT_INCREMENT
 - (b) UNIT_DECREMENT
 - (c) BLOCK_INCREMENT
 - (d) BLOCK_DECREMENT
 - (e) TRACK
- (iii) `public int getValue()`
This method returns the current value in the adjustment event. For example, when any scroll bar event is generated, the current scroll bar value is returned.

3. The ComponentEvent class

This class represents a low-level event which indicates that a component moved, changed size or changed visibility. It is the root class for the other component-level events classes like KeyEvent, MouseEvent, WindowEvent, etc. The class defines four integer constants that represent four types of component events. They are listed below:

- COMPONENT_HIDDEN
This event indicates that the component was rendered invisible.
- COMPONENT_MOVED
This event indicates that the component's position changed.
- COMPONENT_RESIZED
This event indicates that the component's size changed.
- COMPONENT_SHOWN
This event indicates that the component was made visible.

The class defines one useful method:

```
public Component getComponent()
```

The method returns the originator of the event.

4. The Container

The class represents a low-level event which indicates that a container's contents changed because a component was added or removed. The two integers constant, COMPONENT_ADDED and COMPONENT_REMOVED, represent the two event types that indicate adding and removing a component to a container such as Panel. Container events are provided for notification purposes only; the AWT will automatically handle changes to the containers' contents internally so that the program works properly regardless of whether the program is receiving these events or not.

The class is the child class of ComponentEvent class. The class defines the two useful methods:

- public Container getContainer()
This method returns the originator of the event.
- public Component getChild()
This method returns the component that was affected by the event (i.e., added or removed).

5. The FocusEvent class

The class represents a low-level event which indicates that a Component has gained or lost the input focus. This low-level event is generated by a Component (such as a TextField). The event is passed to every FocusListener or FocusAdaptor object which was registered to receive such events using the Component's addFocusListener method. (FocusAdaptor object implements the FocusListener interface.) Each such listener object gets this FocusEvent when the event occurs.

There are two levels of focus events: permanent and temporary. Permanent focus change events occur when focus is directly moved from one Component to another, such as through a call to requestFocus() or as the user uses the TAB key to traverse Components. Temporary focus change events occur when focus is temporarily lost for a Component as the indirect result of another operation, such as Window deactivation or a Scrollbar drag. In this case, the original focus state will automatically be restored once the operation is finished, or, in the case of Window deactivation, when the Window is reactivated. Both permanent and temporary focus events are delivered using the FOCUS_GAINED and FOCUS_LOST event ids; the level may be distinguished in the event using the isTemporary() method.

The method has this signature:

```
public boolean isTemporary()
```

It defines the focus change event as temporary or permanent. It returns true if the focus change is temporary, false otherwise.

6. The InputEvent class

It is the root event class for all component-level input events. The class is the abstract class and MouseEvent and KeyEvent are its direct known subclasses. Input events are delivered to listeners before they are processed normally by the source where they originated. For example, consuming mousePressed events on a button component will prevent the button from being activated.

The class defines the following integer constants which can be used for getting information about the event.

| | | |
|-------------------|-------------------|-------------------|
| ALT_DOWN_MASK | BUTTON1_DOWN_MASK | BUTTON2_DOWN_MASK |
| BUTTON3_DOWN_MASK | CTRL_DOWN_MASK | SHIFT_DOWN_MASK |

All of the above constants designate pressing of certain keys when the event occurred.

The class defines the following methods that can be used to check whether the particular key was pressed when that event occurred. They are given below:

```
boolean isAltDown()
boolean isControlDown()
boolean isShiftDown()
```

Each of the method returns true when any of the modifier Alt key, Ctrl key or Shift key was down on this event.

Apart from these methods there is a method `getModifiersEx()` that returns all of the flags as shown above in the form of integer constants. Its signature is as shown below:

```
public int getModifierEx()
```

The method returns the extends modifier mask for this event. Extended modifiers represent the state of all modal keys, such as ALT, CTRL, SHIFT and the mouse buttons just after the event occurred.

7. The ItemEvent class

The class represents a semantic event (occurred on some GUI object) which indicates that an item was selected or deselected. This high-level event is generated by an `ItemSelected` object (such as a List, Combobox, Checkbox, etc.) when an item is selected or deselected by the user. The event is passed to every `ItemListener` object which registered to receive such events using the component's `addItemListener` method.

The class defines the following integer constants:

- `SELECTED`
This state-change value indicates that an item was selected.
- `DESELECTED`
This state-change value indicates that a selected item was deselected.
- `ITEM_STATE_CHANGED`
This event id indicates that an item's state changed.

Useful methods of this class are as follows:

- (a) `public Object getItem()`
This method returns the item affected by the event (i.e., generated the event).
- (b) `public int getStateChange()`
This method returns the type of state change (i.e., an integer) that indicates whether the item was selected or deselected.
- (c) `public ItemSelectable getItemSelectable()`
This method returns the `ItemSelectable` object that originated the event.

8. The KeyEvent class

The class represents an event which indicates that a keystroke occurred in a component. This low-level event is generated by a component object (such as text field) when a key is pressed, released

or typed. The event is passed to every KeyListener or keyAdapter object which registered to receive such events using the component's addKeyListener method (keyAdapter object implements the keyListener interface). Each such listener object gets this KeyEvent when the event occurs.

The class defines following integer constants that recognize the associated event.

- KEY_TYPED
This is a “key typed” event. This event is generated when a character is entered.
- KEY_PRESSED
This is a “key pressed” event. This event is generated when a key is pushed down.
- KEY_RELEASED
This is a “key released” event. This event is generated when a key is released.

The difference between “key press” and “key typed” is that the latter occurs only when a character is pressed and “key press” occurs for any key that is present on the keyboard.

Apart from the above constants, the class also defines virtual key codes. Virtual key codes are used to report which keyboard key has been pressed, rather than a character generated by a combination of one or more keystrokes (such as “A”, which comes from shift and “a”). For every key present on the keyboard, a virtual key code is defined that is identified by the following integer constants:

- (a) VK_0 to VK_9 for numbers 0 to 9; they are same as ASCII 0 to 9.
- (b) VK_A to VK_Z for alphabets A to Z; same as ASCII A to Z.
- (c) VK_F1 VK_F24 for function keys F1 to F24.
- (d) Some other virtual key codes are given in the following table.

| | | | |
|---------------|--------------|--------------|------------|
| VK_ALT | VK_AMPERSAND | VK) ASTERISK | VK_AT |
| VK_CIRCUMFLEX | VK_COLON | VK_COMMA | VK_CONTROL |
| VK_DOWN | VK_ENTER | VK_ESCAPE | VK_HOME |
| VK_LEFT | VK_NUM_LOCK | VK_PAGE_DOWN | VK_PAGE_UP |
| VK_CAPS_LOCK | VK_DELETE | VK_INSERT | VK_RIGHT |
| VK_SHIFT | VK_TAB | VK_UP | VK_WINDOWS |

The commonly used methods of this class are explained as follows:

- i. `public char getKeyChar()`
This method returns the character associated with the key in this event. For example, the KEY_TYPED event for shift + “a” returns the value for “A”. Actually, the method returns unicode character defined for this key event. If no valid unicode character exists for this key event, CHAR_UNDEFINED is returned.
- ii. `public int getKeyCode()`
This method returns the integer keyCode associated with the actual key on the keyboard in this event. For KEY_TYPED events, the keyCode is VK_UNDEFINED.
- iii. `public static String getKeyText(int keyCode)`
This method returns a String describing the keyCode, such as “HOME”, “F1” or “A”.

9. The MouseEvent class

This class represents an event which indicates that a mouse action occurred in a component. This event is used both for mouse events (click, enter, exit) and mouse motion events (mouse and drags).

The low-level event is generated by a component object for the following:

- Mouse Events
 - (a) A mouse button is pressed.
 - (b) A mouse button is released.

- (c) A mouse button is clicked (pressed and released).
 - (d) The mouse cursor enters the unobscured part of component's geometry.
 - (e) The mouse cursor exits the unobscured part of component's geometry.
- Mouse Motion Events
- (a) The mouse is moved.
 - (b) The mouse is dragged.

For recognizing all the above listed mouse events, the `MouseEvent` class defines the following integer constants. There are some other constants too that are not used for recognizing the mouse events but are used for some other purpose associated with the event.

| Constant | Description |
|-----------------------------|--|
| <code>MOUSE_CLICKED</code> | The "mouse clicked" event. This <code>MouseEvent</code> occurs when a mouse button is pressed and released. |
| <code>MOUSE_DRAGGED</code> | The "mouse dragged" event. This <code>MouseEvent</code> occurs when the mouse position changes while a mouse button is pressed. |
| <code>MOUSE_ENTERED</code> | The "mouse entered" event. This <code>MouseEvent</code> occurs when the mouse cursor enters the unobscured part of component's geometry. |
| <code>MOUSE_EXITED</code> | The "mouse exited" event. This <code>MouseEvent</code> occurs when the mouse cursor exits the unobscured part of component's geometry. |
| <code>MOUSE_MOVED</code> | The "mouse moved" event. This <code>MouseEvent</code> occurs when the mouse position changes. |
| <code>MOUSE_PRESSED</code> | The "mouse pressed" event. This <code>MouseEvent</code> occurs when a mouse button is pushed down. |
| <code>MOUSE_RELEASED</code> | The "mouse released" event. This is the only <code>MouseEvent</code> that occurs when a mouse button is let up. |
| <code>MOUSE_WHEEL</code> | The "mouse wheel" event. This is the only <code>MouseWheelEvent</code> . It occurs when a mouse equipped with a wheel has its wheel rotated. |
| <code>BUTTON1</code> | Indicates mouse button #1 |
| <code>BUTTON2</code> | Indicates mouse button #2 |
| <code>BUTTON3</code> | Indicates mouse button #3 |

Table 15.2 Constants defined by `MouseEvent` class

Some useful methods of this class are discussed as follows:

- (i) `public int getButton()`
This method returns one of the following constants: `NOBUTTON`, `BUTTON1`, `BUTTON2` and `BUTTON3` to indicate which mouse button has changed state.
- (ii) `public int getClickCount()`
This method returns the number of mouse clicks (an integer value) associated with this event.
- (iii) `public Point getPoint()`
This method returns a point object containing the x and y coordinates relative to the source component.
- (iv) `public int getX()`
This method returns the horizontal x position of the event where mouse event occurred.

(v) public void translatePoint(int x, int y)

This method translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets. Parameters x and y are the horizontal x value and vertical y value to add to the current x and y coordinate position, respectively.

10. The **TextEvent** class

This class represents a semantic event which indicates that an object's text changed. This high-level event is generated by an object (such as a `TextComponent`) when its text changes. The event is passed to every `TextListener` object which is registered to receive such events using the component's `addTextListener` method.

This class defines one final integer constant `TEXT_VALUE_CHANGED` that indicates that object's text changed. No such method of this class is of interest.

11. The **WindowEvent** class

This class represents a low-level event that indicates that a window has changed its status. This low-level event is generated by a `Window` object when it is opened, closed, activated, deactivated, iconified or deiconified, or when focus is transferred into or out of the `Window`.

The event is passed to every `WindowListener` or `WindowAdapter` object which registered to receive such events using the `window's addWindowListener` method. (`WindowAdapter` objects implement the `WindowListener` interface.) Each such listener object gets this `WindowEvent` when the event occurs.

The various constants for recognizing the window events are listed in Table 15.3.

| Constant | Description |
|---------------------------------|--|
| <code>WINDOW_ACTIVATED</code> | The window-activated event type. This event is delivered when the <code>Window</code> becomes the active <code>Window</code> . Only a <code>Frame</code> or a <code>Dialog</code> can be the active <code>Window</code> . |
| <code>WINDOW_CLOSED</code> | The window closed event. This event is delivered after the window has been closed as the result of a call to <code>dispose</code> . |
| <code>WINDOW_CLOSING</code> | The “window is closing” event. This event is delivered when the user attempts to close the window from the window’s system menu. |
| <code>WINDOW_DEACTIVATED</code> | The window-deactivated event type. This event is delivered when the <code>Window</code> is no longer the active <code>Window</code> . Only a <code>Frame</code> or a <code>Dialog</code> can be the active <code>Window</code> . |
| <code>WINDOW_DEICONIFIED</code> | The window deiconified event type. This event is delivered when the window has been changed from a minimized to a normal state. |
| <code>WINDOW_ICONIFIED</code> | The window iconified event type. This event is delivered when the window has been changed from a normal to a minimized state. |
| <code>WINDOW_OPENED</code> | The window opened event. This event is delivered only the first time a window is made visible. |

Table 15.3 Constants defined by `WindowEvent` class

The most commonly used method of this class is the `getWindow()` whose signature is given below:

```
public Window getWindow()
```

The method returns the `Window` object that originated the event.

15.7 THE LISTENERS INTERFACES

After discussing the Event classes it is time now to discuss the key interface listeners. Listeners are nothing but instances of classes that implement listeners interface defined by `java.awt.event` package. The `Listeners` classes implement the interfaces by providing the body for the methods. All listener interfaces have their `EventListener` as their super interface. All the listener interfaces and their respective methods are discussed below:

1. The `Actionlistener` interface

It is the listener interface for receiving action event. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that objects' `actionPerformed` method is invoked.

Source of event for this listener is objects of `Button` class. The interface has just one method.

```
void actionPerformed(ActionEvent e)
```

The method is invoked when an action occurs (i.e., when a button is pressed).

2. The `Adjustmentlistener` interface

It is the listener interface for receiving adjustment events. The interface defines just one method:

```
void adjustmentValueChanged(AdjustmentEvent e)
```

Source of event for this listener is object's `Scrollbar` class. The method is invoked when the value of the adjustable has changed.

3. The `ComponentListener` interface

It is the listener interface for receiving component events. The class that is interested in processing a component event either implements this interface (and all the methods it contains) or extends the abstract `ComponentAdapter` class (overriding only the methods of interest). The listener object created from that class is then registered with a component using the component's `addComponentListener` method. When the component's size, location or visibility changes, the relevant method in the listener object is invoked, and the `ComponentEvent` is passed to it.

Component events are provided for notification purposes only. The AWT will automatically handle component moves and resize internally so that GUI layout works properly regardless of whether a program registers a `ComponentListener` or not.

The interface has following four methods:

- `void componentHidden(ComponentEvent e)`
This method is invoked when the component has been made invisible.
- `void componentMoved(ComponentEvent e)`
This method is invoked when the component's position changes.
- `void componentResized(ComponentEvent e)`
This method is invoked when the component size changes.
- `void componentShown(ComponentEvent e)`
This method is invoked when the component has been made visible.

4. The `ContainerListener` interface

It is the listener interface for receiving container events. The class that is interested in processing a container event either implements this interface (and all the methods it contains) or extends the abstract `ContainerAdapter` class (overriding only the method of interest). The listener object created from that class is then registered with a component's `addContainerListener` method. When the container's content changes because a component has been added or removed, the relevant method in the listener object is invoked, and the `ContainerEvent` is passed to it.

The interface has the following methods:

- `void componentAdded(ContainerEvent e)`
This method is invoked when a component has been added to the container.
- `void componentRemoved(ContainerEvent e)`
This method is invoked when a component has been removed from the container.

5. The FocusListener interface

It is the listener interface for receiving keyboard focus events on a component. The class that is interested in processing a focus event either implements this interface (and all the methods it contains) or extends the abstract `FocusAdaptor` class (overriding only the methods of interest). The listener object created from that class is then registered with a component using the component's `addFocusListener` method. When the component gains or loses keyboard focus, the relevant method in the listener object is invoked, and the `FocusEvent` is passed to it.

The interface has two methods:

- `void focusGained(FocusEvent e)`
This method is invoked when a component gains the keyboard focus.
- `void focusLost(FocusEvent e)`
This method is invoked when a component loses the keyboard focus.

6. The ItemListener interface

It is the listener interface for receiving item events. The class that is interested in processing an item event implements this interface. The object created with that class is then registered with a component using the component's `addItemListener` method. When an item-selection event occurs, the listener object's `itemStateChanged` method is invoked. The signature of the method is shown below:

```
void itemStateChanged(ItemEvent e)
```

This method is invoked when an item has been selected or deselected by the user. The code written for this method performs the operations that need to occur when an item is selected (or deselected).

7. The KeyListener interface

It is the listener interface for receiving keyboard events (keystrokes). The class that is interested or processing a keyboard event either implements this interface (and all the methods it contains) or extends the abstract `keyAdaptor` class (overriding only the methods of interest).

The listener object created from that class is then registered with a component using the component's `addKeyListener` method. A keyboard event is generated when a key is pressed, released or typed. The relevant method in the listener object is then invoked, and the `KeyEvent` is passed to it.

The interface declares the following methods:

- `void keyTyped(KeyEvent e)`
This method is invoked when a key has been typed.
- `void keyPressed(KeyEvent e)`
This method is invoked when a key has been pressed.
- `void keyReleased(KeyEvent e)`
This method is invoked when a key has been released.

For more about these events refer discussion of `KeyEvent` class.

8. The MouseListener interface

It is the listener interface for receiving “interesting” mouse events (press, release, click, enter and exit) on a component. The class that is interested in processing a mouse event either implements this

interface (and all the methods it contains) or extends the abstract `MouseAdapter` class (overriding only the methods of interest). The interface declares the following methods:

- `void mouseClicked(MouseEvent e)`
This method is invoked when the mouse button has been clicked (pressed and released) on a component.
- `void mousePressed(MouseEvent e)`
This method is invoked when a mouse button has been pressed on a component.
- `void mouseReleased(MouseEvent e)`
This method is invoked when a mouse button has been released on a component.
- `void mouseEntered(MouseEvent e)`
This method is invoked when the mouse enters a component.
- `void mouseExited(MouseEvent e)`
This method is invoked when the mouse exits a component.

9. The `MouseMotionListener` interface

It is the listener interface for receiving mouse motion events on a component. A mouse motion event is generated when the mouse is moved or dragged. The class that is interested in processing a mouse motion event either implements this interface (and all the methods it contains) or extends the abstract `MouseMotionAdapter` class (overrides only the methods of interest).

The interface declares the following methods:

- `void MouseDragged(MouseEvent e)`
This method is invoked when a mouse button is pressed on a component and then dragged. `MOUSE_DRAGGED` events will continue to be delivered to the component where the drag originated until the mouse button is released.
- `void mouseMoved(MouseEvent e)`
This method is invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

10. The `TextListener` interface

It is the listener interface for receiving text event. The class that is interested in processing a text event implements this interface. The object created with that class is then registered with a component using the component's `addTextListener` method. When the component's text changes, the listener object's `textValueChanged` method is invoked.

The signature of the method is given below:

```
void textValueChanged(TextEvent e)
```

11. The `WindowListener` interface

It is the listener interface for receiving window events. The class that is interested in processing a window event either implements this interface (and all the methods it contains) or extends the abstract `WindowAdapter` class (overriding only the methods of interest). The listener object created from that class is then registered with a Window using the window's `addWindowListener` method. When the window's status changes by virtue of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the `WindowEvent` is passed to it.

The interface declares the following methods for recognizing various window-related events.

- `void windowDeactivated(WindowEvent e)`
This method is invoked when a Window is no longer the active Window. The active Window is always either the focused Window, or the first Frame or Dialog that is an owner of the focused Window.

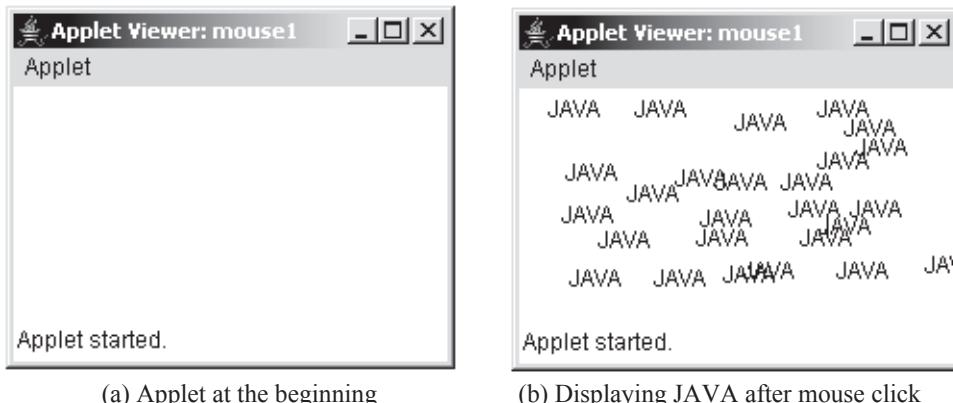
- `void windowClosed(WindowEvent e)`
This method is invoked when a window has been closed as the result of calling dispose on the window.
- `void windowClosing(WindowEvent e)`
This method is invoked when the user attempts to close the window from the window's system menu.
- `void windowActivated(WindowEvent e)`
This method is invoked when the Window is set to be the active Window.
- `void windowDeiconified(WindowEvent e)`
This method is invoked when a window is changed from a minimized to a normal state.
- `void windowIconified(WindowEvent e)`
This method is invoked when a window is changed from a normal to minimized state.
- `void windowOpened(WindowEvent e)`
This method is invoked the first time a window is made visible.

15.8 PROGRAMMING EXAMPLE

15.8.1 Handling Mouse Events

```
/*PROG 15.1 DISPLAYING "JAVA" ON MOUSE CLICK VER 1 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="mouse1" width =350 height = 120>
</applet>
</html>
*/
public class mouse1 extends Applet implements MouseListener
{
    public void init()
    {
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent me)
    {
        Graphics g = getGraphics();
        g.drawString("JAVA", me.getX(), me.getY());
    }
    public void mouseExited(MouseEvent me) { }
    public void mousePressed(MouseEvent me) { }
    public void mouseReleased(MouseEvent me) { }
    public void mouseEntered(MouseEvent me) { }
}
OUTPUT:
```

**Figure 15.1** Output screen of Program 15.1

Explanation: In this program, the class `mouse1` implements `MouseListener` interface. In the program the `mouseClicked` method has only been dealt with so code is written only inside this method. But as it is necessary to provide implementation of all the methods of interface no code is written inside the braces of all the other methods like `mouseExited`, `mousePressed`, etc. The listener is added using the method `addMouseListener` that is provided by the `Component` class. This method can be used directly because the `Component` class is the super class of `Applet` class. The method takes a reference of the `MouseListener` interface. Its signature is as shown below:

```
public void addMouseListener(MouseListener ml)
```

Here, `ml` is a reference to the object receiving mouse events, as this represents current object of the class that has been used as argument to this method.

In the program, the source and listener object as events are generated and listened by the applet. In the programs where GUI elements are used, sources will be `Button`, `List`, `Checkbox`, etc. They will be introduced in the next chapter. The code in the `init` method can be written as follows:

```
MouseListener ml;
addMouseListener(ml);
```

Whenever mouse is clicked in the applet window, `mouseClicked` event is generated and listener can listen this event. In response to the event `mouseClicked`, method is called. In the method, current graphics context is obtained by using `getGraphics` method. The point where mouse was clicked is taken using `getX` and `getY` methods, then using `drawString` on the current coordinates “JAVA” is displayed.

```
/*PROG 15.2 DISPLAYING "JAVA" ON MOUCE CLIKC VER 2 */
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="mouse2" width = 350 height = 120>
</applet>
</html>
```

```

*/
public class mouse2 extends Applet implements MouseListener
{
    String msg = " ";
    int mx = 0, my = 0;
    static int i = 0;
    public void init()
    {
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent me)
    {
        Color[]col ={Color.red,Color.green, Color.blue,
                     Color.pink,Color.magenta,
                     Color.yellow};
        mx = me.getX();
        my = me.getY();
        Graphics g = getGraphics();
        g.setColor(col[i%6]);
        i++;
        g.drawString("JAVA",mx,my);
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, mx, my);
    }
    public void mouseExited(MouseEvent me) { }
    public void mousePressed(MouseEvent me) { }
    public void mouseReleased(MouseEvent me) { }
    public void mouseEntered(MouseEvent me) { }
}

```

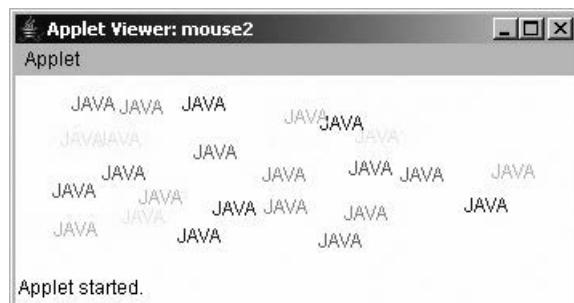
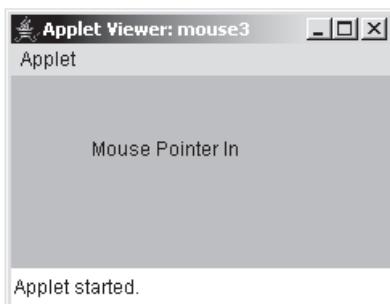


Figure 15.2 Output screen of Program 15.2

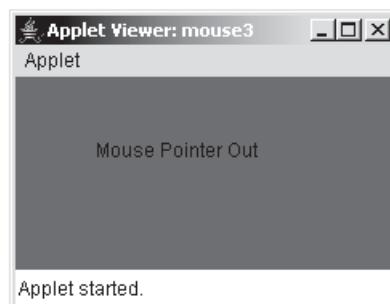
Explanation: The program is similar to the previous one but here “JAVA” has been displayed on current mouse coordinate in colors. For this reason, an array of colors of `Color` class has been created. Whenever the mouse click event occurs, `mouseClicked` method is called. In the method, the new foreground color is set by method `setForeground`. Due to `i%6`, different indexes between 0 to 5 are generated and entries between `col[0]` and `col[5]` are selected.

```
/*PROG 15.3 DEMO OF MOUSE_ENTERED AND EXITED EVENT */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="mouse3" width =350 height = 120>
</applet>
</html>
*/
public class mouse3 extends Applet implements MouseListener
{
    String msg = " ";
    public void init()
    {
        addMouseListener(this);
    }
    public void mouseExited(MouseEvent me)
    {
        setBackground(Color.red);
        msg = "Mouse Pointer Out";
    }
    public void mouseEntered(MouseEvent me)
    {
        setBackground(Color.green);
        msg = "Mouse Pointer In";
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, 50, 50);
    }
    public void mousePressed(MouseEvent me) { }
    public void mouseReleased(MouseEvent me) { }
    public void mouseClicked(MouseEvent me) { }
}
```



(a) When mouse pointer is in



(b) When mouse pointer is out

Figure 15.3 Output screen of Program 15.3

Explanation: The program demonstrates `mouseEntered` and `mouseExited` methods. The program is simple: When the mouse pointer is inside the applet, `mouseEntered` method is called. This causes the `paint` method on the background of the applet window because, `mouseExited` method is invoked. This causes the `paint` method to paint the background of the applet window in red color and display string "Mouse Pointer Out". Note that the `paint` method is called by default after both the events.

```
/*PROG 15.4 BEHAVIOUR OF METHODS OF MOUSELISTENER INTERFACE */
```

```
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
/*
<html>
<applet code="mouse4" width =350 height = 120>
</applet>
</html>
*/
public class mouse4 extends Applet implements MouseListener
{
    String msg = " ";
    int mx, my;
    public void init(){
        addMouseListener(this);
    }
    public void mousePressed(MouseEvent me)
    {
        setForeground(Color.red);
        msg = "Mouse Pressed";
        mx = me.getX();
        my = me.getY();
    }
    public void mouseReleased(MouseEvent me)
    {
        setForeground(Color.green);
        msg = "Mouse Released ";
        mx = me.getX();
        my = me.getY();
    }
    public void paint(Graphics g) {
        g.drawString(msg, mx, my);
    }
    public void mouseClicked(MouseEvent me)
    {
        //msg = "Mouse Clicked";
    }
    public void mouseEntered(MouseEvent me) { }
    public void mouseExited(MouseEvent me) { }
}
```



Figure 15.4 Output screen of Program 15.4

Explanation: The program is simple to understand. The code is written over `mousePressed` and `mouseReleased` methods. One important thing to note here is that `mousePressed` is called first, then `mouseClicked`, and when both are implemented, `mouseReleased` does not work.

```
/*PROG 15.5 DRAWING RANDOM LINES, KEEPING ONE POINT FIX */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="mouse5" width =350 height = 120>
</applet>
</html>
*/
public class mouse5 extends Applet implements
MouseMotionListener
{
    int x, y, w, h;
    public void init(){
        addMouseMotionListener(this);
        h = this.getSize().height;
        w = this.getSize().width;
    }
    public void mouseDragged(MouseEvent me)
    {
    }
    public void mouseMoved(MouseEvent me){
        x = (int)(Math.random() * 10000);
        y = (int)(Math.random() * 10000);
        Graphics g = getGraphics();
        g.setColor(new Color(x%255, y%255, (x * h)%255));
        g.drawLine(w / 2, h / 2, x % w, y % h);
    }
}
```

Explanation: In this program, the MouseMotion Listener interface is implemented. Here, only the mouseMoved event has been dealt with so any coding has not been given in the mouseDragged method. The listener is added using the addMouseMotion Listener which is provided by the Component class as explained with the first program. The signature of the method is given below:

```
public void addMouseMotionListener(MouseMotionListener mm1)
```

The method takes a reference to the object receiving mouse motion events.

Applet window's width and height are obtained using `this.getSize().width` and `this.getSize().height`. Actually, the `getSize()` method returns a reference of Dimension class, and height and width are its two members. So the code can also be written as:

```
Dimension D = this.getSize();
w = D.width;
h = D.height;
```

When the mouse is moved `mouseMoved` method is called. This is how the listener is registered. In this method, two random integer numbers between 0 and 10,000 are returned in `x` and `y`. Using these random integers, random colors are generated. While drawing lines of random length, one point is kept fixed at the centre of the applet window. The end is anywhere between the height and width of the applet window.

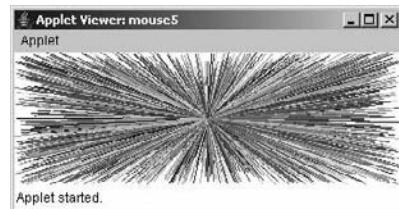


Figure 15.5 Output screen of Program 15.5

```
/*PROG 15.6 FREEHAND DRAWING USING MOUSE */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="mouse6" width =350 height = 120>
</applet>
</html>
*/
public class mouse6 extends Applet implements MouseListener,
MouseMotionListener
{
    String msg = " ";
    int px, py, cy, cx;
    boolean press = true;
    public void init()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
        cx = cy = px = py = 0;
    }
    public void mousePressed(MouseEvent me)
    {
        if (press == true)

```

```

        {
            px = me.getX();
            py = me.getY();
            press = false;
        }
    }
    public void mouseReleased(MouseEvent me)
    {
        press = true;
    }
    public void mouseDragged(MouseEvent me)
    {
        cx = me.getX();
        cy = me.getY();
        Graphics g = getGraphics();
        g.drawLine(px, py, cx, cy);
        px = cx;
        py = cy;
    }
    public void mouseClicked(MouseEvent me)
    {
    }
    public void mouseEntered(MouseEvent me) { }
    public void mouseExited(MouseEvent me) { }
    public void mouseMoved(MouseEvent me) { }
}

```

Explanation: The program demonstrates how freehand drawing can be performed using mouse events. The logic is simple. Think of a line as a series of connected points so close to each other that the line appears smooth. One boolean variable **press** is taken. The initial value of this is true. When the **mouseDragged** method is called in response to mouse-dragged event, **press** will be true. The current mouse coordinate is saved in the **px** and **py** coordinate and method returns due to if condition. The value of **press** becomes false here. It is necessary that till the mouse is dragged, initial point is to be saved only once. Next time when the method is called again, new points are saved in **cx** and **cy**. Now a line is drawn from point (**px**, **py**) to (**cx**, **cy**). After drawing the line, the current coordinates become the previous ones as they are saved to **px** and **py**. Next time again when method is called, line is drawn and **cx** and **cy** are saved to **px** and **py**.

When the mouse is released, **press** becomes true again.

15.8.2 Handling Keyboard Events

```
/*PROG 15.7 IMPLEMENTING KEYLISTENER INTERFACE VER 1 */
```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

```



Figure 15.6 Output screen of Program 15.6

```

/*
<html>
<applet code="keypress1" width =350 height = 120>
</applet>
</html>
*/
public class keypress1 extends Applet implements KeyListener
{
    String msg = " ";
    public void init()
    {
        addKeyListener(this);
    }
    public void keyPressed(KeyEvent ke)
    {
        msg = "Key is down";
        repaint();
    }
    public void keyReleased(KeyEvent ke)
    {
        msg = "Key is UP";
        repaint();
    }
    public void keyTyped(KeyEvent ke) { }
    public void paint(Graphics g)
    {
        g.drawString(msg, 20, 20);
    }
}

```

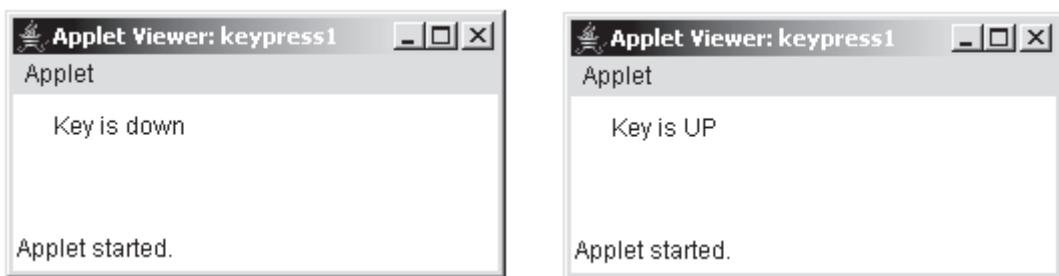


Figure 15.7 Output screen of Program 15.7

Explanation: In this program, KEY_PRESSED and KEY_RELEASED events are demonstrated. Similar to implementing MouseListener and MouseMotionListener interface, KeyListener interface has to be implemented. We also add a listener using addKeyListener method. The signature of the method is as shown:

```
public void addKeyListener(KeyListener l);
```

The method adds the specified key listener to receive key events from this component.

When a KEY_PRESSED event occurs, keyPressed method is called and when KEY_RELEASED event occurs, keyReleased method is called. When any key is pressed, the method keyPressed executes

and msg contains "Key is down". The method repaint() calls the paint method and "Key is down" is displayed at (20, 20). When the key is released, the method keyReleased executes and msg contains "Key is UP". The method repaint() calls the paint method and "Key is UP" is displayed at (20, 20).

```
/*PROG 15.8 IMPLEMENTING KEYLISTENER INTERFACE VER 2*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="keypress2" width =350 height = 120>
</applet>
</html>
*/
public class keypress2 extends Applet implements KeyListener
{
    String msg = " ";
    public void init()
    {
        addKeyListener(this);
    }
    public void keyPressed(KeyEvent ke) { }
    public void keyReleased(KeyEvent ke) { }
    public void keyTyped(KeyEvent ke)
    {
        char ch = ke.getKeyChar();
        msg = msg + ch;
        repaint();
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, 10, 20);
    }
}
```

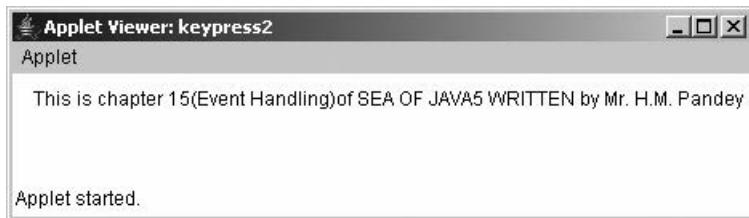


Figure 15.8 Output screen of Program 15.8

Explanation: In this program, the KEY_TYPED event is used. The event is used when the character from the generated key is required. When KEY_TYPED event occurs, keyTyped method is called. In this method, the character associated with the key typed using getKeyChar method is obtained. The character is concatenated with the msg string. The resultant string thus obtained is displayed in the paint method. The method is invoked in response to repaint() method.

```
/*PROG 15.9 IMPLEMENTING KEYLISTENER INTERFACE VER 3*/
```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class keypress3 extends Applet implements KeyListener
{
    String msg = " ";
    int x, y, px, py;
    public void init()
    {
        addKeyListener(this);
        px = 30; py = 50;
        x = px;
        y = py;
    }
    public void keyPressed(KeyEvent ke)
    {
        int code = ke.getKeyCode();
        switch (code)
        {
            case KeyEvent.VK_LEFT:
                x--;
                break;
            case KeyEvent.VK_RIGHT:
                x++;
                break;
            case KeyEvent.VK_UP:
                y--;
                break;
            case KeyEvent.VK_DOWN:
                y++;
                break;
        }
        Graphics g = getGraphics();
        g.drawLine(px, py, x, y);
        px = x;
        py = y;
    }
    public void keyReleased(KeyEvent ke) { }
    public void keyTyped(KeyEvent ke) { }
}

```

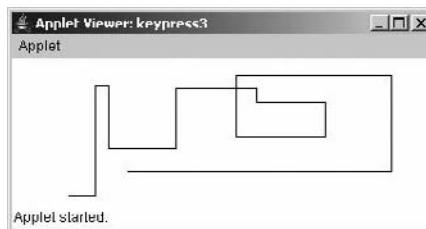


Figure 15.9 Output screen of Program 15.9

Explanation: In this program, the virtual key codes are used for the arrow keys. The key code is taken using the method `getKeyCode` and stored in the code variable. In the program, lines are needed to be drawn using arrow keys. For this purpose, the starting coordinate chosen is 30 and 50 and stored in `px` and `py`. The same is stored in `x` and `y` also. Now depending on which arrow key is pressed, `x` and `y` are incremented/decremented accordingly. For example, when left arrow key is pressed, `x` is decremented by 1 and when up arrow key is pressed, `y` is decremented by 1. Outside the switch block, line is drawn from point (`px`, `py`) to (`x`, `y`) and old points are updated with the values of `x` and `y`.

15.9 ADAPTER CLASSES

Some of the listener interface, such as `MouseListener` and `ComponentListener`, include a lot of methods. The rules for interfaces say that when a class implements an interface, it must include a definition for each method declared in the interface. For example, if `empty` is included only using the `mousePressed` method of the `MouseListener` interface, one ends up including empty definitions for `mouseClicked()`, `mouseReleased()`, `mouseEntered()` and `mouseExited()`. If a specially created nested class is used to handle events, there is a way to avoid this. As a convenience, the package `java.awt.event` includes several adapter classes, such as `MouseAdapter` and `ComponentAdapter`. An adapter class is a class that provides an empty implementation of all methods in an event listener interface. Adapter classes can be used when one is not interested in all of the methods of the interface, except one or two. The `MouseAdapter` class is a trivial class that implements the `MouseListener` interface by defining each of the methods in that interface to be empty. To make the programmer's mouse listener classes, one can extend the `MouseAdapter` class and override just those methods of interest. `ComponentAdapter` and other adapter classes work in the same way.

All of the Adapter classes are defined within the `java.awt.event` package. They are listed in Table 15.4 along with the related listener interface.

| Adapter Class | Listener Interface |
|---------------------------------|----------------------------------|
| <code>ComponentAdapter</code> | <code>ComponentListener</code> |
| <code>ContainerAdapter</code> | <code>ContainerListener</code> |
| <code>KeyAdapter</code> | <code>KeyListener</code> |
| <code>MouseAdapter</code> | <code>MouseListener</code> |
| <code>MouseMotionAdapter</code> | <code>MouseMotionListener</code> |
| <code>WindowAdapter</code> | <code>WindowListener</code> |

Table 15.4 Adapter classes and counterpart listeners

The use of Adapter classes is explained with the help of a small program. The “Java” program is written (i.e., wherever in the applet window mouse is clicked then “Java” will be displayed at that point).

```
/*PROG 15.10 DEMO OF MOUSEADAPTER CLASS */
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
```

```

<applet code="Ademo" width =350 height = 120>
</applet>
</html>
*/
public class Ademo extends Applet
{
    public void init()
    {
        MyMouseAdapter mma = new MyMouseAdapter(this);
        addMouseListener(mma);
    }
}
class MyMouseAdapter extends MouseAdapter
{
    Ademo m;
    public MyMouseAdapter(Ademo ad)
    {
        m = ad;
    }
    public void mouseClicked(MouseEvent me)
    {
        Graphics g = m.getGraphics();
        g.drawString("Java", me.getX(), me.getY());
    }
}

```

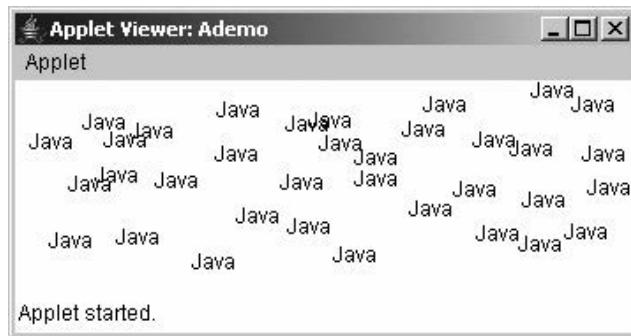


Figure 15.10 Output screen of Program 15.10

Explanation: In this program, a new class `MyMouseAdapter` is created that extends `MouseAdapter` class. In this class, the default empty implementation of `mouseClicked` method is overridden. In the main applet window class `Ademo`, an instance of `MyMouseAdapter` class is implemented and passed as argument to `addMouseListener` method. Note while creating an instance of `MyMouseAdapter` class reference of the current object of `Ademo` class is passed. For this reason, a reference `m` of `Ademo` class is created as a member of `MyMouseAdapter` class. This is necessary as the passed instance can be used for calling any of the method of `Component` or `Applet` class. Using this reference, the method `getGraphics` is called for getting a reference to `Graphics` class. The rest is simple to understand.

As another example `keyTyped` is written using `KeyAdapter` class.

```
/*PROG 15.11 DEMO OF KEYADAPTER CLASS */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<html>
<applet code="Adapter1" width =350 height = 120>
</applet>
</html>
*/
public class Adapter1 extends Applet
{
    String msg = " ";
    public void init()
    {
        addKeyListener(new MyKeyAdapter(this));
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, 10, 20);
    }
}
class MyKeyAdapter extends KeyAdapter
{
    Adapter1 mka;
    MyKeyAdapter(Adapter1 m)
    {
        mka = m;
    }
    public void keyTyped(KeyEvent ke)
    {
        char ch = ke.getKeyChar();
        mka.msg = mka.msg + ch;
        mka.repaint();
    }
}
```



Figure 15.11 Output screen of Program 15.11

Explanation: This program is simple to understand on the basis of explanation of the previous one.

15.10 NESTED AND INNER CLASSES

It is possible to place a class definition within another class definition. This is called nested class. The nested class is a valuable feature because it allows the programmer to group classes that logically belong together and to control the visibility of one within the other. The scope of nested class is limited to the scope of the class in which it is nested. For example, if class In is defined inside the class Out, In will be known to Out class only, but the reverse is not true. Class In can access all the members of the Out class directly regardless of their type: be it private, protected or any other, but class Out cannot access any of the member of the In class.

There are two types of nested classes: static nested class and non-static nested class.

A static nested class is the one that has static keyword written before class definition. A static class can access all the members of its enclosing class only through objects. It cannot access them directly. In general, static nested class is called simply the nested class.

A non-static nested class is the one where static keyword is not present before class definition. The most important type of nested class is the inner class. It can access all the members of its enclosing class without using an object. This being the reason, it is frequently used. In general, non-static nested class is called the inner class. Fields and methods in ordinary inner classes can only be at the outer level of a class, so ordinary inner classes cannot have static data, static fields or nested classes. Inner classes are covered in this section and the static nested classes in the next.

As an elementary example of inner class consider the program given below:

```
/*PROG 15.12 DEMO OF INNER CLASS VER1 */

class demo
{
    void show()
    {
        demo1 d1 = new demo1();
        d1.display();
    }
    class demo1
    {
        void display()
        {
            System.out.println("Hello from display of
                               inner class");
        }
    }//inner class ends
}//outer class ends
class JPS1
{
    public static void main(String[] args)
    {
        demo d = new demo();
        d.show();
    }
}
OUTPUT:
Hello from display of inner class
```

Here, class `demo1` is defined within the class `demo`, so it is an example of nested inner class. The class `demo1` has only one function `display`. The outer class `demo` defines one method named `show` in which an object of inner class `demo` is created through which the object `display` method is called. Note the reverse will not be true; one cannot call `show` method through an object of `demo` class. In the `main`, an object of outer class `demo` is created and `show` method is called which in turn calls the `display` method of inner class `demo1` using `d1`.

Take one more example where `temp` method is accessed from inner class `demo`. The private members are also introduced to in the outer class `demo`.

```
/*PROG 15.13 DEMO OF NESTED INNER CLASS VER2 */

class demo
{
    private int dx = 20;
    private String str = "Outer Class";
    void show()
    {
        demo1 d1 = new demo1();
        d1.display();
    }
    void temp()
    {
        System.out.println("HELLO FROM TEMP OF OUTER CLASS");
    }
    class demo1
    {
        void display()
        {
            System.out.println("\nInside display of
                               inner class");
            System.out.println("dx = " + dx);
            System.out.println("str = " + str);
            temp();
        }
    }//inner class ends
}//outer class ends
class JPS2
{
    public static void main(String[] args)
    {
        demo d = new demo();
        d.show();
    }
}
OUTPUT:
Inside display of inner class
dx = 20
str = Outer Class
HELLO FROM TEMP OF OUTER CLASS
```

The outer class demo has two private members and two methods: show and temp. Inside show, an object of inner class demo1 is created and the display method of inner class is called. In the display method of inner class, the private members dx and str of demo class are accessed. The temp function of demo class is also accessed. Thus, the inner class has full access to all the members of its enclosing class directly. Put in different perspective, treat inner class as any other data member or method of the outer class, then understanding inner class will not be a problem.

Next, consider an example where the members of the inner class is to be accessed in the outer class. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

```
/*PROG 15.14 DEMO OF NESTED AND INNER CLASS VER 3 */

class demo
{
    private String str_out = "OUTER CLASS";
    void show()
    {
        demo1 d1 = new demo1();
        d1.display();
    }
    class demo1
    {
        private String str_in = "INNER CLASS";
        void display()
        {
            System.out.println("\nInside display of
                               inner class");
            System.out.println("str_in = " + str_in);
            System.out.println("str_out = " + str_out);
        }
    }//inner class ends
    void temp()
    {
        System.out.println("\nInside temp of outer
                           class");
        System.out.println("str_in = " + str_in);
        System.out.println("str_out = " + str_out);
    }
}//outer class ends
class JPS3
{
    public static void main(String[] args)
    {
        demo d = new demo();
        d.show();
        d.temp();
    }
}
```

OUTPUT:

```
C:\JPS\ch15>javac JPS3.java
JPS3.java:23: cannot find symbol
symbol  : variable str_in
location: class demo
        System.out.println("str_in = " +str_in);
                                         ^
1 error
```

Clearly, the data member `str_in` of inner class is only within the scope of inner class `demo1`. It can only be used within the class `demo1`, but not outside (i.e., not within its enclosing class that is `demo`). So the compilation error is flashed.

The next example demonstrates how a method can be created in the outer class which returns a reference to inner class. Using this reference in the main, members of inner class can be accessed.

```
/*PROG 15.15 DEMO OF NESTED AND INNER CLASS VER 4 */

class demo
{
    private String str_out = "OUTER CLASS";
    class demo1
    {
        private String str_in = "INNER CLASS";
        void display()
        {
            System.out.println("\n Inside display of
                               inner class");
            System.out.println(" str_in = " + str_in);
            System.out.println(" str_out = " + str_out);
        }
    }//inner class ends
    demo1 getref()
    {
        return new demo1();
    }
}//outer class ends
class JPS4
{
    public static void main(String[] args)
    {
        demo d = new demo();
        demo.demo1 d1 = d.getref();
        d1.display();
    }
}
OUTPUT:
Inside display of inner class
str_in = INNER CLASS
str_out = OUTER CLASS
```

Here, the class `demo1` is inner class once again. The outer class `demo` is having one method `getref` which is returning a reference of inner class. This is perfectly valid. In the `main`, an object `d` of outer class `demo` is created. Note how a reference `d1` of inner class `demo1` has been created:

```
demo.demo1 d1;
```

As `demo1` is the inner class for `demo` class, it can be written using dot operator. Now as method `getref` is returning a reference of `demo1` class, that method is called using object `d` of `demo` class and assigned it to `d1` as:

```
demo.demo1 d1 = d.getref();
```

As it has a valid object of inner class `demo1` represented by `d1`, the method `display` of inner class can be called. The concept to build here is that:

“If you want to make an object of the inner class anywhere except from within a non-static method of the outer class, you must specify the type of that object as `OuterClassName.InnerClassName`, as seen in `main()`.”

However, there is another way of getting the reference of the inner class with slightest of variation from the previous program.

```
/*PROG 15.16 DEMO OF NESTED AND INNER CLASS VER 5 */

class demo
{
    private String str_out = "OUTER CLASS";
    class demo1
    {
        private String str_in = "INNER CLASS";
        void display()
        {
            System.out.println("\n Inside display of
                               inner class");
            System.out.println(" str_in = " + str_in);
            System.out.println(" str_out = " + str_out);
        }
    } //inner class ends
} //outer class ends
class JPS5
{
    public static void main(String[] args)
    {
        demo d = new demo();
        demo.demo1 d1 = d.new demo1();
        d1.display();
    }
}
OUTPUT:
Inside display of inner class
str_in = INNER CLASS
str_out = OUTER CLASS
```

Here, to create the object of inner class `demo1`, an object `d` of outer class `demo` is first created and using `d.new demo1()` statement, the object of class `demo1` is created.

```
/*PROG 15.17 DEMO OF NESTED AND INNER CLASS VER 6 */

class demo
{
    static int num = 20;
    class demo1
    {
        static void display()
        {
            System.out.println("Hello from display num:"+num);
        }
    } //inner class ends
} //outer class ends
class JPS6
{
    public static void main(String[] args)
    {
        demo.demo1.display();
    }
}

OUTPUT:
C:\JPS\ch15>javac JPS6.java
JPS6.java:7: inner classes cannot have static declarations
        static void display()
                           ^
1 error
```

As stated above, inner classes cannot have static data or method. So the compiler flashes error.

15.10.1 Inner Classes in Methods and Scopes

In all the earlier examples, the inner class has been defined within the scope of outer class. But there are a number of other ways for defining inner classes; inner classes can be created within a method or even an arbitrary scope. For example, an inner class can be defined within the block defined by a method or even within the body of a `while`/`for` loop. This is very useful when a complicated problem is being solved and a class is to be created to aid in the solution, but one does not want it publicly available.

Consider first an example where an inner class is defined within a loop.

```
/*PROG 15.18 DEMO OF NESTED INNER CLASSES VER 7 */

class demo
{
    boolean flag = true;
    void show()
    {
```

```

        for (int i = 0; i < 5; i++)
        {
            class demo1
            {
                demo1()
                {
                    System.out.println("\nflag = " + flag);
                }
            }//inner class ends
            new demo1();
        }//for loop ends
    }//method show ends
}//outer class ends
class JPS7
{
    public static void main(String[] args)
    {
        demo d = new demo();
        d.show();
    }
}

```

OUTPUT:

```

flag = true

```

In this program, inner class `demo1` has been defined within the scope of `for` loop. The inner class `demo` has got just one default constructor which displays the value of data member `flag` defined within an object of `demo1` class which calls the default constructor. Note as the inner class is defined within the scope of `for` loop when `new demo1()` executes, it calls the default constructor and executes five times. This displays the value of `flag` five times. We could have also written the code in this manner, if instead of default constructor methods say `display` was used.

```

demo1 d = new demo1();
d1.display();

```

Note: Creation of a class within a local scope is known as local inner class. The scope may be created using a method, loop or any other scope.

In the next example, an inner class is defined within `if` conditional block, that is also within a method.

```

/*PROG 15.19 DEMO OF INNER CLASS VER 8 */

class demo
{
    void show(boolean b)
    {
        if (b)

```

```

{
    class demo1
    {
        String str;
        demo1(String s)
        {
            str = s;
        }
        String getstr()
        {
            return str;
        }
    }//inner class ends;
    demo1 d1 = new demo1("Inner");
    String st = d1.getstr();
    System.out.println("\nIn show str = " + st);
}//if ends here
}//method show ends
}//outer class ends
class JPS8
{
    public static void main(String[] args)
    {
        demo d = new demo();
        d.show(true);
    }
}
OUTPUT:
In show str = Inner

```

In the `show` method of outer class `demo`, an inner class `demo1` has been defined. The creation of `demo1` class is dependent on the boolean argument passed to the `show` method. If the boolean argument `b` is true, the inner class `demo1` is created. The class has got one argument constructor which takes a `String` parameter. The method `getstr` returns the string. Just before the end of if block, an object of `demo1` class is created using one argument `String` constructor. The string passed is returned back using the `getstr` method. The same is displayed in the next line. Note the class `demo1` is nested inside the scope of an `if` statement. This does not mean that the class is conditionally created: it gets compiled along with everything else. However, it is not available outside the scope in which it is defined. Other than that, it looks just like an ordinary class.

15.10.2 Static Nested Class

A brief overview of static nested class has already been given earlier. They are not frequently employed so there is not much concentration on this topic; however, a few programs will be given. They are used where connection between the inner class object and the outer class object is not required. This is commonly called a nested class. To understand the meaning of static when applied to inner classes, it must be remembered that the object of an ordinary inner class implicitly keeps a reference to the object of the enclosing class that created it. This being the reason, in inner classes one is able to access members without objects. It was mentioned earlier that fields and methods in ordinary inner classes can only be at the outer level of a class,

so ordinary inner classes cannot have static data, static fields or nested classes. However, nested classes can have all of these.

Consider the example given below:

```
/*PROG 15.20 DEMO OF STATIC NESTED CLASS */

class demo
{
    int num = 20;
    static class demo1
    {
        void display()
        {
            demo d = new demo();
            System.out.println("\nHello from display num: "
                +d.num);
        }
    }//inner class ends
}//outer class ends
class JPS9
{
    public static void main(String[] args)
    {
        demo.demo1 d1 = new demo.demo1();
        d1.display();
    }
}
OUTPUT:
Hello from display num: 20
```

Explanation: In this program, the class `demo1` is static modifier written prior to class keyword. Due to this, members of the enclosing class cannot be accessed directly inside `demo1` class. An object of the `demo` class must be created and using that object, the members of enclosing class can be called. This is what has been done for accessing the member `num` of enclosing class `demo`.

15.10.3 Inner Classes in Event Handling

The concept of inner classes has been discussed in the previous section. In this section, it is used to show how useful they can be in event handling. Consider the program given below which is a modified form of the program shown earlier.

```
/*PROG 15.21 DEMO OF INNER CLASS IN EVENT HANDLING VER 1*/

import java.applet.*;
import java.awt.event.*;
import java.awt.*;
<html>
<applet code="Inner" width =350 height = 120>
```

```

</applet>
</html>
public class Inner extends Applet
{
    public void init()
    {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter
    {
        public void mousePressed(MouseEvent me)
        {
            Graphics g = getGraphics();
            g.drawString("HELLO", me.getX(), me.getY());
        }
    }
}

```

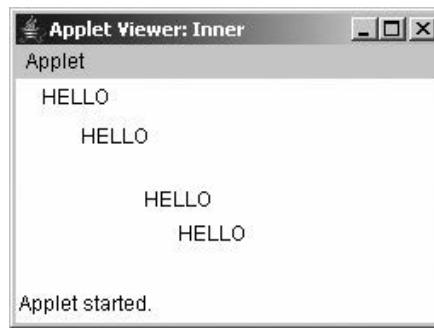


Figure 15.12 Output screen of Program 15.21

Explanation: In this program, the class `MyMouseAdapter` is defined within the class `Inner` so it becomes inner class for `Inner`. Now, all the methods of `Applet` and `Component` class can be used without defining in the `MyMouseAdapter` class. See how easy it has become to write the event handling code using inner and adapter class.

```

/*PROG 15.22 DEMO OF INNER CLASS IN EVENT HANDLING VER 2 */

import java.applet.*;
import java.awt.event.*;
import java.awt.*;
<html>
<applet code="tem1" width =350 height = 120>
</applet>
</html>
public class tem1 extends Applet
{
    public void init()
    {
        addMouseListener(new MyMouseAdapter());
    }
}

```

```
}

class MyMouseAdapter extends MouseAdapter
{
    public void mousePressed(MouseEvent me)
    {
        Graphics g = getGraphics();
        switch(me.getButton())
        {
            case MouseEvent.BUTTON1:
                g.drawString("HELLO1", me.getX(),
                            me.getY());
                break;
            case MouseEvent.BUTTON2:
                g.drawString("HELLO MIDDLE",
                            me.getX(), me.getY());
                break;
            case MouseEvent.BUTTON3:
                g.drawString("HELLO RIGHT",
                            me.getX(), me.getY());
                break;
        }
    }
}
```

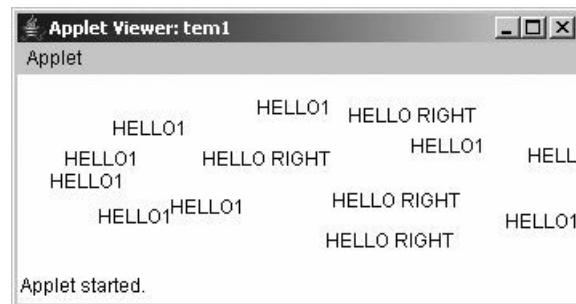


Figure 15.13 Output screen of Program 15.22

Explanation: This program is simple compared to the previous one, but here the types of mouse button pressed are recognized on the basis of writing string onto the current mouse coordinate.

15.10.4 Anonymous Inner Class

Anonymous inner classes are inner classes that do not have a name. To understand this, consider the code given below:

/*PROG 15.23DEMO OF ANONYMOUS CLASS VER 1 */

```
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
<html>
```

```

<applet code="AICdemo" width =350 height = 120>
</applet>
</html>
public class AICdemo extends Applet
{
    public void init()
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent me)
            {
                Graphics g = getGraphics();
                g.drawString("HELLO", me.getX(), me.getY());
            }
        }); //anonymous inner class ends here
    }//init method ends here
}//class ends here

```

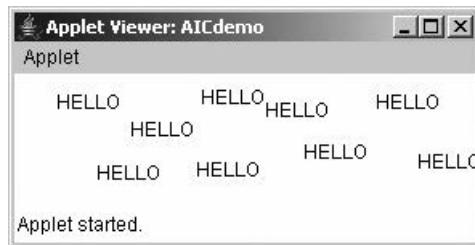


Figure 15.14 Output screen of Program 15.23

Explanation: A new anonymous class is created inside the `addMouseListener` method. Note the class has no name but it has body. The body of the class is shown here separately for better understanding:

```

{
    public void mousePressed(MouseEvent me)
    {
        Graphics g = getGraphics();
        g.drawString("Hello",me.getX(),me.getY());
    }
}

```

The above class is anonymous class as it is having no name but written as an expression/argument for the `addMouseListener` method. Note the entire class is within the parenthesis so when closing parenthesis is put, semicolon has to be written. The class is instantiated automatically when the above code executes. The important point to note here is that syntax `new MouseAdapter()` tells the compiler that anonymous class extends the `MouseAdapter` class.

```
/*PROG 15.24 DEMO OF ANONYMOUS CLASS VER 2 */
```

```

import java.applet.*;
import java.awt.event.*;
import java.awt.*;

```

```

/*
<html>
<applet code="tem2" width =350 height = 120>
</applet>
</html>
*/
public class tem2 extends Applet
{
    public void init()
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent me)
            {
                Graphics g = getGraphics();
                //setForeground(Color.blue);
                g.fillOval(me.getX()-10,me.getY()-10,20,
                           20);
            }
        }); //anonymous inner class ends here
        addKeyListener(new KeyAdapter()
        {
            public void keyPressed(KeyEvent ke)
            {
                if (ke.getKeyCode() == KeyEvent.VK_R)
                    setForeground(Color.red);
                else if (ke.getKeyCode() == KeyEvent.VK_G)
                    setForeground(Color.green);
                if (ke.getKeyCode() == KeyEvent.VK_B)
                    setForeground(Color.blue);
            }
        }); //anonymous inner class ends here
    } //init method ends here
} //class ends

```

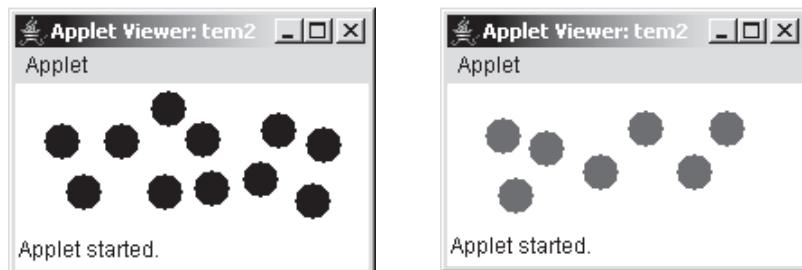


Figure 15.15 Output screen of Program 15.24

Explanation: In this program, two anonymous classes have been created. In the first anonymous inner class, the code is written to handle mouse-clicked event. On the mouseClicked event handler, the code is written to draw small filled circle of radius 10 wherever the mouse is clicked in the applet window. The circles are drawn in the default fill color black.

In the other anonymous inner class inside the addKeyListener method, the KeyAdapter class has been extended and in it written the keyPressed method. Now depending on three keys ‘R’, ‘G’ and ‘B’, the foreground color is changed to red, green and blue, respectively.

15.11 PONDERABLE POINTS

1. An event is any happening or occurring.
2. There are two main models for handling events in Java: The old model is known as Inheritance Event Model and the new one Delegation Event Model. For implementing event-handling mechanism using Delegation Event Model, the package `java.awt.event` must be imported.
3. Event model is based on the concept of an “Event Source” and “Event Listener”. Any object that is interested in receiving messages (or events) is called an Event Listener. Any object that generates these messages (or events) is called Event Source.
4. In an AWT program, the vent source is typically a GUI component and the listener is commonly an “adapter” object which implements the appropriate listener (or set of listeners) in order for an application to control the flow/handling of events.
5. The ActionEvent class is used for handling events like pressing a button, clicking a menu item or list box item. For all these events an ActionEvent is generated.
6. The AdjustmentEvent class represents adjustment events generated by adjustable objects like a scroll bar.
7. The class represents a low-level event which indicates that a component moved, changed size, or changed visibility. It is the root class for the other component-level events classes like KeyEvent, MouseEvent, WindowEvent, etc.
8. The ItemEvent class represents a semantic event (occurred on some GUI object) which indicates that an item was selected or deselected.
9. The KeyEvent class represents an event which indicates that a keystroke occurred in a component.
10. The MouseEvent class represents an event which indicates that a mouse action occurred in a component.
11. For handling Button related event, ActionListener interface must be implemented.
12. For handling ScrollBar related event, AdjustmentListener interface must be implemented.
13. For handling List, CheckBox, radio button related event, ItemListener interface must be implemented.
14. For handling keyboard related event, KeyListener interface must be implemented.
15. For handling mouse events like press, click, enter and exit on a component, MouseListener interface must be implemented.
16. For handling mouse events like mouse move and mouse dragged on a component, MouseMotionListener interface must be implemented.
17. An Adapter class is a class that provides an empty implementation of all methods in an event listener interface. Adapter classes can be used where one is interested in all of the methods of the interface except one or two.
18. It is possible to place definition within another class definition. This is called nested class.
19. A static nested class is the one that has static keyword written before class definition. A static class can access all the members of its enclosing class only through objects. It cannot access them directly. In general, static nested class is called simply nested class.
20. A non-static nested class is that class where static keyword is not present before class definition. The most important type of nested class is the inner class. It can access all the members of its enclosing class without using an object. This being the reason, it is frequently used. In general, non-static nested class is called inner class.

REVIEW QUESTIONS

1. What is event handling? Also explain low level events and high level events.
2. Explain the Event Class hierarchy in Java.
3. What are processing events? Explain with an example.
4. Explain the following terms:
 - (a) Mouse listeners
 - (b) Mouse motion listeners
 - (c) Key events
 - (d) Modifier keys
 - (e) Mouse button modifiers
 - (f) Focus events
5. Write a program to create a calculator using event handling.
6. What is event queue? Explain with the help of a suitable example.
7. Write a program to demonstrate the use of ActionListener.
8. Write a program to demonstrate the use of mouse handling events.
9. Write a program to create a menu for a typical gas agency system.
- (g) Component events
- (h) Consuming event

Multiple Choice Questions

1. For implementing event handling mechanism using Delegation Event Model, the package _____ must be imported.
 - (a) java.Applet.event
 - (c) java.awt.event
 - (b) java.applet.event
 - (d) java.Awt.event
2. Event model is based on the concept of an
 - (a) Applets
 - (c) Event listeners
 - (b) Event source
 - (d) (b) and (c)
3. For handling list, checkbox, radio button related event, _____ must be implemented
 - (a) ActionListener interface
 - (b) ActionEvent class
 - (c) ItemListener interface
 - (d) None of the above
4. For handling keyboard related event _____ must be implemented
 - (a) KeyEvent interface
 - (b) KeyListener interface
 - (c) KeyStroke interface
 - (d) None of the above
5. Which of the following is used to provide an empty implementation of all methods in an event listener interface?
 - (a) KeyEvent class
 - (c) Adapter class
 - (b) Adapter object
 - (d) ItemListener
6. _____ is generated when components are added to or removed from a container.
 - (a) ActionEvent class
- (b) AdjustmentEvent class
- (c) ContainerEvent class
- (d) FocusEvent class
7. Every event is a subclass of
 - (a) java.applet.event
 - (b) java.util.EventObject
 - (c) java.io.Event
 - (d) None of the above
8. Every source has different methods for registering the listeners. The general form is
 - (a) void addSTypeListener()
 - (b) public void addSTypeListener()
 - (c) public void addSType(STypeListener listener)
 - (d) public void addSTypeListener (STypeListener listener)
9. When a button is pressed, an action event is generated that has a command name equal to the label on the button—which of the following will be used?
 - (a) String getAction()
 - (b) public String getAction()
 - (c) public String getActionCommand()
 - (d) None of the above
10. Which method returns a Point object containing the x and y coordinator relative to the source component?
 - (a) public int getX()
 - (b) public int getY()
 - (c) public Point getPoint()
 - (d) (a) and (b)

KEY FOR MULTIPLE CHOICE QUESTIONS

1. c 2. d 3. c 4. b 5. c 6. c 7. b 8. d 9. c 10. c

16

Working with AWT

16.1 INTRODUCTION TO AWT

AWT stands for Abstract Window Toolkit. The toolkit is defined within the **java.awt** package. It contains all of the classes for creating user interfaces like Buttons, Labels, TextBox, Lists, etc., and for painting graphics and images. The AWT provides many classes for programmers to use. It is the connection between an application and the native GUI. The AWT hides the system from the underlying details of the GUI, the application will be running on and thus is at very high level of the abstraction. All of the classes are arranged in a meaningful hierarchy so that many of the features of the top-most classes can be used in a number of subclasses.

16.2 STRUCTURE OF THE AWT

The structure of the AWT is rather simple: Components are added to and then laid out by layout managers in Containers. There is a variety of event handling, menu, fonts and graphics classes in addition to the above two. The hierarchy of AWT classes is as shown below:

- BorderLayout
- CheckboxGroup
- Color
- Component
 - Button
 - Canvas
 - Checkbox
 - Choice
 - Container
 - Panel
 - Window
 - Dialog
 - Frame
 - Label
 - List
 - Scrollbar
 - TextComponent
 - TextArea
 - TextField
- Dimension
- Event

- FileDialog
- FlowLayout
- Font
- FontMetrics
- Graphics
- GridLayout
- GridBagConstraints
- GridbagLayout
- Image
- Insets
- Media Tracker
- MenuComponent
 - MenuBar
 - MenuItem
 - CheckboxMenuItem
 - Menu
- Point
- Polygon
- Rectangle
- Toolkit

Most commonly used classes and their brief description of AWT are given below:

| Class | Description |
|------------------|---|
| AWTEvent | The root event class for all AWT events. |
| BorderLayout | Lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west and centre. |
| Button | Creates a push-button control. |
| Canvas | A Canvas component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user. |
| CardLayout | A CardLayout object is a layout manager for a container. Card layouts emulate index cards. Only the one on top is showing. |
| Checkbox | A graphical component that can be in either an “on” (true) or “off” (false) state. |
| CheckboxGroup | The CheckboxGroup class is used to group together a set of Checkbox buttons. |
| CheckBoxMenuItem | This class represents a check box that can be included in a menu. |
| Choice | The Choice class presents a pop-up menu of choices. |
| Color | The Color class is used to encapsulate colors in the default RGB color space. |
| Component | An object having a graphical representation that can be displayed on the screen and that can interact with the user. |
| Container | A generic AWT container object is a component that can contain other AWT components. |
| Cursor | A class to encapsulate the bitmap representation of the mouse cursor. |
| Dialog | A top-level window with a title and a border that is typically used to take some form of input from the user. |

| | |
|---------------------|---|
| Dimension | The Dimension class encapsulates the width and height of a component (in integer precision) in a single object. |
| FileDialog | The FileDialog class displays a dialog window from which the user can select a file. |
| FlowLayout | Arranges components in a directional flow, much like lines of text in a paragraph. |
| Font | The Font class represents fonts, which are used to render text in a visible way. |
| FontMetrics | The FontMetrics class defines a font metrics object, which encapsulates information about the rendering of a particular font on a particular screen. |
| Frame | A top-level window with a title and a border. |
| Graphics | Encapsulates the graphics context. This context is used by the various output methods to display output in a window. |
| Graphics2D | The Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management and text layout. |
| GraphicsEnvironment | The GraphicsEnvironment class describes the collection of GraphicsDevice objects and Font objects available to a Java™ application on a particular platform. |
| GridBagLayout | The GridBagLayout class is a flexible layout manager that aligns components vertically and horizontally, without requiring that the components be of the same size. |
| GridLayout | The GridLayout class is a layout manager that lays out a container's components in rectangular grid. |
| Image | The abstract class Image is the superclass of all classes that represent graphical images. |
| Insets | An Insets object is a representation of the borders of a container. |
| Label | A Label object is a component for placing text in a container. |
| List | The List component presents the user with a scrolling list of text items. |
| MediaTracker | The MediaTracker class is a utility class to track the status of a number of media objects. |
| Menu | A Menu object is a pull-down menu component that is deployed from a menu bar. |
| MenuBar | TheMenuBar class encapsulates the platform's concept of a menu bar bound to a frame. |
| MenuItem | All items in a menu must belong to the class MenuItem, or one of its subclass. |
| MenuShortcut | The MenuShortcut class represents a keyboard accelerator for a MenuItem. |
| Panel | Panel is the simplest container class. |
| Point | Encapsulates a Cartesian coordinate pair, stored in x and y. |
| Polygon | The Polygon class encapsulates a description of a closed, two-dimensional region within a coordinate space. |
| PopupMenu | A class that implements a menu which can be dynamically popped up at a specified position within a component. |

| | |
|-------------|---|
| Rectangle | Specifies an area in a coordinate space that is enclosed by the Rectangle object's top-left point (x, y) in the coordinate space, its width and its height. |
| Scrollbar | The Scrollbar class embodies a scroll bar, a familiar user-interface object. |
| ScrollPane | A container class which implements automatic horizontal and/or vertical scrolling for a single child component. |
| SystemColor | A class to encapsulate symbolic colors representing the color of native GUI objects on a system. |
| TextArea | A TextArea object is a multi-line region that displays text. |
| TextField | A TextField object is a text component that allows for the editing of a single line of text. |
| Toolkit | This class is the abstract superclass of all actual implementations of the AWT. |
| Window | A Window object is a top-level window with no border and no menu bar. |

Table 16.1 Classes defined by `java.awt` package

16.3 AWT WINDOW HIERARCHY

The AWT window hierarchy is shown by the in Figure 16.1 diagram.

All of the classes are discussed in the following sections.

16.3.1 The Component Class

A component is an object having a graphical representation that can be displayed on the screen and interact with the user. Examples of components are the buttons, checkboxes and scrollbars of a typical graphical user interface. In other words the visible User Interface (UI) controls that the user interacts with all of which have been added to a Container are components. Anything that is derived from the class Component can be a component. The class contains a number of useful methods for event handling, managing graphics painting and sizing and resizing the window. This class is the top-most class in the AWT hierarchy. The Component class is the abstract superclass of the non-menu-related AWT components.

16.3.2 The Container Class

A generic AWT container object is a component that can contain other AWT components. The Container class is the subclass of the Component class. Components added to a container are tracked in a list. The order of the list will define the component's font-to-back stacking order within the container. If no index is specified when adding a component to a container, it will be added to the end of the list.

Containers (Frames, Dialogs, Windows and Panels) can contain components and are themselves components, hence can be added to Containers. Containers usually handle events that occurred to the Component, but the code for handling events can be written in the component, too. A container is responsible for laying out (i.e., positioning) any components that it contains. It does this through the use of various layout managers that are discussed later in Chapter 17.

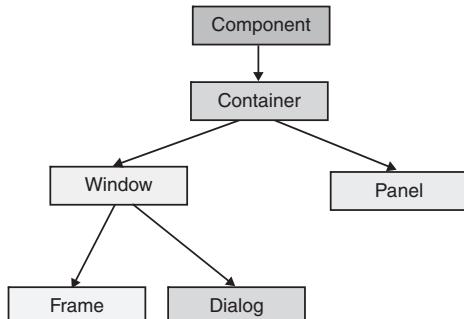


Figure 16.1 AWT window hierarchy

Every Container is a component since it is derived from Component; thus it can also behave and be used like a Component. All of the methods in Component can be used in a Container. Container has two direct subclasses, Window and Panel. They are discussed below.

16.3.3 The Panel Class

Panel is the simplest container class because Container class is the superclass for Panel class. A panel provides space in which an application can attach any other component, including other panels. A Panel is a container that does not exist independently. The Applet class is a subclass of Panel, and as has been seen, as applet does not exist on its own; it must be displayed on a Web page or in some other window. Any panel must be contained inside something else, either a Window, another Panel, or in the case of an Applet, a page in a Web browser. It means that a Panel is a window that contains no title bar, menu bar or border. Due to this, when an applet is run inside a Web browser no title, border or menu is displayed. A simple plain rectangular region is displayed. In case of running, applet using appletviewer, the appletviewer application provides the above-mentioned border, title and menus.

The component can be added to the Panel using add method as will be seen in the programs following. A number of methods defined by Component class like setSize, SetBounds and setTitle can be used for changing their shape and look. The fact that panels can contain other panels means that one can have many levels of components containing other components.

16.3.4 The Window Class

A **Window** objects is a top-level window with no borders and no menu bar. A Window represents an independent to-level window that is not contained in any component. Window is not really meant to be used directly. It has two subclasses: **Frame**, to represent ordinary windows that can have their own bars, and **Dialog** to represent dialog boxes that are used for limited interactions with the user.

16.3.5 The Frame Class

Applets are a fine idea. It is nice to be able to put a complete program in a rectangle on a Web page. But more serious, large-scale programs have to run in their own window, independently of a Web browser. In Java, a program can open an independent window by creating an object of type Frame. A Frame has a title, displayed in the title bar at the top of the window. It can have a menu bar containing one or more pull-down menus. A Frame is a Container, which means that it can hold other GUI components. The default layout manager for a frame is a BorderLayout. A common way to design a frame is to add a single GUI component, such as a Panel or Canvas, in the layout's center position so that it will fill the entire frame.

It is possible for an applet to create a frame. The frame will be a separate window from the Web browser window in which the applet is running. Any frame created by an applet includes a warning message such as "Warning: Insecure Applet Window". The warning is there so that one can always recognize window created by applets. When a Frame window is created by a program rather than an applet, a normal window is created.

16.3.6 Applet Frame Class

The Frame class you will use to create child window's of an applet window. The class defines the following two commonly used constructors.

1. **public Frame()**

This form of constructor constructs a new instance of Frame that is initially invisible. The title of the Frame is empty.

2. `public Frame(String title)`

This form of constructor creates a new, initially invisible `Frame` object with the specified title.

The useful methods of `Frame` class are discussed below:

- `setSize()` method

This method is used for setting the dimensions of the frame window. The method has two forms:

```
void setSize(int w, int h)
void setSize(Dimension d)
```

In the first form, the width and height are specified by `w` and `h` in terms of pixels. In the second object, a dimension object `d` is passed that contains width and height of the frame window.

- `getSize()` method

This method returns the dimension of the frame window as `Dimension` object. Its signature is given below:

```
Dimension getSize()
```

- `setVisible()` method

This frame window after creation is invisible. To make it visible the `setVisible` method can be used. It has the following form:

```
void setVisible(boolean flag)
```

If the flag is true frame window will be visible else, it will be invisible.

- `setTitle()` method

This method can be used for setting the title of the frame window. It has the following form:

```
void setTitle(String title)
```

- `getTitle()` method

```
String getTitle()
```

This method returns the title of the frame. The title is displayed in the frame's border.

- `setResizable()` method

The signature of the method is given below:

```
void setResizable(boolean rs)
```

This method sets whether this frame is resizable by the user. If `rs` is false, the frame window cannot be resized. By default, the frame is resizable.

Some other methods will be demonstrated programmatically.

The frame class can receive the following events:

- `WINDOW_OPENED`
- `WINDOW_CLOSING`
- `WINDOW_CLOSED`
- `WINDOW_ICONIFIED`
- `WINDOW_DEICONIFIED`
- `WINDOW_ACTIVATED`
- `WINDOW_DEACTIVATED`
- `WINDOW_GAINED_FOCUS`
- `WINDOW_LOST_FOCUS`
- `WINDOW_STATE_CHANGED`

One important point to note while working with frame window is that frame window cannot be closed manually by clicking at the close button. In case frame window is created by the applet window, frame window will be closed automatically as the applet window is closed. But in case frame window is created by simple application program, it cannot be closed. For both the cases the event handling code can be

written. For closing the window the `windowClosing` method of `WindowListener` interface will be written for removing the window from the screen.

Now, a few programs are presented that give an idea on how to work with `Frame` class. A `Frame` can be created inside an applet or an application program. The applications programs of `Frame` are first presented. Subsequently, some `Frame`-based applets will be dealt with.

```
/*PROG 16.1 CREATING YOUR FIRST FRAME WINDOW */

import java.awt.*;
class tem1
{
    public static void main(String[] args)
    {
        Frame F = new Frame("This is My First Frame");
        F.setSize(200, 200);
        F.setVisible(true);
    }
}
```

Explanation: The program is very simple. A new `Frame` instance is created by the name `F` and with title "This is My First Frame". The size of the new frame window is 200 by 200. The window is made visible by using `setVisible` method. Note that after the frame window is created, it cannot be destroyed by clicking the close button. It will have to be closed forcibly by using command **Ctrl-C** under window or explicitly closing editor output window, in case some Java editor is used for running the programs. Note the program is a simple application program, and not an applet.

In general, the `Frame` class is extended by some class because events cannot be handled that occurs within `Frame` window. The next program shows how window can be closed and extended.



Figure 16.2 Output screen of Program 16.1

```
/*PROG 16.2 CLOSING WINDOW EVENT WITH FRAME */

import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        Winadp wa = new Winadp(this);
        addWindowListener(wa);
    }
}
class Winadp extends WindowAdapter
{
```

```

MyFrame mf;
public Winadp(MyFrame f)
{
    mf = f;
}
public void windowClosing(WindowEvent we)
{
    mf.setVisible(false);
    System.exit(0);
}
class JPSI
{
    public static void main(String args[])
    {
        Frame fr = new MyFrame("Frame Demo");
        fr.setVisible(true);
        fr.setSize(300, 300);
    }
}

```

Explanation: In this program, the class MyFrame extends the Frame class. The constructor of the MyFrame class takes title as String objects and passes it to the constructor of Frame class. Similar to adapter classes for applets has been seen in the previous chapter, one adapter class is created for handling the WINDOW_CLOSING event. The window listener is added using addWindowListener method. In this method, an object of Winadp class is passed as argument. It means the frame window will be receiving window events. The constructor of Winadp class takes reference of MyFrame class as argument. Recall this is necessary as to work with methods of MyFrame class and its parent class within Winadp class. In the overridden windowClosing method, the visibility of frame window is set to false using setVisible method and the program is terminated using System.exit(0). If not used, window will be hidden, but there will be no prompt/main window back. CTRL-C will have to be pressed.

In the main, the frame window is created using constructor of MyFrame class. The reference is stored in fr. Rest is simple to understand.

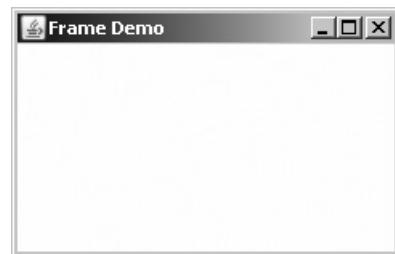


Figure 16.3 Output screen of Program 16.2

```
/*PROG 16.3 DISPLAY "HELLO" ON MOUSE CLICKED */
```

```

import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        addWindowListener(new Winadp(this));
    }
}

```

```

        addMouseListener(new MouseAdp(this));
    }
}
class Winadp extends WindowAdapter
{
    MyFrame mf;
    public Winadp(MyFrame f)
    {
        mf = f;
    }
    public void windowClosing(WindowEvent we)
    {
        mf.setVisible(false);
        System.exit(0);
    }
}
class MouseAdp extends MouseAdapter{
    MyFrame m;
    public MouseAdp(MyFrame ad)
    {
        m = ad;
    }
    public void mouseClicked(MouseEvent me)
    {
        Graphics g = m.getGraphics();
        g.drawString("Hello", me.getX(), me.getY());
    }
}
class JPS2{
    public static void main(String args[])
    {
        Frame fr = new MyFrame("Mouse Events in Frame Window");
        fr.setVisible(true);
        fr.setSize(300,250);
        fr.setBackground(Color.cyan);
        fr.setForeground(Color.blue);
    }
}
}

```

Explanation: The program is simple. Similar to handling window events using adapter class, the mouse event is handled here using adapter class. MouseAdp is created that extends MouseAdapter class in which the method mouseClicked is overridden. The method displays “Hello” wherever the mouse is clicked inside the frame window. In the main method, foreground color of the frame window is set using method setForeground and setBackground.

A number of other methods that have been used in the applet can be used here also. The most useful method is paint. If one wants to perform some drawings in the

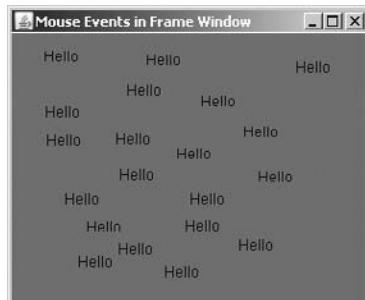


Figure 16.4 Output screen of Program 16.3

frame window, simply define paint method and write code in it. To see this simply add the paint method in the MyFrame class and run the program after compiling.

```
public void paint(Graphics g)
{
    g.drawRect(50,50,50,50);
    g.fillOval(100,100,50,50);
}
```

```
/*PROG 16.4 FRAME WITHIN APPLET */

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        Winadp wa = new Winadp(this);
        addWindowListener(wa);
    }
}
class Winadp extends WindowAdapter
{
    MyFrame mf;
    public Winadp(MyFrame f)
    {
        mf = f;
    }
    public void windowClosing(WindowEvent we)
    {
        mf.setVisible(false);
        System.exit(0);
    }
}
public class Frame1 extends Applet
{
    public void init()
    {
        Frame fr = new MyFrame("Frame Demo");
        fr.setVisible(true);
        fr.setSize(300, 300);
    }
}
```

Explanation: The program is similar to the earlier program of creation of frame with window closing event but here instead of main method, applet class has been used. A new applet class Frame2 is created and inside it the coding for creation and display of frame window is written. Note in the program, the setBounds method is used for setting the x, y coordinate and width and height of the frame window.



Figure 16.5 Output screen of Program 16.4

```
/*PROG 16.5 EVENT HANDLING, BOTH IN APPLET AND FRAME */

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
class MyFrame extends Frame implements MouseListener
{
    MyFrame(String title)
    {
        super(title);
        Winadp wa = new Winadp(this);
        addWindowListener(wa);
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent me)
    {
        Graphics g = getGraphics();
        setForeground(Color.red);
        g.drawString("Frame Window",me.getX(),me.getY());
    }
    public void mouseExited(MouseEvent me) {}
    public void mousePressed(MouseEvent me) {}
    public void mouseReleased(MouseEvent me) {}
    public void mouseEntered(MouseEvent me) {}
}
class Winadp extends WindowAdapter
{
    MyFrame mf;
    public Winadp(MyFrame f)
    {
        mf = f;
    }
    public void windowClosing(WindowEvent we)
    {
        mf.setVisible(false);
        System.exit(0);
    }
}
public class Frame2 extends Applet implements MouseListener
```

```

{
    public void init()
    {
        Frame fr = new MyFrame("Frame Demo");
        fr.setVisible(true);
        fr.setBounds(150,150,250,250);
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent me)
    {
        Graphics g = getGraphics();
        setForeground(Color.blue);
        g.drawString("Applet Window",me.getX(),me.getY());
    }
    public void mouseExited(MouseEvent me) {}
    public void mousePressed(MouseEvent me) {}
    public void mouseReleased(MouseEvent me) {}
    public void mouseEntered(MouseEvent me) {}
}

```

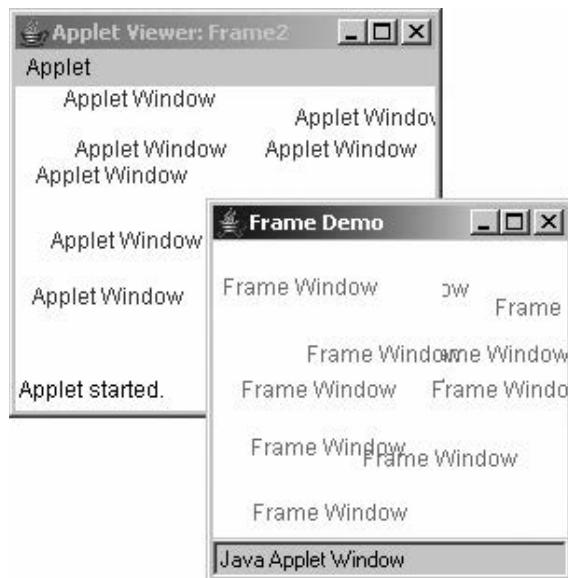


Figure 16.6 Output screen of Program 16.5

Explanation: This program demonstrates how events can be handled both in frame window as well as in applet window. The code has been explained in the earlier programs, so no need to repeat it here again. One thing to note here is that both the classes MyFrame and Frame2 implement the interfaces and define the methods of the interfaces. Both the classes write the code in mouseClicked method and provide empty implementation of other methods of the interfaces.

16.4 AWT CONTROLS

This section discusses some of the GUI interface elements that are represented by subclasses of Component. It also introduces the event classes and listener interfaces associated with each type of component that have been seen earlier.

The Component class itself defines many useful methods that can be used with components of any type. Some of these have already been used in examples. Let `comp` be a variable that refers to any component. Then the following methods are available (among many others):

1. `comp.getSize()` is a function that returns an object belonging to the class Dimension. This object contains two instance variables—`comp.getSize().width` and `comp.getSize().height`—that give the current size of the component. One warning: When a component is first created, its size is zero. The size will be set later, probably by a layout manager. A common mistake is to check the size of a component before that size has been set.
2. `comp.getParent()` is a function that returns a value of type Container. The container is the one that contains the component, if any. For a top-level component such as a Window or Applet, the value will be null.
3. `comp.getLocation()` is a function that returns the location of the top-left corner of the component. The location is specified in the coordinate system of the component's parent. The returned value is an object of type Point. An object of type Point contains two instance variables, `x` and `y`.
4. `comp.setEnabled(true)` and `comp.setEnabled(false)` can be used to enable and disable the component. This is only useful for certain types of component, such as button. When a button is disabled, its appearance changes, and clicking on it will have no effect. There is a boolean-valued function, `comp.getEnabled()`, that can be called to discover whether the component is enabled.
5. `comp.setVisible(true)` and `comp.setVisible(false)` can be called to hide or show the component.
6. `comp.setBackground(color)` and `comp.setForeground(color)` set the background and foreground colors for the component. If no colors are set for a component, it inherits the colors of its parent container. The command `comp.setFont(font)` sets the default font that is used for text displayed on the component. (These should work for all components, but might not work properly for some of the standard components, depending on the version of Java.)

For the rest of this section, subclasses of Component will be discussed that represent common GUI components. Remember that using any component is a multi-step process. The component object must be created with a constructor and added to a container. In many cases, a listener must be registered to respond to events from the component, and in some cases, a reference to the component must be saved in an instance variable so that the component can be manipulated by the program after it has been created.

For adding components to the applet or any other type of window add method have to be used defined by the Component class. The method has six forms. Only the most commonly used forms and their signatures are shown below:

```
Component add (Component comp)
```

Here, `comp` is the component to be added to the applet or any other type of window. The method returns a reference to the component added.

For the removal of the component, the method remove can be used. It has three forms, but only the commonly used form is shown:

```
void remove (Component comp)
```

Here `comp` is the component to be removed.

The various controls supported by AWT are shown below by giving their class name. All these controls have Component as their super class, so all the methods of Component class can be used by these classes.

| | | | | |
|--------|-----------|---------------|-----------|-------|
| Button | Checkbox | CheckboxGroup | Choice | Label |
| List | Scrollbar | TextArea | TextField | |

Apart from these classes, AWT provides classes for creating menus, dialogs, etc. They will be discussed in details later.

16.4.1 The Button Control

The Button class is used for creating the most frequently used component (i.e., a push button). The class defines two constructors:

1. **public Button()**

This form of constructor constructs a button with an empty string for its label. Later, the label can be set using `setLabel` method.

2. **public Button(String label)**

This form of constructor constructs a button with the specified label.

The commonly used methods of this class are as follows:

1. **public void addActionListener(ActionListener l)**

This method adds the specified action listener to receive action events from this button. Action events occur when a user presses or releases the mouse over this button.

2. **public String getActionCommand()**

This method returns the command name of the action event fired by this button. If the command name is null (default), this method returns the label of the button.

3. **public String getLabel()**

This method returns the label of the button.

4. **public void setLabel(String label)**

This method sets the button's label to be the specified string.

For handling events for the button, the listener for the button has to be added using `addActionListener` method. The parent class must implement the `ActionListener` interface. Whenever the button is clicked, the method `actionPerformed` is called and an object of `ActionEvent` is passed to it. In the `actionPerformed` method, the code for handling the button press event can be written.

A number of examples of Button control are provided.

```
/*PROG 16.6 DEMO OF BUTTON CLASS, WORKING WITH SINGLE BUTTON */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "button1" width = 200 height = 200>
</applet>
*/
public class button1 extends Applet implements ActionListener
{
    Button but; String disp = " ";

```

```

public void init()
{
    setBackground(Color.yellow);
    but = new Button("Click Me");
    add(but);
    but.addActionListener(this);
}
public void actionPerformed(ActionEvent AE)
{
    disp = "You Clicked Me";
    repaint();
}
public void paint(Graphics g)
{
    g.drawString(disp, 80, 100);
}
}

```

Explanation: In this program, a button but is created using **Button (Click Me);** constructor form. This creates a button with a label **Click Me**. The button instance but is added to the applet window using add method. The applet class button implements the ActionListener interface so that the button press event can be handled. Using method addActionListener, a listener for the button is added. When the button is clicked, actionPerformed method is called. In the method the String object is initialized to **You Clicked Me** and repaint method calls the paint method. Thus msg is displayed at **30, 70** in the applet window. The empty constructor form and setLabel method could also be used as:

```

but = new Button();
but.setLabel("Click Me")

```



Figure 16.7 Output screen of Program 16.6

```
/*PROG 16.7 DEMO OF BUTTON CLASS, WORKING WITH TWO BUTTONS */
```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "button2" width = 200 height = 200>
</applet>
*/
public class button2 extends Applet implements ActionListener
{
    Button but1, but2; String disp = " ";
    public void init()
    {
        setBackground(Color.CYAN);
        but1 = new Button("Click Me");
        but2 = new Button("Click Me Too");
    }
}

```

```

        add(but1);
        add(but2);
        but1.addActionListener(this);
        but2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent AE)
    {
        String caption = AE.getActionCommand();
        if(caption.equals("Click Me"))
            disp = "Hello from button1";
        else if(caption.equals("Click Me Too"))
            disp = "Hello from button2";
        repaint();
    }
    public void paint(Graphics g)
    {
        g.drawString(disp, 10, 50);
    }
}

```

Explanation: This program is simple compared to the previous one but here there are two buttons. Code in the init method is simple to understand. Note listeners for both buttons have to be added. In the actionPerformed method, one can come to know which button was pressed. This is done by using the method getActionCommand, which returns the label of the button pressed. The label is compared with the label of both the buttons using equals method. Depending on which button was pressed, appropriate message is put into the msg. The same is displayed using paint method. In the output, msg is displayed when button2 is clicked.

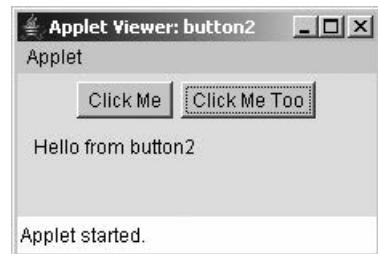


Figure 16.8 Output screen of Program 16.7

```

/*PROG 16.8 DRAWING COLORFUL CRICLES ON BUTTON CLICK */

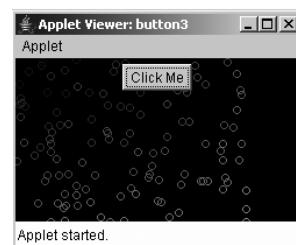
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "button3" width = 250 height = 250>
</applet>
*/
public class button3 extends Applet implements ActionListener
{
    Button but1;
    int h, w;
    public void init()
    {
        setBackground(Color.black);
        but1 = new Button("Click Me");

```

```

        add(but1);
        but1.addActionListener(this);
        h = this.getSize().height;
        w = this.getSize().width;
    }
    public void actionPerformed(ActionEvent AE)
    {
        if (AE.getSource() == but1)
        {
            int x = (int)(Math.random() * 10000);
            int y = (int)(Math.random() * 10000);
            x = x % h;
            y = y % w;
            Graphics g = getGraphics();
            g.setColor(new Color(x%255,y%255,(x*y)%255));
            g.drawOval(x - 3, y - 3, 6, 6);
        }
    }
}

```



Explanation: In this program, circles of radius 6 are drawn in random colors anywhere on to the applet window. Note in the actionPerformed method getSource() method has been used. This method returns the Object that is responsible for the event. Coding in the method is simple to understand.

Figure 16.9 Output screen of Program 16.8

```

/*PRGO 16.9 COUNTING NUMBER OF TIMES BUTTON CLICKED */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "button4" width = 200 height = 100>
</applet>
*/
public class button4 extends Applet implements ActionListener
{
    Button but; String disp = " ";
    static int count = 0;
    public void init()
    {
        setBackground(Color.orange);
        but = new Button("Click Me");
        add(but);
        but.addActionListener(this);
    }
}

```

```

public void actionPerformed(ActionEvent AE)
{
    count++;
    disp = "You clicked me := " + count + "times";
    repaint();
}
public void paint(Graphics g)
{
    g.setColor(Color.BLACK);
    g.drawString(disp, 50, 50);
}
}

```

Explanation: This program is simple. The number of times the button was clicked is to be counted. For this purpose, a static member **count** has been taken. The **count** is incremented by 1 each time the button is clicked in the `actionPerformed` method. This count is then concatenated with the **String** literal “**You clicked me:= count times**” and is displayed in `paint` method.

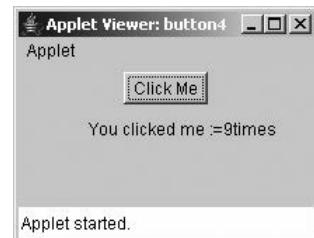


Figure 16.10 Output screen of Program 16.9

```

/*PROG 16.10 SHOWING AND HIDING BUTTONS */

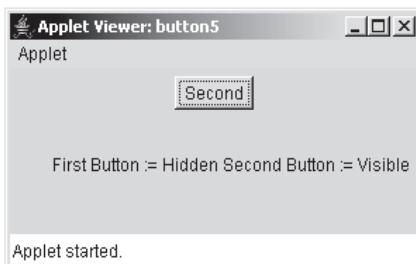
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "button5" width = 200 height = 100>
</applet>
*/
public class button5 extends Applet implements ActionListener
{
    Button but1, but2; String disp = " ";
    public void init()
    {
        setBackground(Color.CYAN);
        but1 = new Button("First");
        but2 = new Button("Second");
        add(but1);
        add(but2);
        but1.addActionListener(this);
        but2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent AE)
    {
        String caption = AE.getActionCommand();
        if (caption.equals("First"))

```

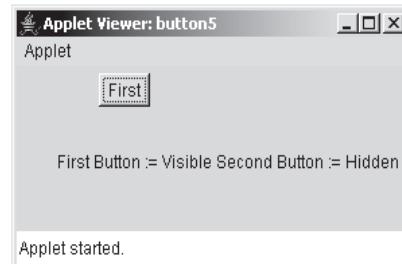
```

        {
            but1.setVisible(false);
            but2.setVisible(true);
            disp = "First Button := Hidden Second Button
                    := Visible";
        }
    else
    {
        but1.setVisible(true);
        but2.setVisible(false);
        disp = "First Button := Visible Second
Button := Hidden";
    }
    repaint();
}
public void paint(Graphics g)
{
    g.drawString(disp, 30, 70);
}
}
}

```



(a) First button hidden



(b) Second button hidden

Figure 16.11 Output screen of Program 16.10

Explanation: In the actionPerformed method, when the first button (with the caption “First”) is clicked it is to be hidden and the second button (with the caption “Second”) displayed. When second button is clicked, the reverse is done (i.e., display the first and hide the second). This has been made possible by using the method setVisible seen earlier.

16.4.2 The Label Class

A Label object is a component for placing text in a container. A label displays a single line of read-only text (i.e., static text). The text can be changed by the application, but a user cannot edit it directly. The Label class defines the following constructors:

1. **public Label()**

This form of constructor constructs an empty label. The text of the label is the empty string “ ”.

2. **public Label(String text)**

This form of constructor constructs a new label with the specified string of text, left justified.

3. **public Label(String text, int alignment)**

This form of constructor constructs a new label that presents the specified string of text with the specified alignment. Possible values for alignment are Label.LEFT, Label.RIGHT and Label.CENTER.

The Label class defines a number of useful methods for getting text/alignment, setting text/alignment of the Label object. They are briefly discussed in the following.

1. **public int getAlignment()**

This method returns the current alignment of this label. The returned values may be any of the Label.LEFT, Label.RIGHT, and Label.CENTER.

2. **public void setAlignment(int alignment)**

This method sets the alignment for this label to the specified alignment. Possible values may be Label.LEFT, Label.RIGHT and Label.CENTER.

3. **public String getText()**

This method returns the text of this label as String object.

4. **public void setText(String text)**

This method sets the text for this label to the specified text.

```
/*PROG 16.11 DEMO OF LABEL CLASS, WORKING WITH SINGLE LABEL */
```

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "label1" width = 150 height = 150>
</applet>
*/
public class label1 extends Applet
{
    Label L1;
    public void init()
    {
        L1 = new Label("This is a Label");
        L1.setBackground(Color.cyan);
        L1.setForeground(Color.red);
        add(L1);
    }
}
```



Figure 16.12 Output screen of Program 16.11

Explanation: This program is simple to understand. Most important thing to note here is that label is placed onto the applet by the default layout manager, that is, FlowLayout manager from left to right. They will be discussed in the next section.

```
/*PROG 16.12 DEMO OF LABEL CLASS, WORKING WITH MULTIPLE LABELS */

import java.applet.*;
import java.awt.*;
/*
<applet code = "label2" width = 200 height = 75>
</applet>
*/
public class label2 extends Applet
{
    Label L[];
    public void init()
    {
        L = new Label[3];
        for (int i = 0; i < L.length; i++)
        {
            L[i] = new Label("Label " + (i + 1));
            L[i].setBackground(Color.cyan);
            L[i].setForeground(Color.red);
            add(L[i]);
        }
        L[0].setAlignment(Label.CENTER);
        L[1].setAlignment(Label.LEFT);
        L[2].setAlignment(Label.RIGHT);
    }
}
```

Explanation: In this program, an array of objects of Label class is created by the name **L**. The array is given length in the **init** method. Note each element of array **L** is initialized separately and assigned the text “Label 1”, “Label 2”, and “Label 3” in the **for** loop. Outside the **for** loop in the **init** method the alignment for various labels is set, but from the output it is clear that it has not worked. This is because the default layout manager which is **FlowLayout** has not been changed, so by default labels or any other component are placed to right.



Figure 16.13 Output screen of Program 16.12

16.4.3 The Checkbox Class

A check box is a graphical component that can be in either an “on”(true) or “off” (false) state. Clicking on a check box changes its state from “on” to “off” or from “off” to “on”. Along with this status of on or off, there is a label associated with check box object. For check box there is the class **Checkbox**.

The class defines the following constructors:

1. **public Checkbox()**

This form of constructor creates a check box with an empty string for its label. The state of this check box is set to “off”.

2. **public Checkbox(String label)**

This form of constructor creates a check box with the specified label. The state of this check box is set to “off”.

3. **public Checkbox(String label, boolean state)**

This form of constructor creates a check box with the specified label and sets the specified state.

4. **public Checkbox(String label, boolean state, CheckboxGroup group)**

5. **public Checkbox(String label, CheckGroup group, boolean state)**

The above two forms of constructors create a check box with the specified label, set to the specified state and in the specified check box group.

The most commonly used methods of these classes are discussed below:

1. **public String getLabel()**

This method returns the label of this check box.

2. **public void setLabel(String label)**

This method sets this check box's label to be the string argument.

3. **public boolean getState()**

This method determines whether this check box is in the “on” or “off” state. The Boolean value true indicates the “on” state and false indicates the “off” state.

4. **public void setState(boolean state)**

This method sets the state of this check box to the specified state. The Boolean value true indicates the “on” state, and false indicates the “off” state.

For handling the events for checkbox `ItemListener` interface have to be implemented in the parent window, that is, applet window or parent window. The listener is registered using the method `addItemListener`. Whenever the checkbox item state is changed, `itemStateChanged` method is called and an object of `ItemEvent` is passed to it. The method serves as an event handler for the checkbox state change event.

```
/*PROG 16.13 IMPLEMENTING CHECKBOX AND ITS EVENT METHODS VER 1 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "checkbox" width = 150 height = 100>
</applet>
*/
public class checkbox1 extends Applet implements ItemListener
{
    String msg = " ";
    Checkbox ch1, ch2;
    public void init()
    {
        setBackground(Color.blue);
    }
}
```

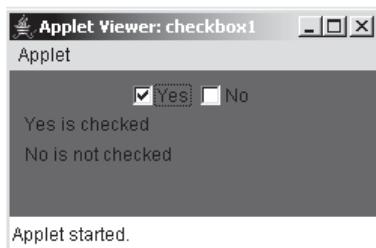
```

        ch1 = new Checkbox("Yes");
        ch2 = new Checkbox("No");
        add(ch1);
        add(ch2);
        ch1.addItemListener(this);
        ch2.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }
    public void paint(Graphics g){
        if (ch1.getState())
            msg = "Yes is checked";
        else
            msg = "Yes is not checked";
        g.drawString(msg, 10, 40);
        if (ch2.getState())
            msg = "No is checked";
        else
            msg = "No is not checked";
        g.drawString(msg, 10, 60);
    }
}

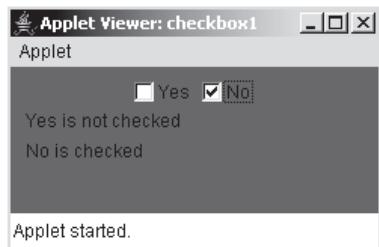
```



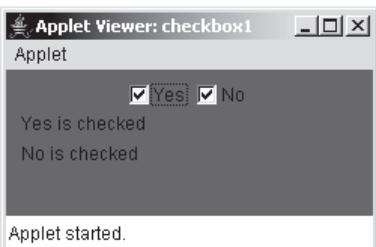
(a) Applet at the beginning



(b) Applet when "Yes" is checked



(c) Applet when "No" is checked



(d) Applet when "Yes and No" are checked

Figure 16.14 Output screen of Program 16.13

Explanation: This program creates two `Checkbox` instances with the name `ch1` and `ch2`. The labels of these are "Yes" and "No". The applet class `checkbox` implements the `ItemListener` interface. Listener for both the checkboxes are added using `addItemListener` method. Whenever any of the checkbox's

state is changed, the `itemStateChanged` method is invoked and an object of `ItemEvent` is passed to it. Inside the method, the `repaint` method causes `paint` to be invoked. In the `paint` method, the state of both the checkboxes is on else, returns false. Depending on the state of checkboxes, appropriate message is displayed using `drawString` method.

```
/*PROG 16.14 IMPLEMENTING CHECKBOX AND ITS EVENT METHODS VER 2*/  
  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
<applet code = "checkbox2" width = 200 height = 150>  
</applet>  
*/  
public class checkbox2 extends Applet implements ItemListener  
{  
    String msg = " ";  
    Checkbox ch1, ch2, ch3, ch4;  
    public void init()  
    {  
        setBackground(Color.red);  
        ch1 = new Checkbox("Sachin", null, true);  
        ch2 = new Checkbox("Saurav");  
        ch3 = new Checkbox("Yuvraj");  
        ch4 = new Checkbox("Dhoni");  
        add(ch1);  
        add(ch2);  
        add(ch3);  
        add(ch4);  
        ch1.addItemListener(this);  
        ch2.addItemListener(this);  
        ch3.addItemListener(this);  
        ch4.addItemListener(this);  
    }  
    public void itemStateChanged(ItemEvent ie)  
    {  
        repaint();  
    }  
    public void paint(Graphics g)  
    {  
        if (ch1.getState())  
            msg = "Sachine is in \n";  
        else  
            msg = "Sachine is out\n";  
        g.drawString(msg, 10, 60);  
        if (ch2.getState())  
            msg = "Saurav is in\n";  
        else  
            msg = "Saurav is out\n";  
        g.drawString(msg, 10, 80);  
    }  
}
```

```

        if (ch3.getState())
            msg = "Yuvraj is in \n";
        else
            msg = "Yuvraj is out\n";
        g.drawString(msg, 10, 100);
        if (ch4.getState())
            msg = "Dhoni is in\n";
        else
            msg = "Dhoni is out\n";
        g.drawString(msg, 10, 120);
    }
}

```

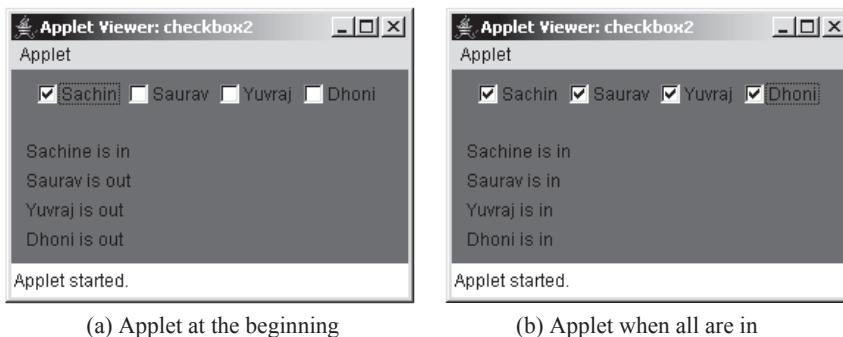


Figure 16.15 Output screen of Program 16.14

Explanation: This program is similar to the previous one with the addition of two more Checkbox instances. Note also that the state of first Checkbox instance is set to true.

16.4.4 The CheckboxGroup Class

The CheckboxGroup class is used to group tighter a set of Checkbox buttons. Exactly one check box button in a CheckboxGroup can be in the “on” state at any given time. Pushing any button, sets its state to “on” and forces any other button in the “on” state into the “off” state. In appearance, they are round shaped, and not square, so sometimes they are also known as radio buttons.

In case of checkboxes any of the checkbox can be set or unset, but after combining a number of checkboxes under one group, only one checkbox can be set.

The class defines just one constructor:

```
public CheckboxGroup()
```

The constructor creates a new instance of CheckboxGroup. The two useful methods of this class are discussed below:

1. **public Checkbox getSelectedCheckbox()**

This method gets the current choice from this check box group. The current choice is the check box in this group that is currently in the “on” state, or null if all check boxes in the group are off.

2. **public void setSelectedCheckbox(Checkbox box)**

This method sets the currently selected check box in this group to be the specified check box. It sets the state of that check box to “on” and all other check boxes in the group to “off”.

```

/*PROG 16.15 IMPLEMENTING CHECKBOXGROUP AND ITS EVENT METHODS
VER 1 */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "checkbox3" width = 200 height = 100>
</applet>
*/
public class checkbox3 extends Applet implements ItemListener
{
    String msg = " ";
    Checkbox ch1, ch2, ch3;
    CheckboxGroup group;
    public void init()
    {
        setBackground(Color.pink);
        group = new CheckboxGroup();
        ch1 = new Checkbox("Yes", group, true);
        ch2 = new Checkbox("No", group, false);
        ch3 = new Checkbox("Cancel", group, false);
        add(ch1);
        add(ch2);
        add(ch3);
        ch1.addItemListener(this);
        ch2.addItemListener(this);
        ch3.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        msg = "You Selected ";
        Checkbox temp = group.getSelectedCheckbox();
        msg = msg + temp.getLabel();
        g.drawString(msg, 10, 50);
    }
}

```

Explanation: In this program, a Checkbox Group instance by the name group is created. Three Checkbox instances ch1, ch2 and ch3 are created and all are made members of the group. Now, out of all three checkboxes, only one can be set any time. In the paint method, the currently selected checkbox instance is obtained using the method getSelectedCheckbox(). This is stored in the temporary reference temp of Checkbox class. The label of this is obtained

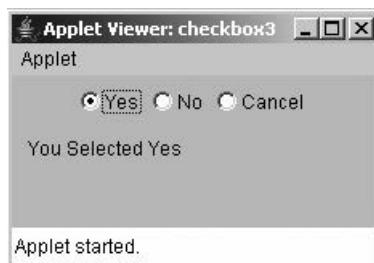
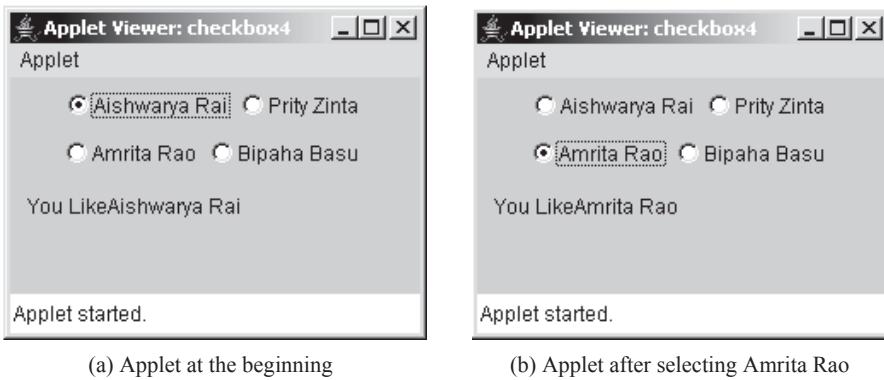


Figure 16.16 Output screen of Program 16.15

using `getLabel` method. This label is then appended to the string “**You Selected**” and displayed using `drawString` method.

```
/*PROG 16.16 IMPLEMENTING CHECKBOX AND ITS EVENT METHODS
VER 2*/
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "checkbox4" width = 200 height = 100>
</applet>
*/
public class checkbox4 extends Applet implements ItemListener
{
    String msg = " ";
    Checkbox ch[];
    CheckboxGroup group;
    public void init(){
        setBackground(Color.LIGHT_GRAY);
        ch = new Checkbox[4];
        group = new CheckboxGroup();
        for (int i = 0; i < ch.length; i++)
        {
            ch[i] = new Checkbox();
            ch[i].setCheckboxGroup(group);
            add(ch[i]);
            ch[i].addItemListener(this);
        }
        ch[0].setLabel("Aishwarya Rai");
        ch[0].setState(true);
        ch[1].setLabel("Prity Zinta");
        ch[2].setLabel("Amrita Rao");
        ch[3].setLabel("Bipaha Basu");
    }
    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        msg = "You Like";
        msg += group.getSelectedCheckbox().getLabel();
        g.drawString(msg, 10, 80);
    }
}
```

**Figure 16.17** Output screen of Program 16.16

Explanation: In this program, an array of checkboxes is created by the name `ch` of size 4. Using `for` loop all array elements are initialized with default constructor of `Checkbox` class. The array elements are assigned the `Checkboxgroup` instance group using the method `setCheckboxGroup`. The labels of each checkbox are set separately using `setLabel` method. The first check box is assumed to be selected. Coding in `paint` method is simple to understand.

16.4.5 The List Class

The `List` component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple ones. Any number of items can be shown at one time. The class provides the following constructors:

1. **public List()**

This form of constructor creates a new scrolling list. By default, there are four visible lines, and multiple selections are not allowed.

2. **public List(int rows)**

This form of constructor creates a new scrolling list initialized with the specified number of visible lines. By default, multiple selections are not allowed.

3. **public List(int rows, boolean msMode)**

This form of constructor creates a new scrolling list initialized to display the specified number of rows. Note that if zero rows are specified, the list will be created with a default of four rows. Also note that the number of visible rows in the list cannot be changed after it has been created. If the value of `msMode` is true, the user can select multiple items from the list. If it is false, only one item at a time can be selected.

There are a number of useful methods of `List` class, which are discussed below:

1. **add**

This method has two forms:

- **public void add(String item)**

This form of `add` method adds the specified item to the end of the scrolling list.

- **public void add(String item, int index)**

This form of `add` method adds the specified item to the scrolling list at the position indicated by the index. The index is zero based. If the value of the index is less than zero, or greater than or equal to the number of items in the list, the item is added to the end of the list.

2. **public String getItem(int index)**
This method gets the item associated with the specified index.
3. **public String[]getItems()**
This method gets the items in the list. It returns a string array containing items of the list.
4. **public int getItemCount()**
This method returns the number of items in the list.
5. **public int getSelectedIndex()**
This method returns the index of the selected item; if no item is selected, or if multiple items are selected, -1 is returned.
6. **public int[] getSelectedIndexes()**
This method returns an array of the selected indexes on the scrolling list; if no item is selected, a zero-length array is returned.
7. **public String getSelectedItem()**
This method returns the selected item on the list; if no item is selected, or if multiple items are selected, null is returned.
8. **public String[]getSelectedItems()**
This method returns an array of the selected items on the scrolling list; if no item is selected, a zero-length array is returned.
9. **public void remove(String item)**
This method removes the first occurrence of an item from the list.
10. **public void remove(int position)**
This method removes the item at the specified position from the scrolling list.
11. **public void removeAll()**
This method removes all items from the list.
12. **public void select(int index)**
This method selects the item at the specified index on the scrolling list.

If an application wants to perform some action based on an item in the list being selected or activated by the user, it should implement `ItemListener` or `ActionListener` as appropriate, and register the new listener to receive events from this list.

When an item is selected or deselected by the user, AWT sends an instance of `ItemEvent` to the list. When the user double-clicks on an item in a scrolling list, AWT sends an instance of `ActionEvent` to the list following the item event. AWT also generates an action event when the user presses the return key while an item in the list is selected.

```
/*PROG 16.17 CREATING LIST AND IMPLEMENTING ITS METHODS VER 1*/
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "list1" width = 200 height = 100">
</applet>
*/
public class list1 extends Applet implements ItemListener
{
    List PL;
    String dispstr = " ";
```

```

public void init()
{
    setBackground(Color.blue);
    PL = new List(4, true);
    PL.add("C++");
    PL.add("VC++");
    PL.add("Java");
    PL.add("C#");
    PL.select(2);
    add(PL);
    PL.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
    repaint();
}
public void paint(Graphics g)
{
    int idx[];
    dispstr = "Selected PL are";
    idx = PL.getSelectedIndexes();
    for(int i = 0;i<idx.length;i++)
        dispstr += PL.getItem(idx[i]) + " , ";
    g.drawString(dispstr, 10, 130);
}
}

```

Explanation: In this program, a List instance PL of four items is constructed with multiple selections allowed. Names of four programming languages are added to the list PL. The third item is shown selected using select(2). The applet window implements ItemListener interface so whenever an item is selected or deselected using mouse click, an ItemEvent instance is sent to the itemStateChanged method. In this method, repaint method causes paint method to be called. In the paint method, an array of selected indexes is obtained using getSelectedIndexes method. Then using this array and getItem method, all selected list items are concatenated to the dispstr. The same is then displayed using drawString method.

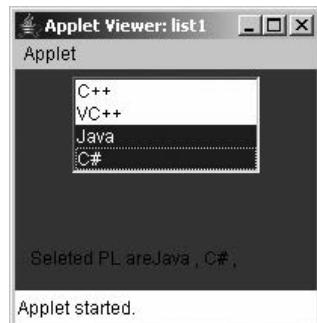


Figure 16.18 Output screen of Program 16.17

```
/*PROG 16.18 CREATING LIST AND IMPLEMENTING ITS METHODS VER 2 */
```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

```

```

/*
<applet code = "list2" width = 200 height = 100>
</applet>
*/
public class list2 extends Applet implements ActionListener
{
    List PL;
    String dispstr = " ";
    public void init()
    {
        setBackground(Color.cyan);
        PL = new List(4, true);
        PL.add("C++");
        PL.add("VC++");
        PL.add("Java");
        PL.add("C#");
        PL.select(2);
        add(PL);
        PL.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        String[] str = null;
        dispstr = "Selected PL are";
        str = PL.getSelectedItems();
        for (int i = 0; i < str.length; i++)
            dispstr += str[i] + ",";
        g.drawString(dispstr, 10, 100);
    }
}

```

Explanation: This program is similar to the previous one but in this case ActionListener interface is implemented and listener is added using addActionListener method. Now, whenever an item in the list is double-clicked, an ActionEvent instance is passed to the method actionPerformed. In this method, all selected entries are returned using getSelectedItems method in the String array str. All array elements are then concatenated to the dispstr and then displayed. Note the same code which had been written in the previous program in the paint method, can be used here too; however, an alternative code has been used.

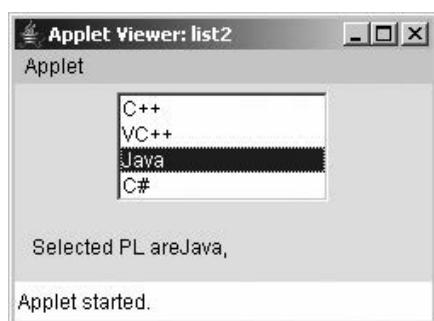


Figure 16.19 Output screen of Program 16.18

```
/*PROG 16.19 CREATING LIST AND IMPLEMENTING ITS METHODS VER 3 */
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "list3" width = 200 height = 200>
</applet>
*/
public class list3 extends Applet implements ActionListener
{
    List OS, PL;
    String dispstr = " ";
    public void init()
    {
        setBackground(Color.RED);
        OS = new List(4, true);
        PL = new List(4, true);
        OS.add("Window XP");
        OS.add("Linux");
        OS.add("Unix");
        OS.add("Window Vista");
        PL.add("C++");
        PL.add("VC++");
        PL.add("Java");
        PL.add("C#");
        PL.select(2);
        OS.select(1);
        add(OS);
        add(PL);
        OS.addActionListener(this);
        PL.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        int idx[];
        dispstr = "Selected OS are";
        idx = OS.getSelectedIndexes();
        for (int i = 0; i < idx.length; i++)
            dispstr += OS.getItem(idx[i]) + ", ";
        g.drawString(dispstr, 10, 100);
        dispstr = "Selected PL are";
        idx = PL.getSelectedIndexes();
        for (int i = 0; i < idx.length; i++)
            dispstr += PL.getItem(idx[i]) + ",";
        g.drawString(dispstr, 10, 120);
    }
}
```

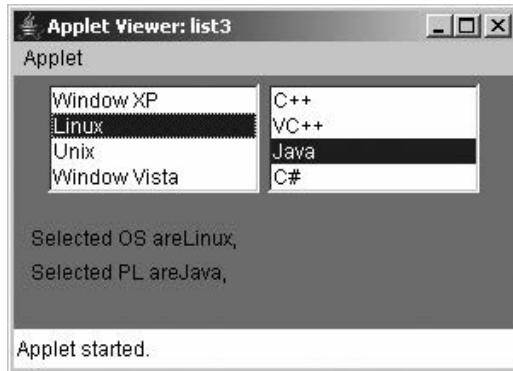


Figure 16.20 Output screen of Program 16.19

Explanation: This program is simple to understand. The only difference it has from previous two programs is that in this case two lists have been worked with.

```
/*PROG 16.20 TRANSFERRING ITEMS BETWEEN LISTS */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "list4" width = 300 height = 100>
</applet>
*/
public class list4 extends Applet implements ActionListener
{
    List L1, L2;
    String dispstr = " ";
    public void init()
    {
        setBackground(Color.PINK);
        L1 = new List(4, false);
        L2 = new List(4, false);
        L1.add("Hari");
        L1.add("Man");
        L1.add("Vijay");
        L1.add("Mohan");
        L2.add("Anjana");
        L2.add("Hemangi");
        L2.add("Malvika");
        L2.add("Vipul");
        add(L1);
        add(L2);
        L1.addActionListener(this);
        L2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
```

```

    {
        if (ae.getSource() == L1)
        {
            L2.add(L1.getSelectedItem());
            L1.remove(L1.getSelectedItem());
        }
        else if (ae.getSource() == L2)
        {
            L1.add(L2.getSelectedItem());
            L2.remove(L2.getSelectedItem());
        }
    }
}

```

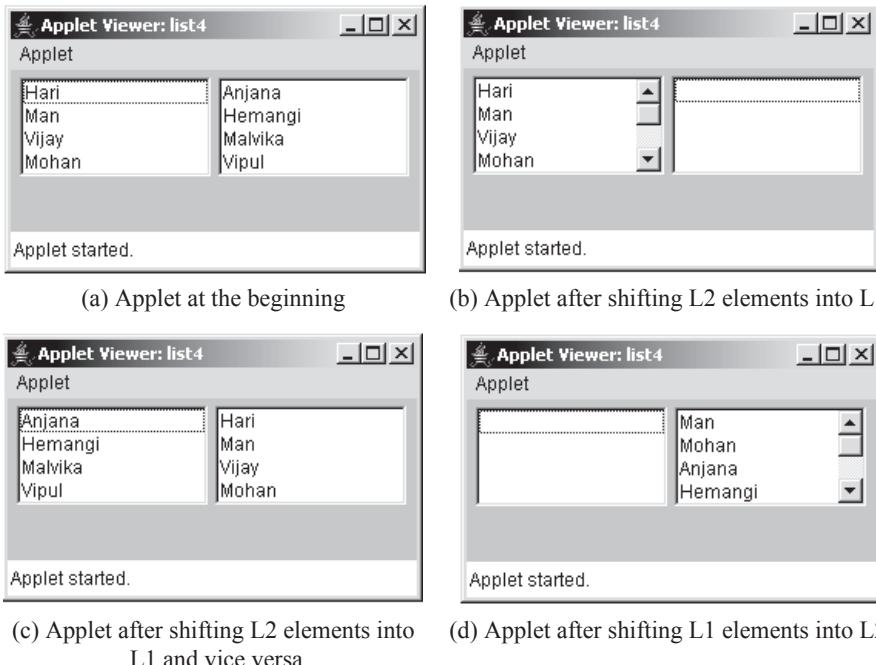


Figure 16.21 Output screen of Program 16.20

Explanation: On double-clicking an item on any of the lists, the item is removed from that list and added to another. For example, when an item is in list L1 and when it is double-clicked, it must be added to list L2 and removed from list L1. In order to identify in which of the lists the item was clicked, the method getSource is used. Assume the list L1 in which an item is double-clicked. The item is added to the list L2 as:

```
L2.add(L1.getSelectedItem());
```

And removed from L1 as:

```
L1.remove(L1.getSelectedItem());
```

16.4.6 The Choice Class

The `Choice` class presents a pop-up menu of choices. The current choice is displayed as the title of the menu. The menu lists a number of items; only the selected item is displayed. However, when the user clicks on the `Choice` component, the entire list is displayed, and the user can select one of the items from the list. There are three types of menu in Java: choice menu, pop-up menu, and pull-down menu. Pop-up and pull-down menus are not components. In fact, `Choice` components are not technically considered to be menus, but it is hard to find another word that adequately describes what they do.

The class defines just one default constructor.

```
public Choice()
```

Various useful methods of `Choice` class are discussed below:

1. **`public void add(String item)`**

This method adds an item to this `Choice` menu.

2. **`public String getItem(int index)`**

This method returns the string at the specified index in this `Choice` menu.

3. **`public int getItemCount()`**

This method returns the number of items in this `Choice` menu.

4. **`public int getSelectedItem()`**

This method returns the index of the currently selected item. If nothing is selected, it returns -1.

5. **`public String getSelectedItem()`**

This method returns a representation of the current choice as a string.

6. **`public void insert(StringItem, int index)`**

This method inserts the item into this choice at the specified position.

7. **`remove`**

- **`public void remove(int position)`**

This method removes an item from the `Choice` menu at the specified position.

- **`public void remove(String item)`**

This method removes the first occurrence of item from the `Choice` menu.

8. **`public void removeAll()`**

This method removes all items from the `Choice` menu.

9. **`select`**

This method has two forms:

- (i) **`public void select(int pos)`**

This method sets the selected item in this `Choice` menu to be the item at the specified position.

- (ii) **`public void select(String str)`**

This method sets the selected item in this `Choice` menu to be the item whose name is equal to the specified string. If more than one item matches (is equal to) the specified string, the one with the smallest index is selected.

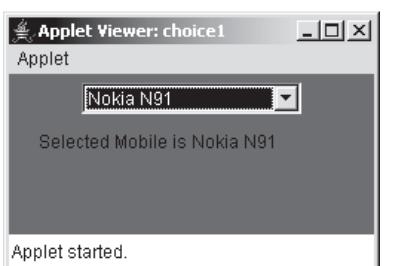
For handling events for `Choice`, the parent class has to implement `ItemListener` interface and listener has to be registered using `addItemListener` method. Whenever an item is selected from the `Choice` instance, event notification is sent to `itemStateChanged` method, as seen earlier with the `List` class.

```
/*PROG 16.21 DEMO OF CHOICE CLASS AND ITS METHOD VER 1*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "choice1" width = 200 height = 100>
</applet>
*/
public class choice1 extends Applet implements ItemListener
{
    Choice MC;
    String dispstr = " ";
    public void init()
    {
        setBackground(Color.red);
        MC = new Choice();
        MC.add("Sony Erricsion W200i");
        MC.add("Nokia N91");

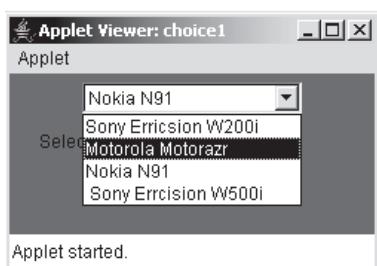
        MC.insert("Motorola Motorazr",1);
        MC.add(" Sony Errcision W500i");

        MC.select(2);

        add(MC);
        MC.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        dispstr = "Selected Mobile is ";
        dispstr += MC.getSelectedItem();
        g.drawString(dispstr, 20, 50);
    }
}
```



(a) Applet at the beginning



(b) Applet with all the choice

Figure 16.22 Output screen of Program 16.21

Explanation: In this program, a Choice instance by the name MC is created. In this instance, few mobile set names are added and inserted. The default choice selected is String object at index 2 using select method. A listener is added for MC using addItemClickListener method. Whenever a new selection from Choice instance MC is made, itemStateChanged method is called and an instance of ItemEvent is passed to it. In this method, repaint method calls paint method. In the paint method, the selected item is retrieved using getSelectedItem method and concatenated to dispstr. The same is displayed using drawString method.

```
/*PROG 16.22 DEMO OF CHOICE CLASS AND ITS METHOD VER 2 */
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code ="choice2" width = 200 height = 200>
</applet>
*/
public class choice2 extends Applet implements ItemListener
{
    Choice PL, OS;
    String dispstr = " ";
    public void init()
    {
        setBackground(Color.PINK);
        OS = new Choice();
        PL = new Choice();

        OS.add("Window XP");
        OS.add("Linux");
        OS.add("Unix");
        OS.add("Window Vista");
        PL.add("C++");
        PL.add("VC++");
        PL.add("Java");
        PL.add("C#");

        PL.select("Java");

        add(OS);
        add(PL);

        OS.addItemListener(this);
        PL.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
```

```

        dispstr = "Selected OS is ";
        dispstr += OS.getSelectedItem();
        g.drawString(dispstr, 20, 60);
        dispstr = "Selected PL is ";
        dispstr += PL.getSelectedItem();
        g.drawString(dispstr, 20, 80);
    }
}

```

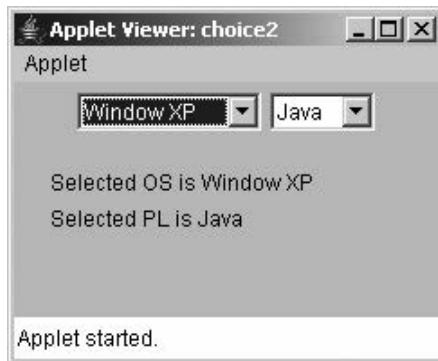


Figure 16.23 Output screen of Program 16.22

Explanation: This program is simple compared to previous one with the difference that two Choice instances are being worked with, instead of one.

16.4.7 The Scrollbar Class

The scrollbar class embodies a scroll bar, a familiar user-interface object. A scroll bar provides a convenient means for allowing a user to select from a range of values. A scroll bar can be either horizontal or vertical. It has five parts as shown in Figure 16.24.

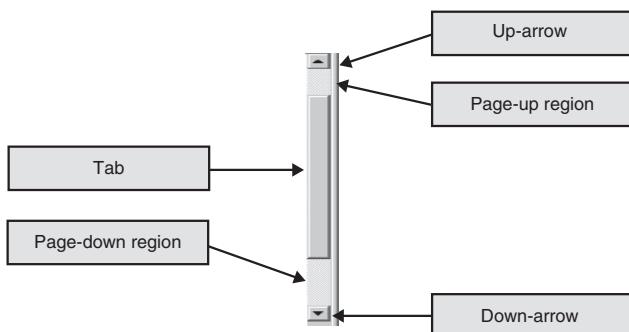


Figure 16.24 Scrollbar and its parts

The position of the tab specifies the currently selected value. The user can move the tab by dragging it or by clicking on any part of the scroll bar. On some platforms, the size of the tab tells what portion of a scrolling region is currently visible. It is actually the position of the bottom or left edge of the tab that represents the currently selected value.

A scroll bar has four associated integer values:

- **min**, which specifies the starting point of a range of values represented by the scroll bar, corresponding to the left or bottom edge of the bar.
- **max**, which specifies the end point of a range of values, corresponding to the right or top edge of the bar.
- **visible**, which specifies the size of the tab.
- **value**, which gives the currently selected value, somewhere in the range between min and max (visible).

The scrollbar class defines the following constructors:

1. **public Scrollbar()**

This form of constructor creates a new vertical scroll bar. The default properties of the scroll bar are listed in Table 16.2.

| Property | Description | Default value |
|-----------------|---|--------------------|
| orientation | Indicates whether the scroll bar is vertical or horizontal. | Scrollbar.VERTICAL |
| value | Controls the location of the scroll bar's bubble. | 0 |
| visible amount | The scroll bar's range, typically represented by the size of the scroll bar's bubble | 10 |
| minimum | The minimum value of the scroll bar. | 0 |
| maximum | The maximum value of the scroll bar | 100 |
| unit increment | Amount the value changes when the Line Up or Line Down key is pressed, or when the end arrows of the scroll bar are clicked. | 1 |
| block increment | Amount the value changes when the Page Up or Page Down key is pressed, or when the scrollbar track is clicked on either side of the bubble. | 10 |

Table 16.2 Scrollbar properties

2. **public Scrollbar(int orientation)**

This form of constructor creates a new scroll bar with the specified orientation. The orientation argument must take one of the two values, Scrollbar.HORIZONTAL or Scrollbar.VERTICAL, indicating a horizontal or vertical scroll bar, respectively.

3. **public Scrollbar(int orientation, int value, int visible, int minimum, int maximum)**

This form of constructor creates a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum value.

The various useful methods of Scrollbar class are discussed below:

1. **public int getBlockIncrement()**

This method gets the block increment of this scroll bar. The block increment is the value that is added or subtracted when the user activates the block increment area of the scroll bar, generally through a mouse or keyboard that the scroll bar receives as an adjustment event.

2. **public void setBlockIncrement()**

This method sets the block increment of this scroll bar.

3. **public int getMaximum()**

This method returns the maximum value of this scroll bar.

4. **public void setMaximum(int newMaximum)**

This method sets the maximum value of this scroll bar. When setMaximum is called, the maximum value is changed, and other values (including the minimum, the visible amount and the current scroll bar value) are changed to be consistent with the new maximum.

5. **public int getMinimum()**

This method returns the minimum value of this scroll bar.

6. **public void setMinimum(int newMinimum)**

This method sets the minimum value of this scroll bar.

7. **public int getOrientation()**

The method returns the orientation of this scroll bar, either Scrollbar.HORIZONTAL or Scrollbar.VERTICAL.

8. **public void setOrientation(int Orientation)**

This method sets the orientation for this scroll bar, either Scrollbar.HORIZONTAL or Scrollbar.VERTICAL.

9. **public int getUnitIncrement()**

This method returns the unit increment for this scrollbar. The unit increment is the value that is added or subtracted when the user activates the unit increment area of the scroll bar, generally through a mouse or keyboard that the scroll bar receives as an adjustment event.

10. **public void setUnitIncrement(int v)**

This method sets the unit increment for this scroll bar.

11. **public int getValue()**

This method returns the current value of this scroll bar.

12. **public void setValue(int newValue)**

This method sets the value of this scroll bar to the specified value.

A scroll bar generates an event of type AdjustmentEvent whenever the user changes the value of the scroll bar. The associated AdjustmentListener interface defines one method, “adjustmentValueChanged(AdjustmentEvent evt)”, which is called by the scroll bar to notify the listener that the value on the scroll bar has been changed. This method should repaint the display or make whatever other change is appropriate for the new value. The method evt.getValue() returns the current value on the scroll bar. If more than one scroll bar are used and it needs to be determined which scroll bar generated the event, evt.getSource() should be used to determine the source of the event.

The getAdjustmentType() of AdjustmentEvent class can be used to find out the type of event occurred on the scroll bar. Its signature is given below:

```
public int getAdjustmentType()
```

This method returns the type of adjustment that caused the value changed event. It will have one of the following values:

- UNIT_INCREMENT
- UNIT_DECREMENT
- BLOCK_INCREMENT
- BLOCK_DECREMENT
- TRACK

```
/*PROG 16.23 DEMO OF SCROLLBAR VER 1*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "scrollbar1" width = 200 height = 100>
</applet>
*/
public class scrollbar1 extends Applet implements
AdjustmentListener
{
    String msg = " ";
    Scrollbar SB;
    public void init()
    {
        setBackground(Color.cyan);
        SB=new Scrollbar(Scrollbar.HORIZONTAL,5,10,5,500);
        SB.setUnitIncrement(5);
        add(SB);
        SB.addAdjustmentListener(this);
    }
    public void adjustmentValueChanged(AdjustmentEvent ae)
    {
        repaint();
    }
    public void paint(Graphics g)
    {
        msg = "Value: " + SB.getValue();
        g.drawString(msg, 30, 50);
    }
}
```

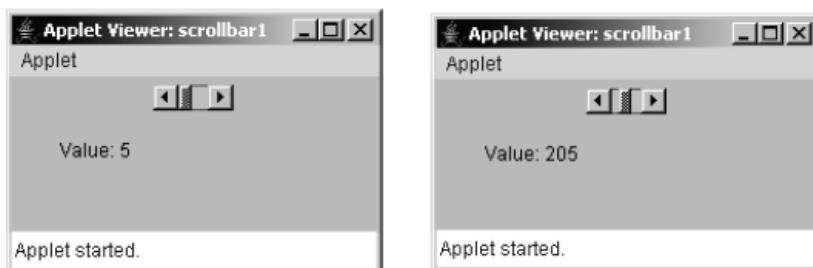


Figure 16.25 Output screen of Program 16.23

Explanation: In this program, AdjustmentListener interface has been implemented in the applet window class scrollbar1. In the init method, a horizontal scroll bar is created with minimum value 5, maximum value 500 and visible amount of thumb size 10. The unit increment of 5 is set. For the Scrollbar instance SB, adjustment listener is added using the method addAdjustmentListener. Whenever either of the arrows or inside the scroll bar area is clicked or the mouse is dragged, the adjustmentValueChanged method is

called. In this method, due to repaint method the paint method is called. In the paint method, current value of SB is shown using drawstring and getValue method.

```
/*PROG 16.24 DEMO OF SCROLLBAR VER2, CONTROLLING MOVEMENT  
OF TEXT */  
  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
<applet code = "scrollbar2" width = 200 height = 300>  
</applet>  
*/  
public class scrollbar2 extends Applet implements  
AdjustmentListener  
{  
    String msg = " ";  
    Scrollbar SB;  
    int X = 10, Y = 70;  
    public void init(){  
        setBackground(Color.pink);  
        SB = new Scrollbar(Scrollbar.VERTICAL,5,10,5, 500);  
        SB.setUnitIncrement(5);  
        add(SB);  
        SB.addAdjustmentListener(this);  
    }  
    public void adjustmentValueChanged(AdjustmentEvent ae)  
{  
        switch (ae.getAdjustmentType()){  
            case AdjustmentEvent.UNIT_INCREMENT:  
                Y += 5;  
                break;  
            case AdjustmentEvent.UNIT_DECREMENT:  
                Y -= 5;  
                break;  
            case AdjustmentEvent.BLOCK_INCREMENT:  
                Y += 25;  
                break;  
            case AdjustmentEvent.BLOCK_DECREMENT:  
                Y -= 25;  
                break;  
        }  
        repaint();  
    }  
    public void paint(Graphics g)  
{  
        msg = "Moving Text you can see";  
        g.setFont(new Font("Calisto MT", Font.BOLD, 20));  
        g.setColor(Color.blue);  
        g.drawString(msg, X, Y);  
    }  
}
```

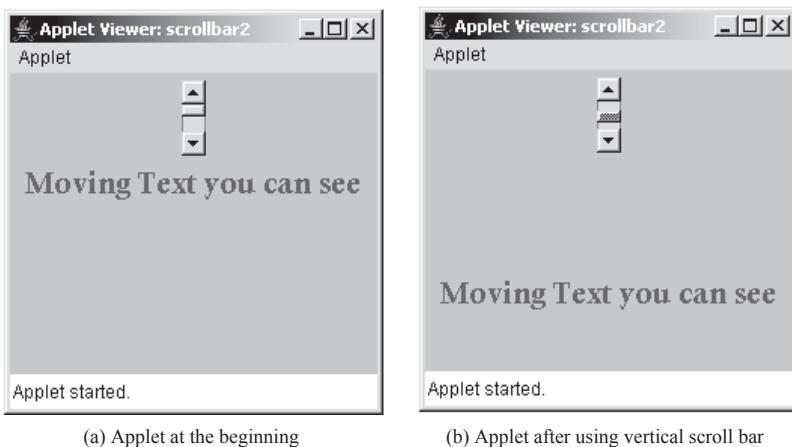


Figure 16.26 Output screen of Program 16.24

Explanation: In the `init()` a vertical scroll bar is created by the name `SB`. The minimum and maximum values for this scroll bar are `5` and `500`, respectively, and unit increment is `5`. In the `adjustmentValueChanged` method, the type of events occurring onto the scroll bar are recognized. For `UNIT_INCREMENT` and `UNIT_DECREMENT` events the value of `Y` was incremented/decremented by `5`. For the `BLOCK_INCREMENT` and `BLOCK_DECREMENT` event, the value of `Y` was incremented/decremented by `25`. The `X` and `Y` coordinates are used for displaying the string “**Moving Text you can see**” using `drawString` method. Each time the down arrow is clicked, values of `Y` increments by `5` and string is displayed at new position. Thus, the string can be moved using events of scroll bar.

16.4.8 The Textfield Class

A `TextField` object is a text component that allows for the editing of a single line of text. The super class for this class is the `TextComponent` class. The users can enter/edit/cut/paste the text of the `TextField` object. It is also possible to set a `TextField` to be read-only so that the user can read the text that it contains but cannot edit it.

The class defines the following constructs:

1. **public TextField()**

This form of constructor creates a new text field.

2. **public TextField(int columns)**

This form of constructor creates a new empty text field with the specified number of columns. The columns specify the number of characters that should be visible in the text field. This is used to determine the width of the text field. (Because characters can be of different sizes, the number of characters visible in the text field might not be exactly equal to columns).

3. **public TextField(String text)**

This form of constructor creates a text field initialized with the specified text.

4. **public TextField(String text, int columns)**

This form of constructor creates a text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

The various useful methods of `TextField`/`TextComponent` class are discussed below:

1. **public String getText()**

This method returns the current text of the `TextField` object.

2. **public void setText(String t)**
This method sets the text that is presented by this text component to be the specified text.
3. **public void select(int sStart, int sEnd)**
This method selects the text between the specified start and end position.
4. **public String getSelectedText()**
This method returns the selected text from the text that is presented by this text component.
5. **public void setEditable(boolean b)**
This method sets the flag that determines whether or not this text component is editable. If the flag is set to true, this text component becomes user editable. If the flag is set to false, the user cannot change the text of this text component.
6. **public boolean isEditable()**
This method indicates whether or not this text component is editable—true if this text component is editable; false otherwise.
7. **public void setEchoChar(char c)**
This method sets the echo character for this text field. An echo character is useful for text fields where user input should not be echoed to the screen, as in the case of a text field for entering a password.
8. **public char getEchoChar()**
This method returns the character that is to be used for echoing.

Every time the user types a key in the text field, one or more key events are sent to the text field. A KeyEvent may be one of three types: keyPressed, keyReleased or KeyTyped. The key event is passed to every keyListener or keyAdapter object which registered to receive such events using the component's addKeyListener method. (KeyAdapter object implements the keyListener interface.)

It is also possible to fire an ActionEvent. If action events are enabled for the text field, they may be fired by pressing the Return key.

```
/*PROG 16.25 DEMO OF TEXTFIELD VER 1, DISPLAYING ENTERED
NAME ON BUTTON CLICK */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "text1" width = 200 height = 100>
</applet>
*/
public class text1 extends Applet implements ActionListener
{
    String msg = " ";
    TextField tf;
    Button b;
    Label L;
    public void init()
    {
        setBackground(Color.cyan);
        L = new Label("Enter your name here",
                     Label.CENTER);
        add(L);
    }
}
```

```

        tf = new TextField(20);
        add(tf);

        b = new Button("Click Me");
        add(b);

        b.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        Graphics g = getGraphics();
        msg = "Hello " + tf.getText();
        g.drawString(msg, 20, 75);
    }
}

```

Explanation: In this program, a Button instance **b**, a Label instance **L** and a TextField instance **tf** have been used. An action listener is added to the Button instance. When the program executes, and user presses the button after entering his/her name into the text box, and `actionPerformed` method is called. In this method, the entered text is retrieved using `getText` method and is appended to the `msg` string. The `msg` is then displayed.

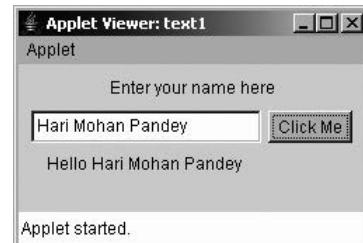


Figure 16.27 Output screen of Program 16.24

```
/*PROG 16.25 DEMO OF TEXTFIELD VER 2, DISPLAYING ENTERED NAME AND  
PASSWORD WITH EVENT HANDLED BY TEXTFIELD BY TEXTFIELD ITSELF*/
```

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "text2" width = 200 height = 200>
</applet>
*/
public class text2 extends Applet implements ActionListener
{
    TextField TN, TP;
    public void init()
    {
        setBackground(Color.red);
        Label LN = new Label("Name: ");
        Label LP = new Label("Password: ");
        TN = new TextField(15);
        TP = new TextField(10);
        TP.setEchoChar('*');

        add(LN);

```

```

        add(TN);
        add(LP);
        add(TP);

        TN.addActionListener(this);
        TP.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae)
    {
        repaint();
    }

    public void paint(Graphics g)
    {
        g.drawString("Name: " + TN.getText(), 10, 70);
        g.drawString("Password: " + TP.getText(), 10, 90);
    }
}

```

Explanation: In this program, two labels and two text boxes are created for name and password. For the second TextField instance TP, the echo character “*” is set. Now whatever character is typed in this text box, ‘*’ is displayed. After entering the text in any of the text boxes when the user presses the Enter key, ActionEvent is generated and actionPerformed method is called. In this method, the entered text is retrieved and displayed.

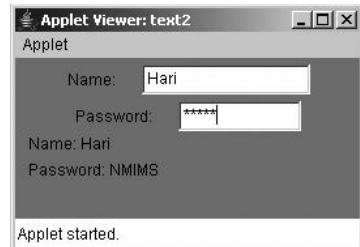


Figure 16.28 Output screen of Program 16.25

```

/*PROG 16.26 MAXIMUM OF TWO NUMBERS */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "text3" width = 200 height = 200>
</applet>
*/
public class text3 extends Applet implements ActionListener
{
    Button but1;
    String result = " ";
    boolean in = false;
    int num1, num2;
    TextField tf1, tf2;
    Label L1, L2;
    public void init()
    {
        setBackground(Color.green);
        L1 = new Label(" Enter first number: ");

```

```

        add(L1);
        tf1 = new TextField(10);
        add(tf1);
        L2 = new Label(" Enter second number: ");
        add(L2);
        tf2 = new TextField(10);
        add(tf2);
        but1 = new Button("Click For Maximum ");
        add(but1);
        but1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent AE) {
        try {
            num1 = Integer.parseInt(tf1.getText());
            num2 = Integer.parseInt(tf2.getText());
            int ans = num1 > num2 ? num1 : num2;
            result = result.valueOf(ans);
            result = "Maximum is:= " + result;
        }
        catch (Exception E) {
            result = "Error!!!!";
        }
        repaint();
    }
    public void paint(Graphics g){
        g.setColor(Color.blue);
        g.drawString(result, 120, 120);
    }
}

```

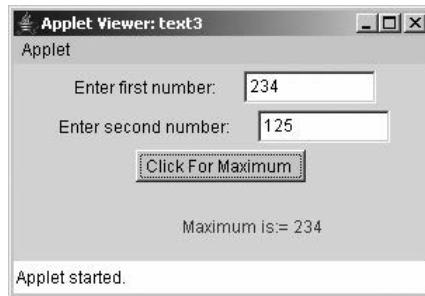


Figure 16.29 Output screen of Program 16.26

Explanation: In the actionPerformed method, the two numbers entered into the two textboxes are first converted into integers. Maximum of two numbers are then stored in the variable ans. The int variable ans is then converted into the String using valueOf() method.

16.4.9 The TextArea Class

A TextArea object is a multi-line region that displays text. It can be set to allow editing or as read-only. A TextArea displays multiple lines and might include scroll bars that the user can use to scroll through the entire contents of the TextArea. Similar to TextField class, TextArea also has TextComponent class as its parent class.

The class defines following constructors:

1. **public TextArea()**

This form of constructor creates a new text area with the empty string as text.

2. **public TextArea(String text)**

This form of constructor creates a new text area with the specified text.

3. **public TextArea(int row, int columns)**

This form of constructor creates a new text area with the specified number of rows and columns and the empty string as text.

4. **public TextString(String text, int rows, int columns)**

This form of constructor creates a new text area with the specified text, and with the specified number of rows and columns.

5. **public TextArea(String text, int rows, int columns, int scrollbars)**

This form of constructor creates a new text area with the specified text, and the rows, columns and scroll bar visibility as specified. All TextArea constructors refer to this one. The TextArea class defines several constants that can be supplied as values for the scrollbars argument:

- SCROLLBARS_BOTH
- SCROLLBARS_VERTICAL_ONLY
- SCROLLBARS_HORIZONTAL_ONLY
- SCROLLBARS_NONE

As TextComponent is also the parent class of TextArea so number of methods of Text Component can be used with TextArea class. They are:

```
getText(), setText(), getSelectedText(), select(), isEditable(), setEditable()
```

Some new methods of TextArea class are discussed below:

1. **public void append(String str)**

This method appends the given text to the text area's current text.

2. **public void insert(String str, int pos)**

This method inserts the specified text at the specified position in this text area.

3. **public void replaceRange(String str, int start, int end)**

This method replaces text between the indicated start and end positions with the specified replacement text. The text at the end position will not be replaced. The text at the start position will be replaced (unless the start position is the same as the end position). The text position is zero based. The inserted substring may be of a different length than the text it replaces.

```
/*PROG 16.27 DEMO OF TEXTAREA CLASS */
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code ="textareal" width = 200 height =100>
</applet>
*/
public class textareal extends Applet implements
ActionListener
{
```

```

TextArea text;
Button b;
public void init()
{
    setBackground(Color.green);
    String mstr = "There are 3 ways to success\n"+
                  "First is to go on \n"+
                  "Second is to go on \n"+
                  "And Third is to go on\n"+
                  "\n Life is like an ice cream\n"+
                  "eat it before it melts\n";
    text = new TextArea(mstr, 5, 30);
    add(text);
    b = new Button("Select some text & click me");
    add(b);
    b.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
    repaint();
}
public void paint(Graphics g)
{
    String str = text.getSelectedText();
    g.drawString("Selected Text: "+str, 10, 145);
}
}

```

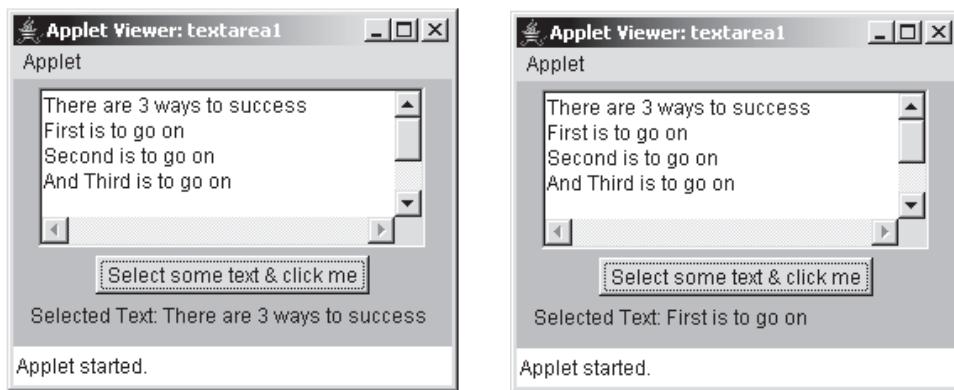


Figure 16.30 Output screen of Program 16.27

Explanation: The `TextArea` instance does not generate any events other than focus events. In this program, a `TextArea` instance `text` is created and some text added to it. The rows and columns for this `TextArea` instance `text` are 5 and 30, respectively. After selecting some text from the contents of text area, the button is pressed, `actionPerformed` method is called. In the method, call to `repaint` method causes `paint` method to be called. In the `paint`, selected text is retrieved using `getSelectedText` method. The same is displayed using `drawString` method.

16.5 MENU AND MENUBARS

A menu is a list of choices. A menu bar displays a list of top-level menu choices. In Java, for implementing menu, a number of classes are provided like Menu, MenuBar and MenuItem.

A Menubar contains a number of instances of Menu each of which further contains a number of instances of MenuItem. Only Frames can have menu since they implement the MenuContainer interface. Therefore, an applet must create a frame in order to use menu.

A Menu class has MenuItem as its super class. A Menu object is a pull-down menu component that is deployed from a menu bar. Each item in a menu must belong to the MenuItem class. It can be an instance of MenuItem, a submenu (an instance of Menu), or a checkbox (an instance of Checkbox MenuItem). A CheckboxMenuItem instance will have a check mark next to it when it is selected.

The Menu class defines the following constructors:

1. **public Menu()**

This form of constructor creates a new menu with an empty label.

2. **public Menu(String label)**

This form of constructor creates a new menu with the specified label.

In both the forms, the menu is not tear-off menu.

3. **public Menu(String label, boolean tearOff)**

This form of constructor creates a new menu with the specified label, indicating whether the menu can be torn off.

After a Menu instance has been created, MenuItem instance can be added to it. All items in a menu must belong to the class MenuItem, or one of its subclasses. The default MenuItem object embodies a simple labelled menu item.

The MenuItem class defines the following constructors:

1. **public MenuItem()**

This form of constructor creates a new MenuItem with an empty label and no keyboard shortcut.

2. **public MenuItem(String label)**

This form of constructor creates a new MenuItem with the specified label and no keyboard shortcut.

3. **public MenuItem(String label, MenuShortcut s)**

This form of constructor creates a menu item with an associated keyboard shortcut.

Once MenuItem instances have been created using one of the constructor forms, they can be added to the Menu instance using add method.

Checkable menu items can also be created. To do this, there is a subclass of the MenuItem named CheckboxMenuItem. This class represents a check box that can be included in a menu. Selecting the check box in the menu changes its state from “on” to “off” or from “off” to “on”.

The CheckboxMenuItem class defines following constructors:

1. **public CheckboxMenuItem()**

This method creates a check box menu item with an empty label. The item's state is initially set to “off”.

2. **public CheckboxMenuItem(String label)**

This method creates a checkbox menu item with the specified label. The item's state is initially set to “off”.

3. **public CheckboxMenuItem(String label, boolean state)**

This method creates a check box menu item with the specified label and state.

To place the Menu instances on the MenuBar instance, it has to be created first. A MenuBar instance can be created with its default constructor form:

```
public MenuBar()
```

A menu bar is attached to the frame window using `setMenuBar` method.

```
/*PROG 16. 28 CREATING MENU AND MENUBAR WITH FRAME VER 1 */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    String msg = " ";
    MyFrame(String title){
        super(title);
        //Menu bar created
        MenuBar mbar = new MenuBar();

        //Menu m is created by name File
        Menu m = new Menu ("File");
        //Three MenuItem are created
        m.add(new MenuItem("New"));
        m.add(new MenuItem("Open"));
        m.add(new MenuItem("Save"));

        //A separator is created (a visible line)
        m.addSeparator();

        //One more MenuItem
        m.add(new MenuItem("Quit"));

        //Menu is attached to menu bar
        mbar.add(m);

        //Menu bar is attached to frame window
        setMenuBar(mbar);
    }
}
public class menu{
    public static void main(String[]args){
        Frame f = new MyFrame("Menu Demo Ver1");
        f.setSize(200, 170);
        f.setVisible(true);
    }
}
```

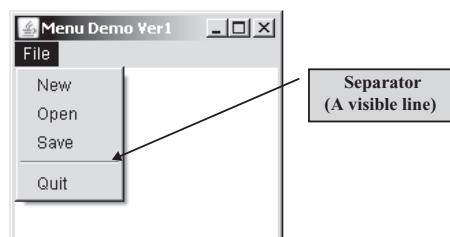


Figure 16.31 Output screen of Program 16.28

Various methods of Menu related classes are discussed below:

1. **public void setEnabled(boolean b)**

This method sets whether or not this menu item can be chosen, that is, can be enabled or disabled.

2. **public boolean isEnabled()**

This method checks whether this menu item is enabled.

3. **public String getLabel()**

This method gets the label for this menu item to the specified label.

4. **public void setLabel(String label)**

This method sets the label for this menu item to the specified label.

5. **public MenuItem add(MenuItem item)**

This method adds the item to the specified Menu instance and returns the same.

6. **public boolean getState()**

This method determines whether the state of this check box is “on” or “off”.

7. **public void setState(boolean b)**

This method sets this check box menu item to the specified state. The Boolean value true indicates “on” while false indicates “off”.

Methods 6 and 7 belong to CheckboxMenuItem class.

Whenever a menu item from a menu is selected, an instance of ActionEvent is generated and passed to the method actionPerformed. For CheckboxMenuItem, an instance of ItemEvent is generated and passed to itemStateChanged method. For the events, the frame class must implement ActionListener and Item Listener interface. Note events are only generated for menu items and not for frame.

```
/*PROG 16. 29 CREATING MENU AND MENUBAR WITH FRAME VER 2 */
```

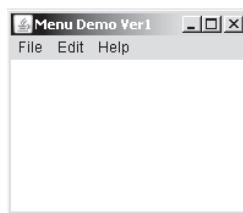
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    String msg = " ";
    MyFrame(String title)
    {
        super(title);
        //Menu bar created
        MenuBar mbar = new MenuBar();
        //Menu m is created by name File
        Menu m = new Menu("File");
        //Three MenuItem's are created
        m.add(new MenuItem("New"));
        m.add(new MenuItem("Open"));
        m.add(new MenuItem("Save"));
        //A separator is created (a visible line)
        m.addSeparator();
        //One more MenuItem
        m.add(new MenuItem("Quit"));
        //Menu is attached to menu bar
        mbar.add(m);
```

```

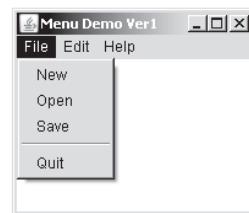
        m = new Menu("Edit");
        m.add(new MenuItem("Cut"));
        m.add(new MenuItem("Copy"));
        m.add(new MenuItem("Paste"));
        m.addSeparator();
        m.add(new MenuItem("Select All"));
        mbar.add(m);

        m = new Menu("Help");
        m.add(new MenuItem("Index"));
        m.addSeparator();
        m.add(new MenuItem("About"));
        mbar.add(m);
        //Menu bar is attached to frame window
        setMenuBar(mbar);
        addWindowListener(new MWA());
    }
    class MWA extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            setVisible(false);
            System.exit(0);
        }
    }
}
public class menu2{
    public static void main(String[] args)
    {
        Frame f = new MyFrame("Menu Demo Ver1");
        f.setSize(200, 170);
        f.setVisible(true);
    }
}
}

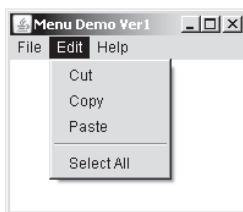
```



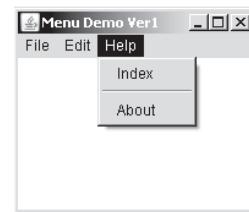
(a) Frame at the beginning



(b) Frame showing "File" option



(b) Frame showing "Edit" option



(b) Frame showing "Help" option

Explanation: In this program, three different types of menus are created. For that just one `Menu` instance `m` is created. For the first menu labelled “**File**”, three `MenuItem`s are added, then a separator and one more `MenuItem` instance is added. Similarly, for other menus “**Edit**” and “**Help**” different `MenuItem` instances have been created. Note after creation of every menu with menu items they have been added to menu bar using `add` method. In the end, the `MenuBar` instance menu bar is attached to the menu using `setMenuBar` method.

For closing window event, window adapter class is created as an inner class. This has been explained in the previous chapter. In the main class menu, an instance of `MyFrame` class is created and “**Menu Demo Ver1**” passed as title of the **Frame** window. Note the code for the event handling for the selection of the menu items is not shown. This is shown in the next program.

```
/*PROG 16.30 CREATING MENU AND MENUBAR WITH FRAME VER 3 */
```

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        MenuBar mbar = new MenuBar();
        Menu colors = new Menu("Demo");

        MenuItem item1 = new MenuItem("Red");
        MenuItem item2 = new MenuItem("Blue");
        MenuItem item3 = new MenuItem("Green");
        MenuItem item4 = new MenuItem("Black");
        colors.add(item1);
        colors.add(item2);
        colors.add(item3);
        colors.add(item4);
        mbar.add(colors);
        setMenuBar(mbar);
        item1.addActionListener(new MMH(this));
        item2.addActionListener(new MMH(this));
        item3.addActionListener(new MMH(this));
        item4.addActionListener(new MMH(this));
        addWindowListener(new MWA());
    }
    class MWA extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            setVisible(false);
            System.exit(0);
        }
    }
    class MMH implements ActionListener
    {
```

```
MyFrame MF;
public MMH(MyFrame mf) {
    MF = mf;
}
public void actionPerformed(ActionEvent ae)
{
    String arg = (String)ae.getActionCommand();
    if (arg.equals("Red"))
        MF.setBackground(Color.red);
    else if (arg.equals("Green"))
        MF.setBackground(Color.green);
    else if (arg.equals("Blue"))
        MF.setBackground(Color.blue);
    else if (arg.equals("Black"))
        MF.setBackground(Color.black);
    MF.repaint();
}
public class menu3{
    public static void main(String[] args)
    {
        Frame f = new MyFrame("Menu Demo");
        f.setSize(250, 250);
        f.setVisible(true);
    }
}
```

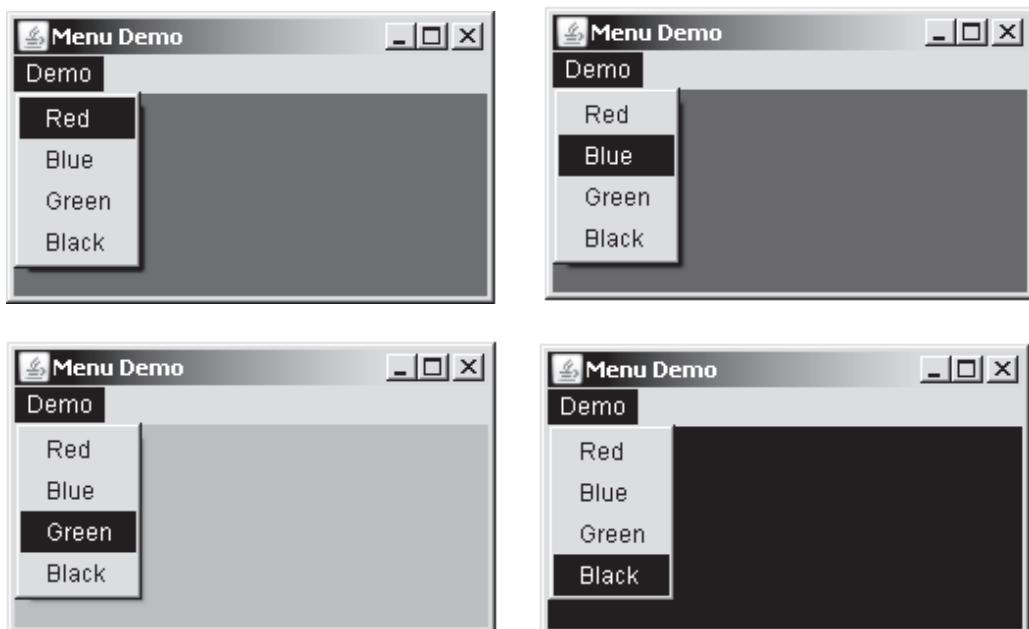


Figure 16.32 Output screen of Program 16.30

Explanation: Whenever a menu item is selected ActionEvent is generated and an instance of it is passed to actionPerformed method. In the MyFrame constructor, three listeners are added for all four menus: item1, item2, item3 and item4. Note that a separate class MMH is created that implements ActionListener interface. In the constructor of MyFrame when listener is added for the menus using addActionListener, as an argument, an instance of MMH class is passed. The constructor of MMH class takes a reference of the MyFrame class. In the actionPerformed method, the label of the currently selected menu item is retrieved using getActionCommand and stored in the arg, which is compared with any of the label “Red”, “Green”, “Blue” and “Black”. Depending on which menu item was selected, the background color of the frame window is set to that color.

16.5.1 Adding Shortcut Key to Menu

Shortcut keys to menu items can be added using the MenuShortcut class. The MenuShortcut class represents a keyboard accelerator for a MenuItem. Menu shortcuts are created using virtual keycodes, not characters. The class provides two constructors for creating shortcuts:

1. public MenuShortcut(int key)
2. public MenuShortcut(int key, boolean Spress)

The first form of constructor creates a new MenuShortcut for the specified virtual keycode. The key is the virtual keycode that would be returned using KeyEvent class if this key were pressed.

In the second form, if Spress is true, the shift key has to be pressed along with the Strl modifier.

Both the forms can be used as:

```
MenuShortcut ms = new MenuShortcut(KeyEvent.VK_R);
```

If item is the MenuItem created earlier, the above created shortcut can be set using setShortcut method as:

```
item.setShortcut(ms);
```

Now, the MenuItem item can be selected using **Ctrl+R** key. In case the shortcut is created as:

```
MenuShortcut ms = new MenuShortcut(KeyEvent.VK_R, true);
```

Then **Ctrl+Shift+R** key has to be pressed to select the item.

```
/*PROG 16.31 DEMO OF ADDING SHORTCUT KEYS TO MENUS */
```

```
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        MenuBar mbar = new MenuBar();
        Menu shapes = new Menu("Demo");
        MenuItem item1 = new MenuItem("Circle");
        MenuItem item2 = new MenuItem("Rectangle");
        MenuItem item3 = new MenuItem("Ellipse");

        shapes.add(item1);
        shapes.add(item2);
        shapes.add(item3);
        mbar.add(shapes);
```

```
        MenuShortcut ms1 = new MenuShortcut(KeyEvent.VK_C);
        MenuShortcut ms2 = new MenuShortcut(KeyEvent.VK_R,
                                         true);
        MenuShortcut ms3 = new MenuShortcut(KeyEvent.VK_E);

        item1.setShortcut(ms1);
        item2.setShortcut(ms2);
        item3.setShortcut(ms3);
        setMenuBar(mbar);

        item1.addActionListener(new MMH(this));
        item2.addActionListener(new MMH(this));
        item3.addActionListener(new MMH(this));

        addWindowListener(new MWA());
    }
    class MWA extends WindowAdapter {
        public void windowClosing(WindowEvent we)
        {
            setVisible(false);
            System.exit(0);
        }
    }
}
class MMH implements ActionListener{
    MyFrame MF;
    public MMH(MyFrame mf){
        MF = mf;
    }
    public void actionPerformed(ActionEvent ae){
        String arg = (String)ae.getActionCommand();
        Graphics g = MF.getGraphics();
        g.setColor(Color.blue);
        if (arg.equals("Circle"))
            g.drawOval(150, 100, 50, 50);
        else if (arg.equals("Rectangle"))
            g.drawRect(30, 60, 40, 60);
        else if (arg.equals("Ellipse"))
            g.drawOval(90, 80, 40, 30);
    }
}
public class menu4{
    public static void main(String args[]){
        Frame f = new MyFrame("Setshortcut demo");
        f.setSize(250, 250);
        f.setVisible(true);
    }
}
```

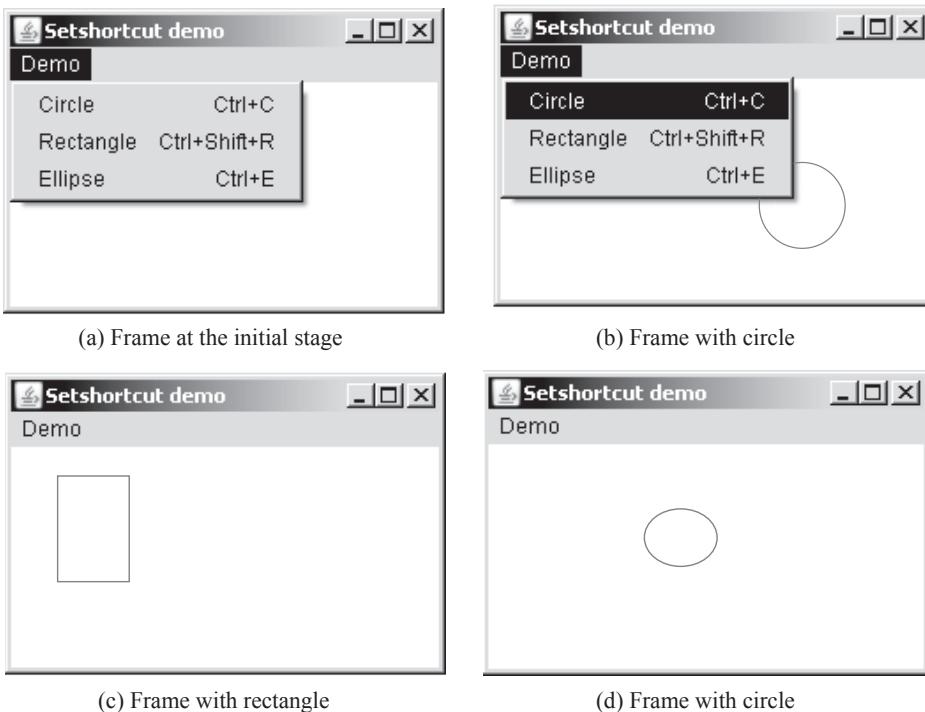


Figure 16.33 Output screen of Program 16.31

Explanation: The program is similar to the previous one but here shortcuts for menu items have been added. The menu items are for drawing various shapes. Creation of menu shortcut and adding them to menu items is discussed in theory part.

16.6 POPUP MENUS

A Popup menu is a menu which can be dynamically popped up at a specified position within a component. It is implemented in java by class `PopupMenu`. Popup menus differ from all the components discussed above because they are not components and they are not usually visible. The user calls up a popup menu by performing some platform-dependent action with the mouse. For example, this might mean clicking with the right mouse button, or clicking the mouse while holding down the control key.

A pop-up menu is an object belonging to the class `PopupMenu`. A newly created pop-up menu is empty. Items can be added to the menu with its `add(String)` method. A separator line can be added with the `addSeparator()` method (There's a lot more one can do with menu items). If `pmenu` is a pop-up menu, it can be added to a component, `comp`, by calling `comp.add(pmenu)`. However, this does not make the menu appear on the screen. To make a menu appear, a program has to call `pmenu.show(comp, x, y)`. The pop-up menu appears with its upper-left corner at the point (x, y) , where the coordinates are given in `comp`'s coordinate system.

The class defines following constructors:

1. **public PopupMenu()**
2. **public PopupMenu(String label)**

This form of constructor creates a new popup menu with an empty name.

This form of constructor creates a new popup menu with the specified name.

It is not much more difficult to use popup menus in a program than it is to use the above components. A popup menu generates an `ActionEvent` when the user selects a command from the menu. The action command associated with that event is the label of the menu item that was selected. One can register an `ActionListener` with the menu to listen for commands from the menu.

Mouse events have to be listened from the component. A `MouseEvent`, `evt`, has a boolean valued method, `evt.isPopupTrigger()` that one can call to determine whether the user is trying to pop up a menu. This could theoretically occur in either the `mousePressed` or in the `mouseReleased` method, so one should test for the pop-up trigger in both of these methods. The `mousePressed` method might look something like this (`mouseReleased` would be similar):

```
public void mousePressed(MouseEvent evt)

{
    if(evt.isPopupTrigger())
    {
        int x = evt.getX();
        int y = evt.getY();
        pmenu.show(this, x, y);
    }
}
```

Note that this method just make the menu appear on the screen. Any command generated by that menu must be handled elsewhere, in an `actionPerformed` method.

```
/*PROG 16.32 DEMO OF POPUP MENU */

import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame implements MouseListener
{
    PopupMenu colors;
    MyFrame(String title)
    {
        super(title);
        colors = new PopupMenu();
        MenuItem item1 = new MenuItem("Red");
        MenuItem item2 = new MenuItem("Blue");
        MenuItem item3 = new MenuItem("Green");

        colors.add(item1);
        colors.add(item2);
        colors.add(item3);

        add(colors);

        colors.addActionListener(new MMH(this));
        addWindowListener(new MWA());
        addMouseListener(this);
    }
    public void mousePressed(MouseEvent evt)
    {
        if (evt.isPopupTrigger())
```

```
{  
    int x = evt.getX();  
    int y = evt.getY();  
    colors.show(this, x, y);  
}  
}  
public void mouseReleased(MouseEvent evt)  
{  
    if (evt.isPopupTrigger())  
    {  
        int x = evt.getX();  
        int y = evt.getY();  
        colors.show(this, x, y);  
    }  
}  
public void mouseExited(MouseEvent me) {}  
public void mouseClicked(MouseEvent me) {}  
public void mouseEntered(MouseEvent me) {}  
  
class MWA extends WindowAdapter  
{  
    public void windowClosing(WindowEvent we)  
    {  
        setVisible(false);  
        System.exit(0);  
    }  
}  
}  
class MMH implements ActionListener  
{  
    MyFrame MF;  
    public MMH(MyFrame mf)  
    {  
        MF = mf;  
    }  
    public void actionPerformed(ActionEvent ae)  
    {  
        String arg = (String)ae.getActionCommand();  
        if (arg.equals("Red"))  
            MF.setBackground(Color.red);  
        else if (arg.equals("Green"))  
            MF.setBackground(Color.green);  
        else if (arg.equals("Blue"))  
            MF.setBackground(Color.blue);  
        MF.repaint();  
    }  
}  
public class menu5  
{  
    public static void main(String[] args)  
    {  
}
```

```

        Frame f = new MyFrame("Demo Popup menu");
        f.setSize(200, 200);
        f.setVisible(true);
    }
}

```

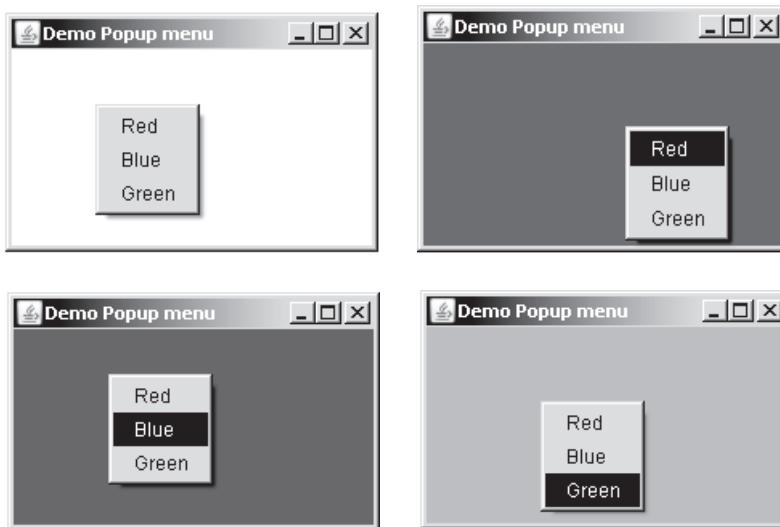


Figure 16.34 Output screen of Program 16.32

Explanation: The code in the constructor of `MyFrame` is easy to understand. An empty `PopupMenu` instance `colors` is created. Number of menu items are added to this `PopupMenu` instance `colors`. After adding menu items to `colors`, it is added to the frame window. Note `actionListener` is added only to the `PopupMenu` instance `colors` and not to all menu items. A mouse listener is also added to the frame window. Whenever the mouse pressed or mouse released event occurs, an instance of `MouseEvent` is passed to the `mouseReleased` and `mousePressed` method. Inside these methods, using `isPopupTrigger` method, it is checked whether or not this mouse event is the popup menu trigger event for the platform. If it is, then popup menus is shown at the current mouse position. When the popup menu appears during execution, any of the menu items can be selected. This works same as seen earlier, difference is only that menu is not attached to the menu bar.

16.7 DIALOGS

Like a frame, a dialog box is a separate window. Unlike a frame, however, a dialog box is not completely independent. Every dialog box is associated with a frame, which is called its parent. The dialog box is dependent on its parent. For example, if the parent is closed, the dialog box will also be closed.

Dialog boxes can be either modal or modeless. When a modal dialog is created, its parent frame is blocked. That is, the user will not be able to interact with the parent or even bring the parent to the front of the screen until the dialog box is closed. Modeless dialog boxes do not block their parents in the same way, so they seem a lot more like independent windows.

In Java, a dialog box is an object belonging to the class `Dialog` or to one of its subclasses. A `Dialog` cannot have a menu bar, but it can contain other GUI components, and it generates the same `WindowEvents`

as a Frame. Dialog objects can be either modal or modeless. A parameter to the constructor specifies which it should be. Most commonly used constructor of the Dialog class is shown below:

1. **public Dialog(Frame owner)**

This form of constructor creates an initially invisible, non-modal Dialog with an empty title and the specified owner frame.

2. **public Dialog(Frame owner, boolean modal)**

This form of constructor creates an initially visible Dialog with an empty title, the specified owner frame and modality. If modal is true the dialog box is Modal and if false then it is false.

3. **public Dialog(Frame owner, String title, boolean modal)**

This form of constructor creates an initially invisible Dialog with the specified owner frame, title, and modality.

When a dialog box is closed, `dispose()` method must be called. The method is defined by parent of the Dialog class i.e. Window class. Its signature is shown below:

```
public void dispose()
```

The method releases all of the native screen resources used by this Window, its subcomponents, and all of its owned children. That is, the resources for these Component will be destroyed, any memory they consumes will be returned to the OS, and they will be marked as undisplayable.

```
/*PROG 16.33 DEMO OF DIALOG VER 1*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class dialog extends Dialog implements ActionListener
{
    dialog(Frame parent, String title)
    {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);
        Button but1 = new Button("Close");
        add(but1);
        but1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        dispose();
        System.exit(0);
    }
}
class MyFrame extends Frame
{
    public static void main(String[] args)
    {
        dialog d = new dialog(new MyFrame(), "Demo Dialog");
        d.setVisible(true);
    }
}
```

Explanation: In the program class dialog extends Dialog class and implements the ActionListener interface. The interface is needed as a button is created on to the dialog window. The constructor of the dialog class takes two parameter of Frame and String type respectively. Using super the constructor of parent class Dialog is called and three parameters are passed. The two are from constructor of dialog class and third parameter being false represents a modeless dialog box. The default layout for a Dialog class is BorderLayout, so it is changed to FlowLayout using setLayout method. A button to the dialog window and also a listener using addActionListener method are added. In the actionPerformed dispose method is called for closing the dialog and to terminate the program System.exit(0) is called.

The class MyFrame extends the Frame class. In the main method, an object d of dialog class is created using the constructor. In the constructor, an instance of MyFrame class is passed and String “**Demo Dialog**” that denotes title of the dialog window. The dialog window must be shown by a call to setVisible method and passing argument true.



Figure 16.35 Output screen of Program 16.33

```
/*PROG 16.34 DEMO OF DIALOG VER 2 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "dialog1" width = 200 height = 200>
</applet>
*/
class dialog extends Dialog implements ActionListener
{
    dialog(Frame parent, String title)
    {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);
        Button but1 = new Button("Close");
        add(but1);
        but1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        dispose();
        System.exit(0);
    }
}
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        setSize(200, 200);
    }
}
```

```

        }
    }

public class dialog1 extends Applet{
    public void init()
    {
        setBackground(Color.pink);
        dialog d = new dialog(new MyFrame("Hello"), "First");
        d.setVisible(true);
    }
}

```

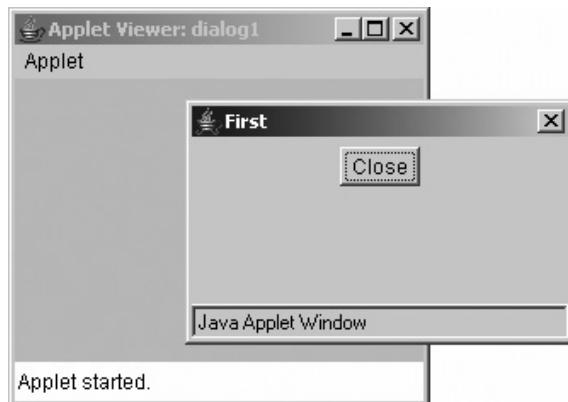


Figure 16.36 Output screen of Program 16.34

Explanation: The program is similar to previous one with the difference that instead of main method, applet window is used. In the program, there is one additional class dialog1 that denotes our applet window. The coding which was in the main in the previous program is now in init method of dialog1 class.

```

/*PROG 16.35 DEMO OF DIALOG VER 3 */

import java.awt.*;
import java.awt.event.*;
class dialog extends Dialog implements ActionListener
{
    dialog(Frame parent, String title)
    {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(150, 100);
        add(new Label("Press OK to exit", Label.CENTER));
        Button but1 = new Button("OK");
        add(but1);
        but1.addActionListener(this);
        Button but2 = new Button("Cancel");
        add(but2);
    }
}

```

```
        but2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String arg = (String)ae.getActionCommand();
        if (arg.equals("OK"))
            dispose();
    }
}
class MyFrame extends Frame implements ActionListener
{
    MyFrame(String title)
    {
        super(title);
        MenuBar mbar = new MenuBar();
        Menu m = new Menu("Demo");
        MenuItem item1 = new MenuItem("Show Dialog");
        m.add(item1);
        mbar.add(m);
        setMenuBar(mbar);
        item1.addActionListener(this);
        addWindowListener(new MWA());
    }
    class MWA extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            setVisible(false);
            System.exit(0);
        }
    }
    public void actionPerformed(ActionEvent ae)
    {
        String arg = (String)ae.getActionCommand();
        if (arg.equals("Show Dialog"))
        {
            dialog d=new dialog(new MyFrame("Hello"),"First");
            d.setVisible(true);
        }
    }
}
public class dialog2
{
    public static void main(String[] args)
    {
        Frame f = new MyFrame("Menu Demo");
        f.setSize(250, 250);
        f.setVisible(true);
    }
}
```

Explanation: In this program, the dialog box is called and displayed on clicking a menu item. The class dialog represents a dialog window. It is similar to classes seen earlier. In the constructor of this dialog class, one label and two buttons are added. When “OK” button is pressed the dialog box will be disposed.

In the MyFrame class constructor a menu is created. In the menu, just one menu item is added by the name “Show Dialog”. When this menu item is clicked, actionPerformed method will be called. In the method an instance of dialog class is created and displayed using setVisible method by passing argument true.

In the dialog box when OK button is pressed, actionPerformed method is called in the dialog class and code in the method disposes the dialog box.



Figure 16.37 Output screen of Program 16.35

16.8 THE FILEDIALOG CLASS

The FileDialog class displays a dialog window from which the user can select file. It is a subclass of Dialog class. The dialog window provided by an object of the FileDialog class is the standard file selection window which the operating system provides. It is modal dialog box. Since it is a modal dialog, when the application calls its show method to display the dialog, it blocks the rest of the application until the user has chosen a file.

The class defines the following frequently used constructors:

1. **public FileDialog(Frame parent)**

This form of constructor creates a file dialog for loading a file. The title of the file dialog is initially empty.

2. **public FileDialog(Frame parent, String title)**

This form of constructor creates a file dialog window with the specified title of loading a file. The files shown are those in the current directory.

3. **public FileDialog(Frame parent, String title, int mode)**

This form of constructor creates a file dialog window with the specified title for loading or saving file. If the value of mode is LOAD, then the file dialog is finding a file to read, and the files shown are those in the current directory. If the value of mode is SAVE, the file dialog is finding a place to write a file.

LOAD and SAVE are two properties of the FileDialog class.

```
/*PROG 16.36 DEMO OF FILEDIALOG VER 1 */

import java.awt.*;
import java.awt.event.*;
class FDdemo
{
    public static void main(String args[])
    {
        FileDialog fd;
        fd = new FileDialog(new Frame("Demo"), "File Dialog");
        fd.setVisible(true);
    }
}
```

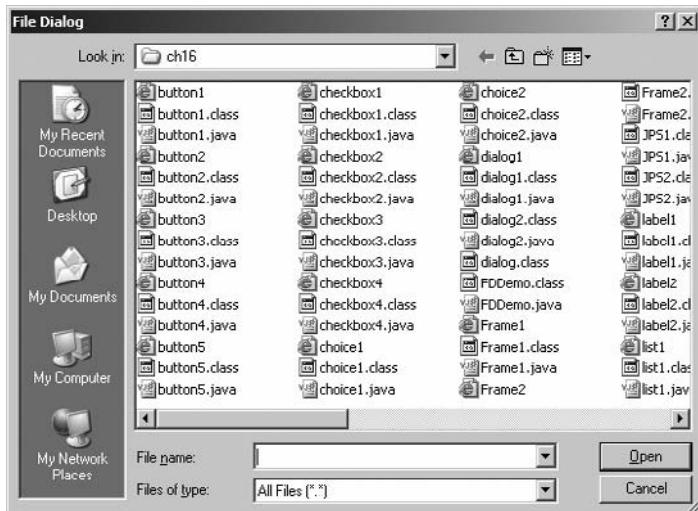


Figure 16.38 Output screen of Program 16.36

Explanation: The program is self-explanatory.

```
/*PROG 15.37 DEMO OF FILEDIALOG VER 2 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "FDDemo1" width = 200 height = 200>
</applet>
*/
class MyFrame extends Frame{
    MyFrame(String s)
    {
        super(s);
        setSize(200, 200);
    }
}
public class FDDemo1 extends Applet{
    FileDialog fd;
    Frame f;
    public void init(){
        setBackground(Color.orange);
        f = new MyFrame("Demo");
        fd = new FileDialog(f, "File Dialog");
        fd.setVisible(true);
    }
    public void paint(Graphics g) {
        g.drawString("File selected is " + fd.getFile(),
                    20, 50);
    }
}
```

```
        String d = "Directory of the File is " +  
                 fd.getDirectory();  
        g.drawString(d, 20, 70);  
    }  
}
```

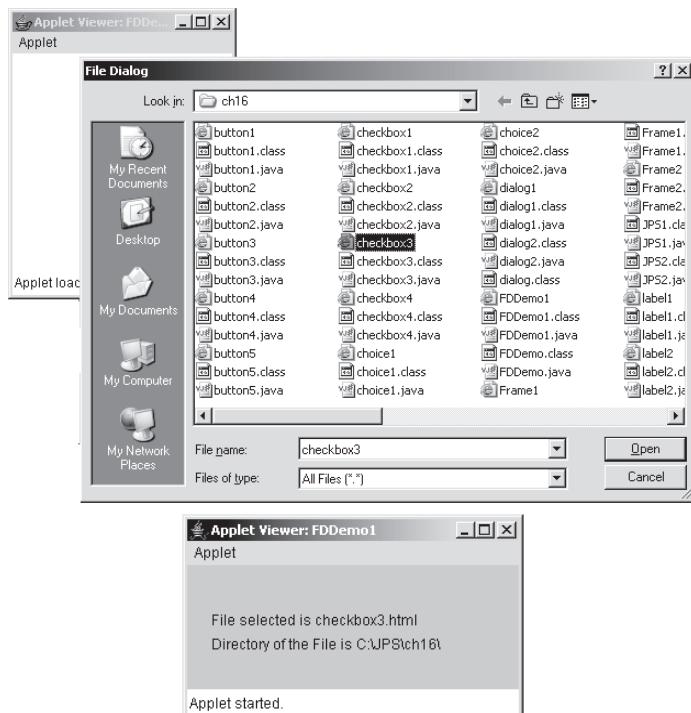


Figure 16.39 Output screen of Program 16.37

Explanation: This time an applet is created instead of an application for the creation of file dialog. The program also demonstrates how selected file name and directory name can be retrieved using `getFileName` and `getDirectoryName` method.

```
/*PROG 16.38 DEMO OF FILEDIALOG VER 3 */

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
/*
<applet code = "FDDemo2" width = 200 height = 200>
</applet>
*/
class MyFrame extends Frame
{
```

```
MyFrame(String s)
{
    super(s);
    setSize(200, 200);
}
}

public class FDDemo2 extends Applet{
    FileDialog fd;
    Frame f;
    byte b[];
    String s = null;
    public void init()
    {
        setBackground(Color.yellow);
        f = new MyFrame("Demo");
        fd = new FileDialog(f,"File Dialog");
        fd.setVisible(true);
        String f = fd.getFile();
        try {
            FileInputStream fis = new FileInputStream(f);
            b = new byte[64*64];
            fis.read(b);
            s = new String(b);
            fis.close();
        }
        catch(Exception E) {
            s = "Error in opening file";
        }
        TextArea ta = new TextArea(s, 15, 30);
        add(ta);
    }
}
```

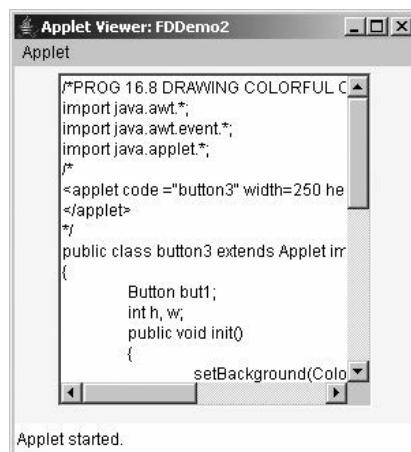


Figure 16.40 Output screen of Program 16.38

Explanation: In this program, the selected file from file dialog is retrieved and opened using `FileInputStream`. The contents of file are read using `read` method and stored in byte array `b`. the size of array is chosen **64*64** as an approximation of size of file. The byte array is converted into `String s` . the string is added as contents of the `TextArea` instance `tb` of 15 rows and 30 columns.

16.9 PONDERABLE POINTS

1. AWT stands for Abstract Window Toolkit. The toolkit is defined within the `java.awt` package. It contains all of the classes for creating user interfaces like Buttons, Labels, TextBox, Lists etc, and for painting graphics and images.
2. Component class is the topmost class in the window hierarchy.
3. A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.
4. A generic AWT container object is a component that can contain other AWT components. The Container class is the subclass of the Component class. Containers (Frames, Dialogs, Windows and Panels) can contain component and are themselves components, thus can be added to Containers.
5. A panel provides space in which an application can attach any other component, including other panels. A Panel is a container that does not exist independently. It is a plain rectangular area.
6. In Java, a program can open an independent window by creating an object of type Frame. A Frame has a title, displayed in the title bar at the top of the window. It can have a menu bar containing one or more pull-down menus.
7. A Frame can be used within an applet or in an application program.
8. The component class is the base class for the AWT component.
9. For handling various AWT components, the respective listener interfaces are given below:

| | |
|---------------|------------------------------|
| Button | ActionListener |
| Checkbox | ItemListener |
| CheckboxGroup | ItemListener |
| List | ItemListener, ActionListener |
| Choice | ItemListener |
| ScrollBar | Adjustment Listener |
| Menu | ActionListener |

10. A Label object is a component for placing text in a container. A label displays a single line of read-only text i.e. static text.
11. A check box is a graphical component that can be in either an “on” (true) or “off” (false) state. Clicking on a check box changes its state from “on” to “off”, or from “off” to “on”.
12. The CheckboxGroup class is used to group together a set of Checkbox buttons. Exactly one check box button in a CheckboxGroup can be in the “on” state at any given time.
13. The List component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple items.
14. The Choice class presents a pop-up menu of choice. The current choice is displayed as the title of the menu. The menu lists a number of items. Only the selected item is displayed.
15. The Scrollbar class embodies a scroll bar, a familiar user-interface object. A scroll bar provides a convenient means for allowing a user to select from a range of values. A scroll bar can be either horizontal or vertical.

16. A `TextField` object is a text component that allows for the editing of a single line of text.
17. A `TextArea` object is a multi-line region that displays text. It can be set to allow editing or to be read-only. A `TextArea` displays multiple lines and might include scroll bars that the user can use to scroll through the entire contents of the `TextArea`.
18. A menu is a list of choices. A menu bar displays a list of top-level menu choice.
19. Only `Frame` can have menus since they implement the `MenuContainer` interface. Therefore an applet must create a frame in order to use menus.
20. Each menu item is an instance of `MenuItem` class attached to the menu.
21. Shortcut keys to menu items can be added using the `MenuShortcut` class. The `MenuShortcut` class represents a keyboard accelerator for a `MenuItem`. Menu shortcuts are created using virtual keycodes.
22. A `PopupMenu` is a menu which can be dynamically popped up at a specified position within a component. It is implemented in java by class `PopupMenu`.
23. Like a frame, a dialog box is a separate window. Unlike a frame, however, a dialog box is not completely independent. Every dialog box is associated with a frame, which is called its parent.
24. Dialog boxes can be either modal or modeless. When a modal dialog is created its parent frame is blocked. That is, the user will not be able to interact with the parent or even bring the parent to the front of the screen until the dialog box is closed. Modeless dialog boxes do not block their parents in the same way, so they seem a lot more like independent windows.
25. In Java, a dialog box is an object belonging to the class `Dialog` or to one of its subclasses.
26. The `FileDialog` class displays a dialog window from which the user can select a file. It is a subclass of `Dialog` class.

REVIEW QUESTIONS

1. What do you mean by Abstract Window Toolkit?
2. What are the different types of layout managers in Java?
3. What are the different components of Abstract Window Toolkit?
4. Explain the procedure to design menu in Java?
5. Differentiate between text area and text box.
6. Differentiate between the Component and Container class.
7. Explain inset value and how you can set the inset values.
8. What is the difference between checkbox and checkbox group?
9. Write a Java program using checkboxes to display different presidents' names.
10. Write a Java program to display the different TV channels using checkboxgroup.
11. Write a Java program to display the different DLF-IPL 20/20 team names using checkboxes and checkboxgroup.
12. Write a Java program to display the different prices of the books using the choice object.
13. Write Java program to display the different car names using list object.
14. Write a Java program to display the different IIT's names in India using the choice object and list object.
15. Create a menu is a frame as shown below:

| | | | |
|--------------|-----------------|-------------|---------------|
| Books | Language | Film | Search |
|--------------|-----------------|-------------|---------------|

The Books menu has the following items: C, C++, Java, Oracle, VB, and ASP. The language menu consists of French, German, English, Tamil, and Hindi. The File menu has the following: Titanic, Jurassic Park, Tomorrow never Dies, Jeans, and Slumdog. The search engine consists of Yahoo, Hotmail, Sify, Google, and Lycos.

Multiple Choice Questions

1. _____ is the topmost class in the window hierarchy
 - (a) Component
 - (c) Container
 - (b) Event
 - (d) Window
2. Match the following:

| | |
|--------------|-----------------------|
| 1. Button | a. ItemListener |
| 2. Checkbox | b. ActionListener |
| 3. ScrollBar | c. AdjustmentListener |

 - (a) 1-a, 2-b, 3-c
 - (c) 1-c, 2-a, 3-b
 - (b) 1-b, 2-c, 3-a
 - (d) 1-b, 2-a, 3-c
3. A _____ object is a component for placing text in a container.
 - (a) Textbox
 - (c) Label
 - (b) Checkbox
 - (d) None of the above
4. The _____ class is used to group together a set of Checkbox buttons.
 - (a) TextArea
 - (b) CheckboxGroup
 - (c) CheckBoxGroup
 - (d) MenuShortcut
5. Which is the constructor of Frame class
 - (a) void Frame()
 - (b) public setFrame()
 - (c) public Frame(String title)
 - (d) void Frame(String title)
6. Which of the following methods is used to returns the text of the label as String object
 - (a) public String getText()
 - (b) public void getText()
 - (c) public Char getText(String text, String s)
 - (d) None of the above
7. Which is the method of Checkbox class
 - (a) public String getLabel()
 - (b) public String setLabel()
 - (c) public boolean getState()
 - (d) (a) and (c)
8. The public int getSelectedIndex() method returns the index of the selected item; if no item is selected, or if multiple items are selected, it returns
 - (a) 0
 - (c) -1
 - (b) +1
 - (d) -2
9. The method public String getSelectedItem() returns the selected item on the list, if no item is selected, or if multiple items are selected, it returns
 - (a) 0
 - (c) -1
 - (b) +1
 - (d) None of the above
10. Each menu item is an instance of
 - (a) MenuShortcut class
 - (b) PopupMenu class
 - (c) MenuItem
 - (d) None of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. d 3. c 4. b 5. c 6. a 7. d 8. c 9. a 10. c

17

Working with Layout

17.1 LAYOUT AND LAYOUT MANAGER

A layout manager automatically arranges the controls within a window by using some types of algorithm. In Java `LayoutManager` is an interface for classes that know how to lay out Containers. A layout manager is an instance of any class that implements the `LayoutManager` interface.

The sizes and positions of the components in a container are usually controlled by a layout manager. Different layout managers implement different ways of arranging components. There are several predefined layout manager classes in the AWT: `FlowLayout`, `GridLayout`, `BorderLayout`, `CardLayout` and `GridBagLayout`. It is also possible to define new layout managers, if none of these suit one's purpose. Every container is assigned a default layout manager when it is first created. For Panels, including applets, the default layout manager belongs to the class `FlowLayout`. For Windows, the default layout manager is a `BorderLayout`. The layout manager of a container can be changed using its `setLayout()` method. The method's signature is as shown below:

```
public void setLayout(LayoutManager mgr)
```

The method sets the layout manager `mgr` for this container.

At first instance, it may seem fairly difficult to get the exact layout that would be on the applets and windows, but after some practice and experimentation it will become easy for one to decide which layout manager will be most suitable for the applications.

All the `LayoutManager` classes defined by Java are discussed below. They are part of the `java.awt` package.

17.2 FLOWLAYOUT

`FlowLayout` is the default layout manager. A flow layout arranges components in a directional flow, much like lines of text in a paragraph. A `FlowLayout` simply lines up its components as they come in a form of row. After laying out as many items as will fit in a row across the container, it will move on to the next row. The components in a given row can be either left-aligned, right-aligned, or centred, and there can be horizontal and vertical gaps between components. The class defines the following constructors:

1. **public FlowLayout()**

This form of constructor creates a new `FlowLayout` with centred alignment and a default 5-unit horizontal and vertical gap.

2. **public FlowLayout(int align)**

This form of constructor creates a new `FlowLayout` with the specified alignment and a default 5-unit horizontal and vertical gap. The value of the alignment argument must be one of `FlowLayout.LEFT`, `FlowLayout.RIGHT`, `FlowLayout.CENTER`, `FlowLayout.LEADING` or `FlowLayout.TRAILING`.

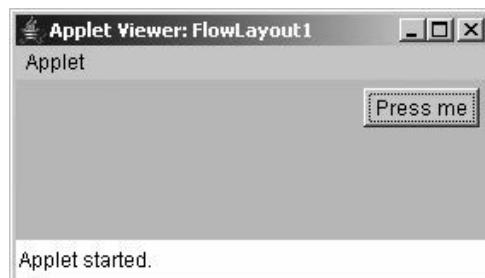
3. **public FlowLayout(int align, int hgap, int vgap)**

This form of constructor creates a new flow layout manager with the indicated alignment and, horizontal and vertical gaps.

For example, suppose an applet wants to contain one button, located in its upper right corner. The default layout manager for an applet is a FlowLayout that uses centre alignment. It would centre the button horizontally. The applet needs to be given a new layout manager that uses right alignment, which will show the button to the right edge of the applet. The following init() method will carry out this function.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class FlowLayout1 extends Applet
{
    public void init()
    {
        setBackground(Color.pink);
        setLayout(new FlowLayout(FlowLayout.RIGHT,5,5));
        add(new Button("Press me"));
    }
}
```

The above code will produce the following output:



```
/*PROG 17.1 DEMO OF FLOWLAYOUT */
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayout2" width =200 height = 200>
</applet>
*/
public class FlowLayout2 extends Applet
{
    Button but[];
```

```

public void init()
{
    setBackground(Color.magenta);
    setLayout(new FlowLayout(FlowLayout.LEFT));
    but = new Button[10];
    for(int i=0;i<but.length;i++)
    {
        but[i] = new Button("Button"+(i+1));
        add(but[i]);
    }
}

```

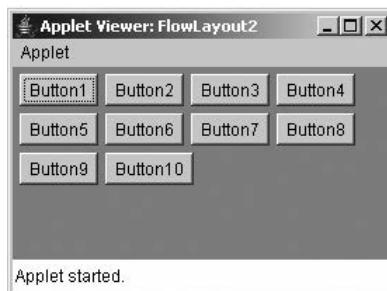


Figure 17.1 Output screen of Program 17.1

Explanation: This program is simple. The default alignment for the FlowLayout is CENTER. The LEFT alignment is set and an array of 10 buttons are created. All buttons instances are added to the applet window using the `for` loop.

17.3 BORDERLAYOUT

A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west and centre. Each region may contain no more than one component, and is identified by a corresponding constant: NORTH, SOUTH, EAST, WEST and CENTER. A BorderLayout places one component in the centre of a container. The central component is surrounded by up to four other components that border it to the "North", "South", "East" and "West", as shown in the diagram. Each of the four bordering components is optional. The layout manager first allocates space to the bordering components. Any space that is left over goes to the centre component. **BorderLayout** is the default layout manager for all windows, such as **Frames** and **Dialogs**.

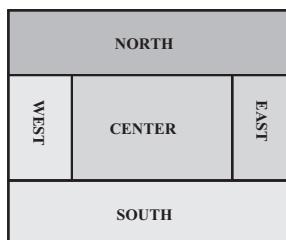


Figure 17.2 BorderLayout regions

The class defines the following constructors:

1. **public BorderLayout()**

This form of constructor creates a new border layout with no gaps between components.

2. **public BorderLayout(int hgap, int vgap)**

This form of constructor creates a border layout with the specified gap between components. The horizontal gap is specified by hgap and the vertical gap by vgap.

When adding component with BorderLayout, the other form of add method has to be used which takes two parameters: one is the component to be added and second is one of the constants defined by the BorderLayout class. This is as shown below:

```
void add(Component comp, Object region);
```

The method is defined by the Container class.

```
/*PROG 17.2 DEMO OF BORDERLAYOUT */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="BorderLayout1" width =200 height = 200>
</applet>
*/
public class BorderLayout1 extends Applet
{
    public void init()
    {
        setLayout(new BorderLayout());
        add(new Button("I'm at Top"),BorderLayout.NORTH);
        add(new Button("I'm at Bottom"),BorderLayout.SOUTH);
        add(new Button("I'm at Right"),BorderLayout.EAST);
        add(new Button("I'm at Left"), BorderLayout.WEST);
        add(new Button("I'm at Center"),BorderLayout.CENTER);
    }
}
```

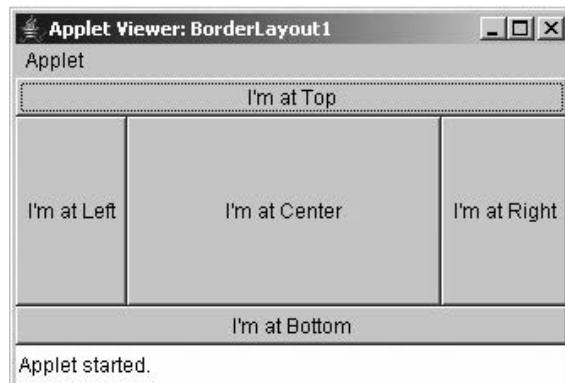


Figure 17.3 Output screen of Program 17.2

Explanation: This program demonstrates **BorderLayout** layout manager. Five buttons are added in the five regions defined by BorderLayout. Note the second parameter in the add method as Shown in the program 17.2. This determines where the first parameter (i.e., component) has to be placed.

17.4 GRIDLAYOUT

A GridLayout lays out components in a grid of equal-size rectangles. The illustration shows how the components would be arranged in a grid layout with three rows and two columns as given in the following program. If a container uses a GridLayout, the appropriate add method takes a single parameter of type Component (e.g., add(myButton)). Components are added to the grid in the order shown; that is each row is filled from left to right before going onto the next row.

The class defines the following constructors:

1. **public GridLayout()**

This form of constructor creates a grid layout with a default of one column per component in a single row.

2. **public GridLayout(int rows, int cols)**

This form of constructor creates a grid layout with the specified number of rows and columns. All components in the layout are given equals size. One, but not both, of rows and columns can be zero, which means that any number of objects can be placed in a row or a column.

3. **public GridLayout(int rows, int cols, int hgap, int vgap)**

This form of constructor creates a grid layout with the specified number of rows and columns. All components in the layout are given equal size. In addition, the horizontal and vertical gaps are set to the specified values. Horizontal gaps are placed between each of the columns, vertical gaps are placed between each of the rows.

In the second and third form of constructor specifying rows as zero allows for unlimited-length columns and specifying columns as zero allows for unlimited-length rows.

```
/*PROG 17.3 DEMO OF GRIDLAYOUT VER 1*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridLayout1" width =200 height = 200>
</applet>
*/
public class GridLayout1 extends Applet
{
    public void init()
    {
        setLayout(new GridLayout(3, 2));
        add(new Button("First"));
        add(new Button("Second"));
        add(new Button("Third"));
        add(new Button("Four"));
        add(new Button("Five"));
        add(new Button("Six"));
    }
}
```

Explanation: In this program, the **GridLayout** of three rows and two columns are set as layout for the applet window. Six Button instances have been added. It is clear from the output that buttons are placed in a grid of 3 by 2. Note in the program if layout is set as `setLayout(new GridLayout(0, 2))`; output remains the same. Why is it so?

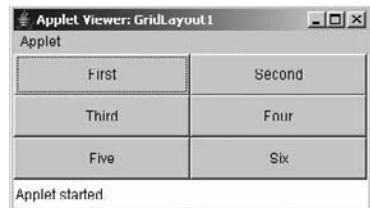


Figure 17.4 Output screen of Program 17.3

```
/*PROG 17.4 DEMO OF GRIDLAYOUT VER 2*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridLayout2" width =200 height = 200>
</applet>
*/
public class GridLayout2 extends Applet
{
    public void init()
    {
        setLayout(new GridLayout(1, 2, 10, 0));
        Panel P1 = new Panel();
        P1.setBackground(Color.orange);
        P1.setLayout(new GridLayout(2, 3, 5, 5));
        for (int i = 1; i <= 6; i++)
            P1.add(new Button(" " + i));
        Panel P2 = new Panel();
        P2.setBackground(Color.pink);
        P2.setLayout(new GridLayout(3, 2, 5, 5));
        for (int i = 1; i <= 6; i++)
            P2.add(new Button(" " + i));
        add(P1);
        add(P2);
        setBackground(Color.yellow);
    }
}
```

Explanation: In this program, two Panel instance P1 and P2 are created. For the applet window the two components are two Panel instance P1 and P2. These two are added onto the applet window in a grid of one row and two columns with horizontal gap and vertical gap of 10 and 0 pixels, respectively. For the first Panel instance **P1**, the GridLayout set is of two rows, three columns with horizontal gap and vertical gap of 5 pixels each.

Initially Button instances are added to **P1** using the `for` loop, then again for the **P2**. Both the Panel

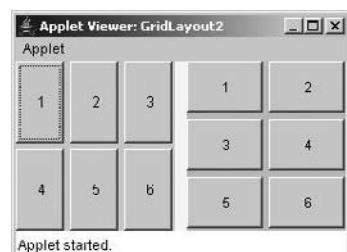


Figure 17.5 Output screen of Program 17.4

instances `P1` and `P2` are then added to applet window. The background colour of two panels and applet window is chosen different just for differentiation.

17.5 GRIDBAGLAYOUT

A `GridBagLayout` is similar to a `GridLayout` in that the container is broken down into rows and columns of rectangles. However, a `GridBagLayout` is much more sophisticated because the rows do not all have to be of the same height, the columns do not all have to be of the same width and a component in the container can spread over several rows and columns. There is a separate class, `GridBagConstraints`, that is used to specify the position of a component, the number of rows and columns that it occupies and several additional properties of the component.

Using a `GridBagLayout` is rather complicated, so it will not be discussed here much.

17.6 CARDLAYOUT

`CardLayouts` differ from other layout managers in that in a container that uses a `CardLayout`, only one of its components is visible at any given time. Think of the components as a set of “cards”. Only one card is visible at a time, but one can flip from one card to another. Methods are provided in the `CardLayout` class for flipping to the first card, to the last card and to the next card in the deck. A name can be specified for each card as it is added to the container, and there is a method in the `CardLayout` class for flipping directly to the card with a specified name.

The class defines the following constructors:

1. **public CardLayout()**

This form of constructor creates a new card layout with gaps of size zero.

2. **public CardLayout(int hgap, int vgap)**

This form of constructor creates a new card layout with the specified horizontal and vertical gaps. The horizontal gaps are placed at the left and right edges; the vertical gaps are placed at the top and bottom edges.

When cards are added to the panel, they are usually given a name. For that, a new form of add method can be used. Its signature is given below:

```
void add(Component Pobj, Object name);
```

Here, `name` is a string that specifies the name of the card whose panel is specified by `Pobj`.

The class defines the methods for flipping among the cards and showing a specific card.

1. **public void first(Container parent)**

This method flips to the first card of the container.

2. **public void next(Container parent)**

This method flips to the next card of the specified container. If the currently visible card is the last one, this method flips to the first card in the layout.

3. **public void previous(Container parent)**

This method flips to the previous card of the specified container. If the currently visible card is the first one, this method flips to the last card in the layout.

4. **public void last(Container parent)**

This method flips to the last card of the container.

5. **public void show(Container parent, String name)**

This method flips to the component that was added to this layout with the specified name. If no such component exists, then nothing happens.

When CardLayout is used, the use of Panels is a must. The steps which must be followed in order to use CardLayout are as follow:

1. Create a new Panel instance and an instance of CardLayout.
2. Set the layout for Panel instance as CardLayout using its instance created in first step.
3. Now for each card, create a separate Panel instance and add whatever components are needed to add to that panel like buttons, checkboxes, labels, list, choice, etc.
4. Each Panel instance created in Step 3 is added to the main Panel instance created in Step 1, giving a unique card name for each card.
5. Finally, the main Panel is added to the applet window.

All these steps are illustrated in the programs given below:

```
/*PROG 17.5 DEMO OF CARDLAYOUT VER 1*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "CardLayout1" width = 200 height = 200>
</applet>
*/
public class CardLayout1 extends Applet implements
ActionListener
{
    Panel P1, P2, P3, Main;
    CardLayout CLO;
    Button B[];
    public void init()
    {
        B = new Button[4];
        B[0] = new Button("First");
        B[1] = new Button("Next");
        B[2] = new Button("Last");
        B[3] = new Button("Previous");
        for (int i = 0; i < B.length; i++)
        {
            add(B[i]);
            B[i].addActionListener(this);
        }
        //step1
        CLO = new CardLayout();
        Main = new Panel();
        //Step2
        Main.setLayout(CLO);
```

```

//Step3 start here
P1 = new Panel();
P1.add(new Label("Card 1 Label"));
P1.add(new Button("Card 1 Button"));
P1.setBackground(Color.red);

P2 = new Panel();
P2.add(new Label("Card 2 Label"));
P2.add(new Button("Card 2 Button"));
P2.setBackground(Color.blue);
P3 = new Panel();
P3.add(new Label("Card 3 Label"));
P3.add(new Button("Card 3 Button"));
P3.setBackground(Color.magenta);

//Step3 ends here
//Step4 starts here
Main.add(P1, "C1");
Main.add(P2, "C2");
Main.add(P3, "C3");

//Step4 ends here

//Step 5

add(Main);
}

public void actionPerformed(ActionEvent ae)
{
    if (ae.getSource() == B[0])
        CLO.first(Main);
    if (ae.getSource() == B[1])
        CLO.next(Main);
    if (ae.getSource() == B[2])
        CLO.last(Main);
    if (ae.getSource() == B[3])
        CLO.previous(Main);
}
}
}

```

Explanation: In this program, in Step1 an instance of CardLayout CLO is created and added to an instance of Panel (i.e., Main) setting the layout for Main panel as CardLayout. Next, three panels P1, P2 and P3 are created. Each panel contains one label and one button. The background for each color is set to a different color. Next, the three panels are added to the Main panel using add method where card name for each panel is set to “C1”, “C2” and “C3”. In the final step, the Main panel is added to the applet window.

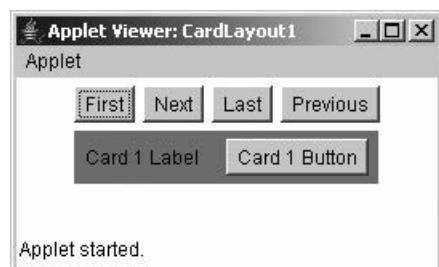


Figure 17.6 Output screen of Program 17.5

An array of **Button** is created and added to the main applet window. Listeners are added for the buttons to receive buttons press notifications. When any of the buttons is pressed `actionPerformed` method is called. In the method, it is first checked which button is pressed, depending on that, the first, next, last or previous method is called. Each method takes its parent as argument that is `Main` panel. By default, the first card in a deck of cards is shown.

```
/*PROG 17.6 DEMO OF CARDLAYOUT VER 2*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code ="CardLayout2" width = 200 height=200>
</applet>
*/
public class CardLayout2 extends Applet implements
ActionListener
{
    Label[] L;
    Panel[] P;
    Panel Main;
    CardLayout CLO;
    Button B[];
    int i;
    Color[] col;
    public void init()
    {
        CLO = new CardLayout();
        Main = new Panel();
        Main.setLayout(CLO);
        P = new Panel[3];
        B = new Button[3];
        L = new Label[3];
        col = new Color[] { Color.red, Color.green,
                           Color.blue };
        for (i = 0; i < B.length; i++)
        {
            B[i] = new Button("Button " + (i + 1));
            add(B[i]);
            B[i].addActionListener(this);
        }
        for (i = 0; i < L.length; i++)
        {
            L[i] = new Label("Hello from Card " +(i+1));
            P[i] = new Panel();
            P[i].add(L[i]);
            Main.add(P[i], "Card " + (i + 1));
            P[i].setBackground(col[i]);
        }
        add(Main);
    }
    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == B[0])
            CLO.previous(Main);
        else if (e.getSource() == B[1])
            CLO.last(Main);
        else if (e.getSource() == B[2])
            CLO.first(Main);
        else
            CLO.next(Main);
    }
}
```

```

        }
    public void actionPerformed(ActionEvent ae)
    {
        if (ae.getSource() == B[0])
        {
            CLO.show(Main, "Card 1");
        }
        else if (ae.getSource() == B[1])
        {
            CLO.show(Main, "Card 2");
        }
        else
        {
            CLO.show(Main, "Card 3");
        }
    }
}

```

Explanation: This program is similar to the previous one but here an array of different types are used to make the processing simpler. Just one label is added to each panel. In the actionPerformed method, show method of CardLayout class is used for showing any of the cards. The second argument to this method specifies which card is to be seen.

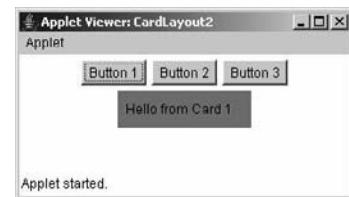


Figure 17.7 Output screen of Program 17.6

```

/*PROG 17.7 DEMO OF CARDLAYOUT VER 3*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code ="CardLayout3" width = 200 height=200>
</applet>
*/
public class CardLayout3 extends Applet implements
ActionListener, ItemListener
{
    Checkbox ch1, ch2, ch3, ch4, ch5, ch6, ch7;
    Panel P1, P2, Main;
    CardLayout CLO;
    Button B1, B2;
    String msg = " ";
    public void init() {
        setBackground(Color.orange);
        B1 = new Button("Fruits");
        B2 = new Button("Vegetables");
        add(B1);

```

```
add(B2);

CLO = new CardLayout();
Main = new Panel();
Main.setLayout(CLO);

ch1 = new Checkbox("Pomegranate");
ch2 = new Checkbox("Mango");
ch3 = new Checkbox("Guava");
ch4 = new Checkbox("Banana");
ch5 = new Checkbox("Spinach");
ch6 = new Checkbox("Beans");
ch7 = new Checkbox("Potato");

P1 = new Panel();
P1.add(ch1);
P1.add(ch2);
P1.add(ch3);
P1.add(ch4);
P1.setBackground(Color.cyan);
P2 = new Panel();
P2.add(ch5);
P2.add(ch6);
P2.add(ch7);
P2.setBackground(Color.yellow);
//add panels to main card panel

Main.add(P1, "Fruits");
Main.add(P2, "Vegetables");

//add cards to main applet panel

add(Main);

//register to receive action events

B1.addActionListener(this);
B2.addActionListener(this);
ch1.addItemListener(this);
ch2.addItemListener(this);
ch3.addItemListener(this);
ch4.addItemListener(this);
ch5.addItemListener(this);
ch6.addItemListener(this);
ch7.addItemListener(this);
}

public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == B1)
    {
        CLO.show(Main, "Fruits");
    }
    else
```

```

        {
            CLO.show(Main, "Vegetables");
        }
    }
    public void itemStateChanged(ItemEvent ie){
        repaint();
    }
    public void paint(Graphics g) {
        msg = "You Selected:";

        if (ch1.getState())
            msg += ch1.getLabel() + " , ";
        if (ch2.getState())
            msg += ch2.getLabel() + " , ";
        if (ch3.getState())
            msg += ch3.getLabel() + " , ";
        if (ch4.getState())
            msg += ch4.getLabel() + " , ";
        if (ch5.getState())
            msg += ch5.getLabel() + " , ";
        if (ch6.getState())
            msg += ch6.getLabel() + " , ";
        if (ch7.getState())
            msg += ch7.getLabel() + " , ";
        g.drawString(msg, 10, 80);
    }
}
}

```

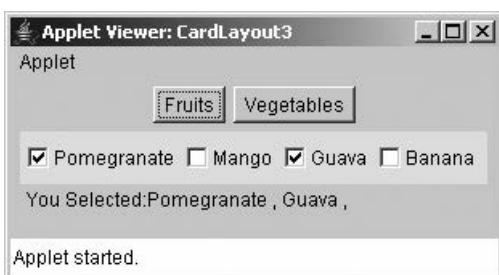


Figure 17.8 Output screen of Program 17.7 (Part 1)

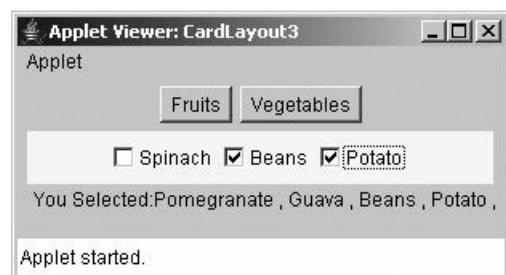


Figure 17.9 Output screen of Program 17.7 (Part 2)

Explanation: Onto the main applet window, two buttons labelled “Fruits” and “Vegetables” are created. Each of these buttons when clicked, invoke one of the two created panels. The panel instance **P1** contains four checkboxes with fruit names as their labels. The Panel instance **P2** contains three checkboxes with vegetable names as their labels.

For each of the checkbox **ch1** to **ch7**, item listeners are added. When any of the panel is selected, it is shown to the user as `actionPerformed` method is called. In any of the **Panel P1** or **P2**, user can check the checkboxes. As user selected/deselected the checkboxes, the state of the checkbox changes and `itemStateChanged` method is invoked. In the method, depending on which checkbox was checked, its label is appended to the `String` object `msg`. In the end, using `drawString` method `msg` is displayed.

17.7 INSETS

An Insets object is a representation of the borders of a container. It specifies the space that a container must leave at each of its edges. The space can be a border, a blank space or a title. These values are used by the layout manager to inset the components when it lays out the window. The class defines the following constructor:

```
public Insets(int top, int left, int bottom, int right)
```

The constructor creates and initializes a new Insets object with the specified top, left, bottom and right insets.

In order to use insets in the program, one has to override `getInsets` method of Container class. Inside a new Insets, object can be created by using the above constructor.

The signature of the method is:

```
public Insets getInsets()
```

The method determines the insets of this container, which indicates the size of the container's border. A Frame object, for example, has a top inset that corresponds to the height of the frame's title bar.

```
/*PROG 17.8 DEMO OF INSETS */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code ="Insets1" width = 200 height =200>
</applet>
*/
public class Insets1 extends Applet
{
    public void init()
    {
        //set background colors so insets can be easily seen
        setBackground(Color.magenta);
        setLayout(new BorderLayout());
        add(new Button("I'm at Top"), BorderLayout.NORTH);
        add(new Button("I'm at Bottom"), BorderLayout.SOUTH);
        add(new Button("I'm at Right"), BorderLayout.EAST);
        add(new Button("I'm at Left"), BorderLayout.WEST);
        add(new Button("I'm at Center"), BorderLayout.CENTER);
    }
    //add insets
    public Insets getInsets()
    {
        return new Insets(10, 20, 10, 20);
    }
}
```

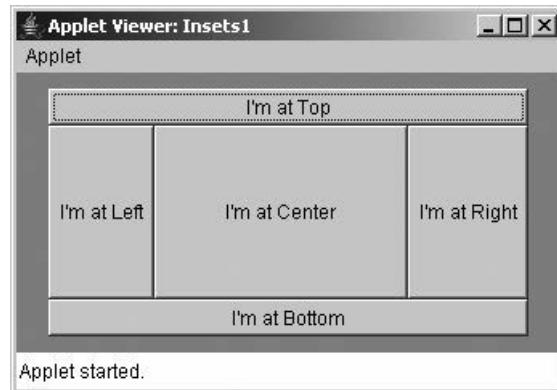


Figure 17.10 Output screen of Program 17.8

Explanation: This program is same as was seen in **BorderLayout** section. Here, the `getInsets` method is added that returns a new `Insets` object. The method insets the components on the applet window so that they are 10 pixels away from top and bottom and 20 pixels away from left and right. The background has been set to **magenta** so that insets are clearly visible.

17.8 PONDERABLE POINTS

1. A layout manager automatically arranges the controls within a window by using some type of algorithm. A layout manager is an instance of any class that implements the `LayoutManager` interface.
2. `FlowLayout` is the default layout manager. A flow layout arranges components in a directional flow, much like lines of text in a paragraph.
3. A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west and center. Each region may contain no more than one component, and is identified by a corresponding constant: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`.
4. `GridLayout` lays out components in a grid of equal size rectangles.
5. `CardLayout` differs from other layout managers in that it has a container that uses a `CardLayout`, only one of its components is visible at any given time.
6. An `Insets` object is a representation of the border of a container. It specifies the space that a container must leave at each of its edges.

REVIEW QUESTIONS

1. What is a Layout Manager and what are the different Layout Managers available in `java.awt` and what is the default Layout manager for the panel and the panel subclass?
2. How can we create a borderless window?
3. Can we add the same component to more than one container?
4. How are the elements of different layouts organized?
5. Which containers use a `BorderLayout` as their default layout?
6. Which containers use a `FlowLayout` as their default layout?
7. Which method is used to set the layout of a container?
8. Which method returns the preferred size of a component?

9. Which layout should be used to organize the components of a container in a tabular form?
10. What is the default layout for an applet, a frame and a panel?
11. An Applet has its Layout Manager set to the default of FlowLayout. What code would be the correct to change to another Layout manager? Justify your answer.

```
setLayoutManager (new GridLayout());
setLayout(new GridLayout(2, 2));
set GridLayout(2, 2);
setBorderLayout();
```

Multiple Choice Questions

1. _____ returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
 - (a) Object[] toArray(Objectarray[])
 - (b) Object[] toArray()
 - (c) void toArray[]
 - (d) None of the above
2. The signature of getInsets() method is:
 - (a) public Insets getInsets()
 - (b) public Insets getInsets(int top, int left, int bottom)
 - (c) public Insets getInsets(int top, int left, int bottom, int right)
 - (d) None of the above
3. Which is the constructor of CardLayout?
 - (a) public CardLayout(int top, int bottom, int left, int bottom)
 - (b) public CardLayout(int top, int left, int bottom, int right)
 - (c) public CardLayout(int hgap, int vgap)
 - (d) public CardLayout(int hgap, int left, int vgap, int right)
4. public GridLayout() creates a grid layout with a default of
5. _____ returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
 - (a) One column per component
 - (b) Two column per component
 - (c) Three columns per component
 - (d) None of the above
6. A border layout lays out a container and resizes its components to fit in
 - (a) 3 regions
 - (b) 4 regions
 - (c) 5 regions
 - (d) 6 regions
7. The basic signature of setLayout manager is:
 - (a) void setLayout()
 - (b) public void setLayout()
 - (c) public void setLayout(int left, int right)
 - (d) public void setLayout(Layout Manager mgr)
8. All the LayoutManager classes are part of
 - (a) java.lang
 - (b) java.util
 - (c) java.awt
 - (d) java.Applet
9. When you are using CardLayout, you must use:
 - (a) GridLayout
 - (b) Panels
 - (c) Frame
 - (d) Applet
10. Insets object is a representation of the _____ of a container.
 - (a) border
 - (b) visibility
 - (c) (a) and (b)
 - (d) none of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

-
- | | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|-------|
| 1. b | 2. a | 3. c | 4. a | 5. c | 6. d | 7. c | 8. a | 9. b | 10. a |
|------|------|------|------|------|------|------|------|------|-------|

18

The Collection Framework

18.1 INTRODUCTION

The Collection Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection. The framework provides a convenient API to many of the abstract data types used in data structure curriculum: maps, sets, lists, trees, arrays, hashtables and other collections. Because of their object-oriented design, the Java classes in the collections framework encapsulate both the data structures and the algorithms associated with these abstractions. The framework provides a standard programming interface to several most common abstractions, without burdening the programmer with too many procedures and interfaces. The operations supported by the collections framework nevertheless permit the programmer to easily define higher level of abstractions, such as stacks, queues and thread-safe collections. Some of the features of collections are presented below:

1. The implementation of the fundamental collections like dynamic arrays, linked lists, trees and hash tables is highly efficient which results in high performance.
2. All the collections provide almost same look and feel and their way of working is similar to each other.
3. Extending a collection is very easy, and so is adapting.
4. Whole of the collections are designed around a set of standard interfaces (discussed shortly).
5. Collection Framework also allows creating one's own collection, if required.

18.2 COLLECTION FRAMEWORK

The collections framework is made up of a set of interfaces for working with groups of objects. The different interfaces describes different types of groups. For the most part, once the differences are understood, understanding the framework is easy. While one always needs to create specific implementations of the interfaces, access to the actual collection should be restricted to the use of the interface methods, thus allowing a programmer to change the underlying data structure, without altering the rest of the code. Figure 18.1 shows the framework hierarchy.

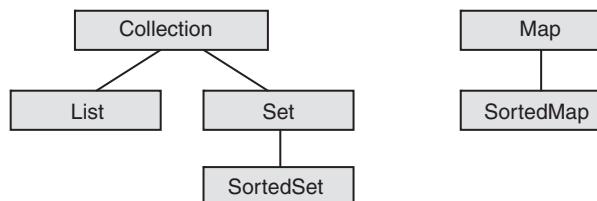


Figure 18.1 Collection interface hierarchy

In mathematics, a map is just a collection of pairs. In the Collection Framework, however, the interfaces `Map` and `Collection` are distinct. The reasons for this distinction have to do with the ways that `Set` and `Map` are used in the Java libraries. The typical application of a `Map` is to provide access to values stored by keys. The sets of collection operations are all there, but one works with a key-value pair instead of an isolated element. `Map` is therefore designed to support the basic operations of `get()` and `put()`, which are not required by `Set`.

The following points need to be remembered regarding Collection Framework:

- (i) The `Collection` interface is a group of objects, with duplicates allowed.
- (ii) The `Set` interface extends `Collection` but forbids duplicates.
- (iii) The `List` interface extends `Collection`, allows duplicates, and introduces positional indexing.
- (iv) The `Map` interface extends neither `Set` nor `Collection`.

The other interfaces of the Collection Framework are described later in the chapter. In this section, the aforesaid interfaces are discussed in detail.

18.2.1 Collection Interface

The `Collection` interface is used to represent any group of objects, or elements. The interface is used when a group of elements is to be worked in as general a manner as possible. This interface is implemented by all collection classes. Frequently used method of this interface is given in the Table 18.1. As the `Collection` method is at the root of the hierarchy, understanding all its methods is a must.

The interface supports basic operations like adding and removing. Object is added to collection using ‘add’ and removed using ‘remove’. When an element is needed to be removed, only a single instance of the element in the collection is removed, if present. Both of the method takes object as argument; it means any type of argument can be added to the collection but it must be an object. Primitive types must first be converted to objects before they can be added to collection.

The `Collection` interface also supports query operations: `size()`, `isEmpty()`, `contains()` etc.

The `iterator()` method of the `Collection` interface returns an `Iterator`. With the `Iterator` interface methods, a collection can be traversed from start to finish and elements safely removed from the underlying `Collection`. It is discussed in detail later in the chapter.

The `containsAll()` method allows to discover if the current collection contains all the elements of another collection, a subset. The remaining methods are optional in that a specific collection might not support the altering of the collection. The `addAll()` method ensures all elements from another collection are added to the current collection, usually a union. The `clear()` method removes all elements from the current collection. The `removeAll()` method is like `clear()` but only removes a subset of elements. The `retainAll()` method is similar to the `removeAll()` method but does what might be perceived as the opposite: It removes from the current collection those elements not in the other collection, an intersection.

| Method Signature | Description |
|---|---|
| <code>boolean add(Object obj)</code> | Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates. |
| <code>boolean addAll(Collection c)</code> | Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false. |
| <code>void clear()</code> | Removes all elements from the invoking collection. |

(Continued)

| | |
|--|--|
| <code>boolean contains(Object obj)</code> | Returns true if obj is an element of the invoking collection. Otherwise, returns false. |
| <code>boolean containsAll(Collection c)</code> | Returns true if the invoking collection contains all elements of c. otherwise, returns false. |
| <code>boolean equals(Object obj)</code> | Returns true if the invoking collection and obj are equal. Otherwise, returns false. |
| <code>boolean isEmpty()</code> | Returns true if the invoking collection is empty. Otherwise, returns false. |
| <code>Iterator iterator()</code> | Returns an iterator for the invoking collection. |
| <code>Boolean remove(Object obj)</code> | Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false. |
| <code>Boolean removeAll(Collection c)</code> | Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. |
| <code>Boolean retainAll(Collection c)</code> | Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. |
| <code>int size()</code> | Returns the number of elements held in the invoking collection. |
| <code>Object[] toArray()</code> | Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements. |
| <code>Object[] toArray(Objectarray[])</code> | Returns an array containing only those collection elements whose type matches that of the array. |

Table 18.1 Method of Collection interface

18.2.2 The List Interface

The List interface extends the Collection interface to define an ordered collection, permitting duplicates. The interface adds position-oriented operations, as well as the ability to work with just a part of the list. The first element in the list starts at index **0**. Elements can be added and accessed by their position in the list. The method of List interface are shown in the Table 18.2.

| Method Signature | Description |
|--|--|
| <code>Void add(index, Object obj)</code> | Inserts into the invoking list at the index passed in index. Any expressing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| <code>Boolean addAll(int index, Collection c)</code> | Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise. |
| <code>Object get(int index)</code> | Returns the object stored at the specified index within the invoking collection. |

| | |
|---|--|
| <code>int indexOf(Object obj)</code> | Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned. |
| <code>int lastIndexOf(Object obj)</code> | Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned. |
| <code>ListIterator listIterator()</code> | Returns an iterator to the start of the invoking list. |
| <code>ListIterator listIterator(int index)</code> | Returns an iterator to the invoking list that begins at the specified index. |
| <code>Object remove(int index)</code> | Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one. |
| <code>Object set(int index, Object obj)</code> | Assigns obj to the location specified by index within the invoking list. |
| <code>List subList(int start, int end)</code> | Returns a list that includes elements from start to end; -1 in the invoking list. Elements in the returned list are also referenced by the invoking object. |

Table 18.2 Methods of List interface

The position-oriented operations include the ability to insert an element or Collection, get an element, as well as remove or change an element. Searching for an element in a List can be started from the beginning or end and will report the position of the element, if found.

```
void add(int index, Object element)
boolean addAll(int index, Collection collection)
Object get(int index)
int indexOf(Object element)
int lastIndexOf(Object element)
Object remove(int index)
Object set(int index, Object element)
```

The List interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position-friendly manner:

```
ListIterator listIterator()
ListIterator listIterator(int startIndex)
List subList(int fromIndex, int toIndex)
```

In working with `subList()`, it is important to mention that the element at `fromIndex` is in the sublist, but the element at `toIndex` is not. This loosely maps to the following loop test cases:

```
for(int i = fromIndex; i<toIndex; i++)
{
    process element at position i
}
```

In addition, it should be mentioned that changes to the sublist (like `add()`, `remove()` and `set()` calls) have an effect on the underlying List.

The `ListIterator` interface extends the `Iterator` interface to support bidirectional access, as well as adding or changing elements in the underlying collection. It will be discussed later in detail.

18.2.3 The Set Interface

The Set interface extends the Collection interface and, by definition, forbids duplicates within the collection. All the original methods are present and no new method are introduced. The concrete Set implementation classes rely on the `equals()` method of the object added to check for equality.

18.2.4 The SortedSet Interface

The Collection Framework provides a special `Set` interface for maintaining elements in a sorted order: `SortedSet`. The methods of this interface are shown in the Table 18.3.

| Method Signature | Description |
|---|---|
| <code>Comparator comparator()</code> | Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned. |
| <code>Object first()</code> | Returns the first element in the invoking sorted set. |
| <code>SortedSet headSet (Object end)</code> | Returns a <code>SortedSet</code> containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set. |
| <code>Object last()</code> | Returns the last element in the invoking sorted set. |
| <code>SortedSet subset (Object start Object end)</code> | Returns a <code>SortedSet</code> that includes those elements between start and end-1. Elements in the returned collection are also referenced by the invoking object. |
| <code>SortedSet tailSet (Object start)</code> | Returns a <code>SortedSet</code> that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object. |

Table 18.3 Method of `SortedSet` interface

The interface provides access methods to the ends of the set as well as to subset of the set. When working with subset of the list, changes to the subset are reflected in the source set. In addition, changes to the source set are reflected in the subset. This works because subsets are identified by elements at the end points, not indices. In addition, if the `fromElement` is part of the source set, it is part of the subset. However, if the `toElement` is part of the source set, it is not part of the subset. If a particular `toElement` is to be in the subset, the next element must be found. In the case of a `String`, the next element is the same string with a null character appended element (`string"\0"`).

18.3 THE COLLECTION CLASS

In this section, all the collection classes are discussed that implement all or some of the interfaces discussed earlier. Each class will be described in detail. They are given in the Table 18.4.

| Method Signature | Description |
|-------------------------------------|---|
| <code>AbstractCollection</code> | Implements most of the <code>Collection</code> interface. |
| <code>AbstractList</code> | Extends <code>AbstractCollection</code> and implements most of the <code>List</code> interface. |
| <code>AbstractSequentialList</code> | Extends <code>AbstractList</code> for use by a collection that uses sequential, rather than random, access of its elements. |
| <code>LinkedList</code> | Implements a linked list by extending <code>AbstractSequentialList</code> . |
| <code>ArrayList</code> | Implements a dynamic array by extending <code>AbstractList</code> . |
| <code>AbstractSet</code> | Extends <code>AbstractCollection</code> and implements most of the <code>Set</code> interface. |
| <code>HashSet</code> | Extends <code>AbstractSet</code> for use with a hash table. |
| <code>TreeSet</code> | Implements a set stored in a tree. Extends <code>AbstractSet</code> . |

Table 18.4 Collection classes

18.4 ARRAYLIST AND LINKEDLIST CLASSES

There are two general-purpose List implementations in the Collection Framework: ArrayList and LinkedList. Which of two List implementations should be used depends on the specific needs. If random access needs to be supported, without inserting or removing elements from any place other than the end ArrayList offers the optimal collection. If, however, one needs to frequently add and remove elements from the middle of the list and only access the list elements sequentially LinkedList offers the better implementation.

Each of the classes is discussed in turn:

ArrayList Class

The declaration of ArrayList class is given as:

```
public class ArrayList extends AbstractList
    implements List, RandomAccess, Cloneable, Serializable
```

It is visible that ArrayList class extends the class AbstractList and implements the four interfaces:

```
List, RandomAccess, Cloneable and Serializable.
```

The class is resizable-array implementation of the List interface. It implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provide methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector (discussed later), except that it is unsynchronized.)

The class ArrayList is like dynamic array of objects. The elements can be added or removed dynamically. When elements are added, array list grows and when elements are removed it shrinks. The initial size of array in list is some fixed size. As this size exceeds, size is automatically increased. Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically.

The ArrayList has the following constructors:

- 1. public ArrayList()**

This form of constructor creates an empty list with an initial capacity of ten.

- 2. public ArrayList(Collection c)**

This form of constructor creates a list containing the elements of the specified collection c. The ArrayList instance has an initial capacity of 110 per cent the size of the specified collection.

- 3. public ArrayList(int initialCapacity)**

This form of constructor creates an empty list with the specified initial capacity.

```
/*PROG 18.1 DEMO OF ARRAYLIST CLASS VER 1*/
import java.util.*;
class JPS1
{
    public static void main(String args[])
    {
        ArrayList AL1 = new ArrayList();
        System.out.println("\nInitial capacity of AL1:="+
        "+AL1.size()");
        AL1.add("one");
        AL1.add("two");
        ArrayList AL2 = new ArrayList(AL1);
```

```

        ArrayList AL3 = new ArrayList(5);
        System.out.println("Capacity of AL1:= "+AL1.size());
        System.out.println("Capacity of AL2:= "+AL2.size());
        System.out.println("Capacity of AL3:= "+AL3.size());
    }
}

OUTPUT:
Initial capacity of AL1:= 0
Capacity of AL1:= 2
Capacity of AL2:= 2
Capacity of AL3:= 0

```

Explanation: As stated earlier, capacity is the size of the array list, that is, the number of elements in it, so initial size of AL1 is displayed as 0. After adding two elements the size of AL1 becomes 2. The array list AL2 is initialized with AL1, that is, initializing a collection with a collection, and AL3 is constructed with initial capacity of 5 yet, size is displayed as 0.

```

/*PROG 18.2 DEMO OF ARRAYLIST CLASS VER 2 */

import java.util.*;
class JPS2
{
    public static void main(String args[])
    {
        ArrayList AL = new ArrayList();
        System.out.println("\nInitial size of AL:= "
                           + AL.size());
        AL.add("Moon");
        AL.add("Sun");
        AL.add("Stars");
        AL.add(1, "Earth");
        System.out.println("Size of AL after additions: "
                           + AL.size());
        System.out.println("Array List contains \n"+ AL);
        AL.remove("Sun");
        AL.remove(1);
        System.out.println("After removing two elements
                           size:= " + AL.size());
        System.out.println("Array List now contains \n"
                           + AL);
    }
}

OUTPUT:
Initial size of AL: = 0
Size of AL after additions: 4
Array List contains
[Moon, Earth, Sun, Stars]
After removing two elements size: = 2
Array List now contains
[Moon, Stars]

```

Explanation: This program constructs an empty array list AL and adds some elements to it using overloaded form of add method. After addition, size automatically increases. Note when displaying the array list, the reference AL is displayed which internally calls `toString` method. Removal of elements is done using remove method: one is through index and second is using elements itself. Note the list is displayed in its default form.

```
/*PROG 18.3 DEMO OF ARRAYLIST CLASS VER */

import java.util.*;
class JPS3
{
    public static void main(String args[])
    {
        ArrayList AL = new ArrayList();
        System.out.println("\nInitial size of AL: " +
                           + AL.size());
        AL.add(new String("Moon"));
        AL.add(new Integer(12));
        AL.add(new Date());
        AL.add(new Double(23.344));
        AL.add(new Boolean(true));
        AL.add(new Character('P'));
        System.out.println("Array List contains \n" +AL);
    }
}

OUTPUT
Initial size of AL: 0
Array List contains
[Moon, 12, Sat Dec 20 10:19:32 PST 2008, 23.344, true, P]
```

Explanation: In this program, object of different types are stored into the `ArrayList` AL. Rest is self-explanatory. In case, the list is to be displayed in a manner different from the above one, the `get` method can be used as:

```
for(int i = 0; i<AL.size();i++)
System.out.println("Element "+i+":"+AL.get(i));
```

The output of this will be:

```
Element 0: Moon
Element 1: 12
Element 2: Sat Dec 20 10:39:32 PST 2008
Element 3: 23.344
Element 4: true
Element 5: P
```

```
/*PROG 18.4 DEMO OF ARRAYLIST CLASS VER 4 */
```

```
import java.util.*;
class JPS4
{
```

```

public static void main(String args[])
{
    ArrayList AL = new ArrayList();
    int i, t, max;
    for(i=0;i<10;i++)
        AL.add(new Integer(i+1));
    System.out.println("\nArray list contains\n"+AL);
    Object obarr[] = AL.toArray();
    max = ((Integer)obarr[0]).intValue();
    for(i=1;i<obarr.length;i++)
    {
        t = ((Integer)obarr[i]).intValue();
        if(max < t)
            max = t;
    }
    System.out.println("\nMaximum of array is "+max);
}
}

```

Explanation: This program demonstrates how an `ArrayList` instance can be converted into an array. This may prove helpful in a number of programming situations. Initially in the `ArrayList` instance some integer objects are put. The `ArrayList` instance `AL` is converted into an array using `toArray` method of `ArrayList` class. The array returned is of type `Object` class. Later the maximum of 10 integers are found out by getting integer values from object elements using `intValue` method. Note one has to make sure in the beginning that `ArrayList` contains `Integer` objects.

18.4.1 Maintaining the Capacity

For the management of capacity, the `ArrayList` class provides two functions: `ensureCapacity` and `trimToSize()`

The signature of the method `ensureCapacity()` is given as:

```
public void ensureCapacity(int minCapacity)
```

The method increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. The reason for having this function is that in case one is sure in advance that `minCapacity` elements will be used in the `ArrayList` instance then this function can be used. Later when it is realized that capacity was assigned more than it was needed the capacity can be brought equal to the number of elements in the `ArrayList` instance using `trimToSize` function. The signature of the method is given as:

```
public void trimToSize()
```

The method trims the capacity of this `ArrayList` instance to be the list's current size. An application can use this operation to minimize the storage of an `ArrayList` instance.

18.4.2 The `LinkedList` Class

The declaration of `LinkedList` class given in Java is as follows:

```
public class LinkedList extends AbstractSequentialList
    implements List, Queue, Cloneable, Serializable
```

The class is a linked list implementation of the `List` interface. It implements all optional list operations, and permits all elements (including null). In addition to implementing the `List` interface, the `LinkedList` class

provides uniformly named methods to get, remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue or double-ended queue (deque).

The class has got two constructors.

1. **public LinkedList()**

This form of constructor creates an empty list:

2. **public LinkedList(Collection c)**

This form of constructor creates a list containing the elements of the specified collection **c**.

The various new methods (apart from the interfaces inherited) and other methods are shown in the program given below:

```
/*PROG 18.5 DEMO OF LINKED LIST CLASS VER 1 */

import java.util.*;
class JPS5
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        LinkedList LL = new LinkedList();
        show("\nInitial size of LL: " + LL.size());

        LL.addFirst(new Integer(1));
        LL.addFirst(new Integer(2));
        LL.addFirst(new Integer(3));
        LL.addFirst(new Integer(4));
        LL.addLast(new Float(2.5f));
        LL.addLast(new Float(12.5f));

        show("List is " + LL);
        show("First element:" + LL.getFirst());
        show("Last element:" + LL.getLast());

        show("Removed First element: "+LL.removeFirst());
        show("Removed Last element: " +LL.removeLast());

        show("List Now");
        for (int i = 0; i < LL.size(); i++)
            System.out.println("Element"+i+":"+LL.get(i));
    }
}

OUTPUT:
Initial size of LL: 0
List is [4, 3, 2, 1, 2.5, 12.5]
First element:4
Last element:12.5
```

```
Removed First element: 4
Removed Last element: 12.5
List Now
Element 0:3
Element 1:2
Element 2:1
Element 3:2.5
```

Explanation: The method `addFirst` adds the specified element at the beginning of the list, and `addLast` at the end of the list. Methods `getLast` and `getFirst` return as the last and the first element of the list, respectively. Methods `removeLast` and `removeFirst` remove from the list last and first element, respectively. In the end, using `get` method and `for` loop, the whole list is displayed.

```
/*PROG 18.6 DEMO OF LINKEDLIST CLASS VER 2 */

import java.util.*;
class JPS6
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        LinkedList LL1 = new LinkedList();
        for(int i=0;i<5;i++)
            LL1.addFirst(new Integer(i+1));
        LinkedList LL2 = new LinkedList(LL1);
        LL1.clear();
        show("\nList LL2 is "+LL2);
        show("List contains 2:"+LL2.contains(new
                                         Integer(2)));
        show("Index of 2 is "+LL2.indexOf(new Integer(2)));
        show("List LL2 before "+LL2);
        LL2.set(4,new Float(99.9999));
        LL2.add(3,new String ("added"));
        show("List LL2 now after add and set "+LL2);
    }
}

OUTPUT:

List LL2 is [5, 4, 3, 2, 1]
List contains 2:true
Index of 2 is 3
List LL2 before [5, 4, 3, 2, 1]
List LL2 now after add and set [5, 4, 3, added, 2, 99.9999]
```

Explanation: In this program, a `LinkedList` instance `LL1` is created and five integer objects added at the beginning of the `LL1`. A new `LinkedList` instance `LL2` is then created from `LL1`. `LL1` list is cleared using `clear` method. The `contains` method is then made use of to check whether list `LL2` contains 2 as object; the method returns true. Next, the index of the object 2 is displayed. The program also demonstrates the usage of `set` method for setting the value at specified index. Note the `addLast` method is equivalent to `add` method seen earlier. For adding element anywhere in the list, the methods `add(index, object)` form can be used.

18.5 ITERATING ELEMENTS OF COLLECTION

Iterating means accessing each element of the collection one by one; that is, by using an iterator, each of the element of the collection can be cycled through. The `iterator()` method of the `Collection` interface returns an `Iterator`. With the `Iterator` methods, a collection can be traversed from start to finish. The following code shows the use of the `Iterator` interface for a general `Collection`:

```
Collection collection = ....;
Iterator iterator = collection.iterator();
while(iterator.hasNext())
{
    Object element = iterator.next();
    System.out.println(element);
}
```

The iterator is obtained using `iterator` method of `Collection` interface. The method `hasNext()` returns till there is some element in the collection. The next element from the collection is obtained through `next` method. Afterwards any operation can be performed over the element.

The various methods of `Iterator` interface are shown in the Table 18.5.

| Method Signature | Description |
|--------------------------------|---|
| <code>boolean hasNext()</code> | Returns true if the iterator has more elements (In other words, returns true if next would return an element rather than throwing an exception). |
| <code>Object next()</code> | Returns the next element in the iteration. Calling this method repeatedly until the <code>hasNext()</code> method returns false will return each element in the underlying collection exactly once. |
| <code>void remove()</code> | Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to <code>next</code> . |

Table 18.5 Methods of `Iterator` interface

18.5.1 The `ListIterator` Interface

It is an iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. The `ListIterator` interface extends the `Iterator` interface. A `ListIterator` has no current element; its cursor position always lies between the element that would be returned by a call to `previous()` and the element that would be returned by a call to `next()`. In a list of length **n**, there are **n + 1** valid index values, from **0** to **n**, inclusive.

The various methods of this interface are given in the Table 18.6.

| Method Signature | Description |
|-----------------------|--|
| void add(object obj) | Inserts obj into the list in front of the element that will be returned by the next call to next(). |
| boolean hasNext() | Returns true if there is a next element. Otherwise, returns false. |
| boolean hasPrevious() | Returns true if there is a previous element. Otherwise, returns false. |
| Object next() | Returns the next element. A NoSuchElementException is thrown if there is not a next element. |
| int nextIndex() | Returns the index of the next element. If there is not a next element, returns the size of the list. |
| Object previous() | Returns the previous element. A NoSuchElementException is thrown if there is not a previous element. |
| Int previouslyIndex() | Returns the index of the previous element. If there is not a previous element, returns -1. |
| Void remove() | Remove the current element from the list. An IllegalStateException is thrown if remove() is called before next() or previous() is invoked. |
| Void set(Object obj) | Assign obj to the current element. This is the element last returned by a call to either next() or previous() |

Table 18.6 Methods of ListIterator interface

To work with iterator the three steps has to be followed:

1. Obtain an iterator. This is done using iterator method of the collection. For example, for an instance of ArrayList AL iterator can be obtained as:

```
Iterator itr = AL.iterator();
```
2. Use a loop that runs as long as hasNext() returns true. This is done as:

```
while(itr.hasNext())
```
3. Inside the loop, obtain the next element using next method.

```
/*PROG 18.7 DEMO OF ITERATING COLLECTION VER 1*/
```

```
import java.util.*;
class JPS7
{
    public static void main(String args[])
    {
        ArrayList AL = new ArrayList();
        for (int i = 0; i < 10; i++)
            AL.add(new Integer(i + 1));
        Iterator itr = AL.iterator();
        while (itr.hasNext())
        {
            Object E = itr.next();
            System.out.print(E + " ");
        }
    }
}
```

```

        System.out.println();
    }
}

```

OUTPUT:

```
1 2 3 4 5 6 7 8 9 10
```

Explanation: This program creates an `ArrayList` instance and adds **10** `Integer` objects to it using `for` loop and `add` method. An iterator to the collection `ArrayList` instance is obtained using `iterator` method and stored in `Iterator` reference `itr`. In the `while` loop, `hasNext` method returns true till there is an element in the `ArrayList AL`. The next element is obtained using `next` method and stored in `E`. The same is displayed back.

```
/*PROG 18.8 DEMO OF ITERATING COLLECTION VER 2*/
```

```

import java.util.*;
class JPS8
{
    public static void main(String args[])
    {
        LinkedList LL = new LinkedList();
        for (int i = 0; i < 5; i++)
            LL.add(new Float(2.5f + i));
        System.out.println("\nAccessing List in forward
                           direction");
        ListIterator itr = LL.listIterator();
        while (itr.hasNext())
        {
            Object E = itr.next();
            System.out.print(E + " ");
        }
        System.out.println();
        System.out.println("\nAccessing List in backward
                           direction");
        while (itr.hasPrevious())
        {
            Object E = itr.previous();
            System.out.print(E + " ");
        }
        System.out.println();
    }
}
```

OUTPUT:

```
Accessing List in forward direction
2.5 3.5 4.5 5.5 6.5
Accessing List in backward direction
6.5 5.5 4.5 3.5 2.5
```

Explanation: This program demonstrates `ListIterator` and `listIterator` method. An instance of `LinkedList` class is created and is populated with five `float` objects. For accessing the list in the forward direction, the simple iterating steps can be followed. Note when list is displayed in the forward direction, the iterator will be past the end of the list. Now the list can be displayed in the reverse direction. In the `while` loop using `hasPrevious` method it is checked whether any previous element is present or not. The previous element is obtained in the body of the `while` loop and is displayed.

```
/*PROG 18.9 DEMO OF ITERATING COLLECTION VER 3 */

import java.util.*;
class JPS9
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        LinkedList LL = new LinkedList();
        for (int i = 0; i < 5; i++)
            LL.add(new Float(2.5f + i));
        show("List contents: " + LL);
        show("Modifying List using iterator");
        ListIterator itr = LL.listIterator();
        while (itr.hasNext())
        {
            Object E = itr.next();
            float val = ((Float)E).floatValue();
            LL.set(LL.indexOf(E), new Float(val + 10));
        }
        show("List after modification");
        for (itr = LL.listIterator(); itr.hasNext(); )
            show(itr.next() + " ");
    }
}
OUTPUT:
List contents: [2.5, 3.5, 4.5, 5.5, 6.5]
Modifying List using iterator
List after modification
12.5
13.5
14.5
15.5
16.5
```

Explanation: In this program, the contents of the `LinkedList` instance `LL` are modified. For that using a `listIterator` and `next` method, each method is obtained as `Object` instance `E` of the list. Using typecasting with `Float` and `floatValue()` method of `Float` class, the associated float value is obtained in `val`. The index of the object `E` is obtained by using `indexOf` method. At this index, a new `Float` object is stored with `val+10` as its float value. This is done by using `set` method.

```
/*DEMO OF ITERATING COLLECTION VER 4 */

import java.util.*;
class Book
{
    String bname;
    String author;
    int price;
    Book(String bn, String a, int p)
    {
        bname = bn;
        author = a;
        price = p;
    }
    public String toString()
    {
        String s = "Name = " + bname + "\tAuthor=" + author;
        s = s + "\tPrice=" + price;
        return s;
    }
}
class JPS10{
    public static void main(String args[])
    {
        LinkedList LL = new LinkedList();
        LL.add(new Book("Sea of C ","Hari Mohan ",295));
        LL.add(new Book("Sea of C++ ","Hubbured ",100));
        LL.add(new Book("Sea Of Java","H.M.Pandey ",400));

        ListIterator itr;
        System.out.println("\n*****Book Details
                           *****\n");
        for(itr = LL.listIterator();itr.hasNext();)
            System.out.println(itr.next()+" ");
    }
}

OUTPUT:
*****Book Details *****
Name = Sea of C      Author=Hari Mohan   Price=295
Name = Sea of C++     Author=Hubbured    Price=100
Name = Sea Of Java    Author=H.M.Pandey  Price=400
```

Explanation: Similar to storing the objects of built-in classes, the objects of user-defined classes can be stored. In this program, a class **Book** has been defined which is having three members: **bname**, **author** and **price**. The class is having a parameterized constructor which initializes these members. The class also overrides **toString** method so that it can be used for displaying objects of this class as String object. In the main, the objects of Book class is added to the **LinkedList** instance **LL** using the parameterized constructor form. The objects are displayed as usual using for/while loop.

18.6 HASHSET AND TREESET CLASSES

The Collections framework provides two general-purpose implementations of the Set interface: HashSet and TreeSet. Both are discussed in turn.

18.6.1 HashSet Class

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantee as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. More often than not, a HashSet will be used for storing the duplicate-free collection. For efficiency, objects added to a HashSet need to implement the `hashCode()` method in a manner that properly distributes the hash codes. While most system classes override the default `hashCode()` implementation in `Object`, when creating your own classes to add to a HashSet remember to override `hashCode()`.

The `hashcode` is generated by a mechanism known as hashing. In hashing, on the basis of data a key value is generated by the use of some hashing functions. This key value is known as `hashecode`. The `hashcode` is determined internally and one should have no concern with it.

The advantage of hashing is that it allows the execution time of basic operations, such as `add()`, `contains()`, `remove()` and `size()`, to remain constant even for large sets.

Commonly used constructors of this class are shown below:

- 1. public HashSet()**

This form of constructor creates an empty HashSet.

- 2. public HashSet(Collection c)**

This form of constructors creates a new set containing the elements in the specified collection `c`.

- 3. public HashSet(int initialCapacity)**

This form of constructor creates a new, empty set that has the specified initial capacity.

The class defines no new method of its own. An example of usage of HashSet is given below.

```
/*PROG 18.11 DEMO OF HASHSET CLASS VER 1 */

import java.util.*;
class JPS11
{
    public static void main(String args[])
    {
        HashSet HS = new HashSet();
        HS.add(new Integer(1));
        HS.add(new Integer(12));
        HS.add(new Integer(13));
        HS.add(new Integer(24));
        HS.add(new Integer(50));
        HS.add(new Integer(50));
        HS.add(new Integer(24));
        System.out.println("\nContents of HashSet HS:"+HS);
    }
}

OUTPUT:
Contents of HashSet HS: [50, 1, 24, 12, 13]
```

Explanation: This program is simple to understand. One important thing to note here is that elements of HashSet are not ordered and duplicate elements are displayed only once.

18.6.2 TreeSet Class

This class implements the Set and SortedSet interface. As the name implies, the class uses the tree for storage of its elements. The TreeSet implementation is useful when one needs to extract elements from a collection in a sorted manner. In order to work properly, elements added to a TreeSet must be sortable. A tree knows how to keep elements of the java.lang wrapper classes sorted. It is generally faster to add elements to a HashSet, and then convert the collection to a TreeSet for sorted traversal.

The class defines the following constructors:

1. **public TreeSet()**

This form of constructor creates a new empty set, sorted according to the elements' natural order. All elements inserted into the set must implement the Comparable interface. Furthermore, all such elements must be mutually comparable: `e1.compareTo(e2)` must not throw a `ClassCastException` for any element `e1` and `e2` in the set. If the user attempts to add an element to the set that violates this constraint (e.g., the user attempts to add a string element to a set whose elements are integers), the `add(object)` call will throw a `ClassCastException`.

2. **public TreeSet(Collection c)**

This form of constructor creates a new set containing the elements in the specified collection `c`, sorted according to the elements' natural order.

3. **public TreeSet(Comparator c)**

This form of constructor creates a new, empty set, sorted according to the specified comparator (discussed later).

4. **public TreeSet(SortedSet s)**

This form of constructor creates a new set containing the same elements as the specified sorted set, sorted according to the same ordering.

The new methods of TreeSet classes are given in the Table 18.7.

| Method Signature | Description |
|--|---|
| <code>Comparator comparator()</code> | Returns the comparator used to order this sorted set, or null if this tree set uses its elements' natural ordering. |
| <code>Object first()</code> | Returns the first (lowest) element currently in this sorted set. |
| <code>Object last()</code> | Returns the last (highest) element currently in this sorted set. |
| <code>SortedSet tailSet (Order fromElement)</code> | Returns a view of the portion of this set whose elements is greater than or equal to fromElement. The returned sorted set is backed by this set, so changes in the returned sorted set are reflected in this set, and vice versa. The returned sorted set supports all optional set operations. |
| <code>SortedSet headSet (Object toElement)</code> | Returns a view of the portion of this set whose elements are strictly less than toElement. The returned sorted set is backed by this set, so changes in the returned sorted set are reflected in this set, and vice versa. The returned sorted set supports all optional set operations. |

Table 18.7 Methods of TreeSet class

See few programming examples of usage of **TreeSet**.

```
/*PROG 18.12 DEMO OF TREESSET CLASS VER 1*/
import java.util.*;
class JPS12
{
    public static void main(String args[])
    {
        TreeSet TS = new TreeSet();
        TS.add(new Integer(1));
        TS.add(new Integer(12));
        TS.add(new Integer(13));
        TS.add(new Integer(24));
        TS.add(new Integer(50));
        TS.add(new Integer(50));
        TS.add(new Integer(24));
        System.out.println("\nContents of TreeSet TS:" + TS);
    }
}
OUTPUT:
Contents of TreeSet TS: [1, 12, 13, 24, 50]
```

Explanation: This program is simple to understand. Important thing to note here is that duplicate elements are not present and elements are sorted in ascending order.

```
/*PROG 18.13 DEMO OF TREESSET CLASS VER 2 */
import java.util.*;
class JPS13
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        TreeSet TS = new TreeSet();
        TS.add(new Integer(1));
        TS.add(new Integer(12));
        TS.add(new Integer(13));
        TS.add(new Integer(24));
        TS.add(new Integer(20));
        TS.add(new Integer(60));
        TS.add(new Integer(56));

        Iterator itr;
        show("\nElements of TS");
        for(itr = TS.iterator();itr.hasNext();)
```

```

        System.out.print(itr.next() + " ");
        show("\nFirst element:" + TS.first());
        show("\nLast element:" + TS.last());
        show("\nSub set of TS");

        SortedSet SS = TS.subSet(new Integer(12),
                               new Integer(24));
        for (itr = SS.iterator(); itr.hasNext(); )
            System.out.print(itr.next() + " ");

        show("\nHead set of TS");
        SS = TS.headSet(new Integer(24));
        for (itr = SS.iterator(); itr.hasNext(); )
            System.out.print(itr.next() + " ");
        show("\nTail set of TS");
        SS = TS.tailSet(new Integer(24));
        for (itr = SS.iterator(); itr.hasNext(); )
            System.out.print(itr.next() + " ");
    }
}

OUTPUT:

Elements of TS
1 12 13 20 24 56 60
First element:1
Last element:60
Sub set of TS
12 13 20
Head set of TS
1 12 13 20
Tail set of TS
24 56 60

```

Explanation: This program demonstrates some of the methods of TreeSet class. After adding elements to the TreeSet instance TS, they are displayed using iterator. The first element (lowest) is returned using `first()` method and `last()` method returns the last (highest) element from the TS. The method `subset` returns the number of elements between the specified two arguments. The first argument is inclusive but second is not. In the output, all elements between 12 and 24 are displayed. As mentioned in the description part of this method, the set returned is part of the original set; no new set is returned. The method `headSet` returns all elements greater than the specified element. The method `tailSet` returns all elements smaller than the specified element.

```
/*PROG 18.14 DEMO OF TREESSET CLASS VER 3*/
```

```

import java.util.*;
class JPS14
{
    static void show(String s)
    {
        System.out.println(s);
    }
}

```

```

    }
    public static void main(String args[])
    {
        HashSet HS = new HashSet();
        HS.add("Hari");
        HS.add("Hemangi");
        HS.add("Varsha");
        HS.add("Malvika");
        HS.add("Deshmukh");

        show("\nThe HashSet elements\n");
        show(HS + " ");

        TreeSet TS = new TreeSet(HS);
        show("\nThe TreeSet elements\n");
        show(TS + " ");
    }
}

OUTPUT:

The HashSet elements
[Varsha, Deshmukh, Hemangi, Malvika, Hari]
The TreeSet elements
[Deshmukh, Hari, Hemangi, Malvika, Varsha]

```

Explanation: In this program, a HashSet instance HS is created and populated with some String objects. The elements of HS are then displayed. An instance TS of TreeSet is then constructed using HS as argument for the TreeSet constructor. In general programming, this is a preferred method of creating an ordered set.

18.7 WORKING WITH MAPS

A map is used to store key-value pairs, with the values retrieved using key. For each given key, there will be a corresponding value. In essence, each element of the map is stored as a (key, value) pair. Given a key, a value can be retrieved. This is the most powerful feature of map. Both key and value are objects. The keys must be unique but values may be duplicated.

The Collection Framework defines Map, Map.Entry and SortedMap interfaces to work with map.

18.7.1 The Map Interface

The Map interface is not an extension of the Collection interface. Instead, the interface starts off its own interface hierarchy for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition. The methods of this interface are given in the Table 18.8.

| Method Signature | Description |
|---------------------------------|---|
| void clear() | Removes all key-value pairs from the invoking map. |
| boolean containsKey(Object k) | Returns “true” if the invoking map contains k as a key. Otherwise, returns false. |
| Boolean containsValue(Object v) | Returns “true” if the map contains v as a value. Otherwise, returns “false”. |

| | |
|---|---|
| <code>Set entrySet()</code> | Returns a set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a setview of the invoking map. |
| <code>boolean equals(Object obj)</code> | Returns “true” if obj is a Map and contains the same entries. Otherwise, returns “false”. |
| <code>Object get(Object k)</code> | Returns the value associated with the key k. |
| <code>int hashCode()</code> | Returns the hash code for the invoking map. |
| <code>boolean isEmpty()</code> | Returns “true” if the invoking map is empty. Otherwise, returns “false”. |
| <code>Set KeySet()</code> | Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map. |
| <code>Object put(Object k, Object v)</code> | Puts an entry in the invoking map, overwritten any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned. |
| <code>void putAll(Map m)</code> | Puts all the entries from m into this map. |
| <code>Object remove(Object k)</code> | Removes the entry whose key equals k. |
| <code>Int size()</code> | Returns the number of key-value pairs in the map. |
| <code>Collection values()</code> | Returns a collection containing the values in the map. This method provides a collection-view of the values in the map. |

Table 18.8 Methods of Map interface

The interface methods can be broken down into three sets of operations: altering, querying and providing alternative views.

The alteration operations allow one to add and remove key-value pairs from the map. Both the key and value can be null. However, a Map should not be added to itself as a key or value.

```
Object put(object key, Object value)
Object remove(Object key)
void putAll(Map mapping)
void clear()
```

The query operations allow one to check on the contents of the map:

```
Object get(Object key)
boolean containskey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()
```

The last set of methods allows one to work with the group of keys or values as a collection.

```
public Set keySet()
public Collection values()
public Set entrySet()
```

Because the collection of keys in a map must be unique, a Set back is obtained. Because the collection of values in a map should not be unique, a Collection back is obtained. The last method returns a set of elements that implements the Map.Entry interface.

18.7.2 Map.Entry Interface

The `entrySet()` method of `Map` returns a collection of objects that implements the `Map.Entry` interface. Each object in the collection is a specific key-value pair or `Map.entry` object in the underlying `Map`. The methods of this interface are given in the Table 18.9.

| Method Signature | Description |
|---|--|
| <code>Boolean equals(Object obj)</code> | Returns “true” if <code>obj</code> is a <code>Map-Entry</code> whose key and value are equal to that of the invoking object. |
| <code>Object getKey()</code> | Returns the key for this map entry. |
| <code>Object getValue()</code> | Returns the value for this map entry. |
| <code>int hashCode()</code> | Returns the hash code for this map entry. |
| <code>Object setValue(Object v)</code> | Replaces the value corresponding to this entry with the specified value. |

Table 18.9 Methods of `Map.Entry` interface

Iterating through this collection, the key or value can be obtained, and the value of each entry can be changed as well. However, the set of entries becomes invalid, causing the iterator behaviour to be undefined, if the underlying `Map` is modified outside the `setValue()` method of the `Map.Entry` interface.

18.7.3 The SortedMap Interface

The `SortedMap` interface extends `Map` interface. It is a map that further guarantees that it will be in ascending key order, sorted according to the natural ordering of its keys (see the `Comparable` interface), or by a comparator provided at sorted map creation time. The interface provides access method to the ends of the `Map` as well as to subsets of the `Map`. Working with a `SortedMap` is just like a `SortedSet`, except the sort is done on the `Map` keys. The implementation class provided by the Collections Framework is a `TreeMap`.

The methods of this interface are given in the Table 18.10.

| Method Signature | Description |
|---|---|
| <code>Comparator comparator()</code> | Returns the invoking sorted map’s comparator. If the natural ordering is used for the invoking map, null is returned. |
| <code>Object firstKey()</code> | Returns the first key in the invoking map. |
| <code>SortedMap headMap(Object end)</code> | Returns a sorted map for those map entries with keys that are less than end. |
| <code>Object lastKey()</code> | Returns the last key in the invoking map. |
| <code>SortedMap subMap(Object start, Object end)</code> | Returns a map containing those entries with keys that are greater than or equal to start and less than end. |
| <code>SortedMap tailMap(Object start)</code> | Returns a map containing those entries with keys that are greater than or equal to start. |

Table 18.10 Methods of `SortedMap` interface

18.8 WORKING WITH MAP CLASSES

The interfaces discussed in the previous section are implemented by the following Map classes: AbstractMap, HashMap, TreeMap and WeakHashMap. All these classes are discussed below.

1. AbstractMap class

This class provides a skeletal implementation of the Map interface, to minimize the efforts required to implement this interface. The AbstractMap class is the base class for all the other three classes.

2. HashMap and TreeMap classes

The Collection Framework provides two general-purpose Map implementations: HashMap and TreeMap. As with all the concrete implementations, which implementation is being used depends on the specific needs. For inserting, deleting and locating elements in a Map, the HashMap offers the best alternative. If, however, the keys need to be traversed in a sorted order TreeMap is the better alternatives. Depending on the size of the collection, it may be faster to add elements to a HashMap convert the map to a TreeMap for sorted key traversal. Using a HashMap requires that the class of key added have a well-defined hashCode () implementation. With the TreeMap implementation, elements added to the map must be sortable.

The base class of HashMap class is the AbstractMap class and it implements Serializable, Cloneable and Map interface.

The class HashMap defines following constructor:

1. **public HashMap()**

This form of constructor creates an empty HashMap with the default initial capacity (17) and the default load factor (0.75).

2. **public HashMap(Map m)**

This form of constructor creates a new HashMap with the same mappings as the specified Map. The HashMap is created with default load factor (0.75) and an initial capacity sufficient to hold the mappings in the specified Map.

3. **public HashMap(int initialCapacity)**

This form of constructor creates an empty HashMap with the specified initial capacity and the default load factor (0.75).

4. **public hashMap(int initialCapacity, float loadFactor)**

This form of constructor creates an empty HashMap with the specified initial capacity and load factor.

Similar to a HashSet, a HashMap does not store its elements in sorted order.

```
/*PROG 18.15 DEMO OF HASHMAP CLASS VER 1 */

import java.util.*;
class JPS15
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        HashMap HM = new HashMap();
        HM.put("Hari Mohan", new Integer(101));
    }
}
```

```

HM.put("Man Mohan ", new Integer(102));
HM.put("Ranjana ", new Integer(103));
HM.put("Anjana ", new Integer(104));
Set s = HM.entrySet();
show("The elements of the Map are \n");
Iterator itr = s.iterator();
show("KEY \t\t VALUES");
while (itr.hasNext())
{
    Map.Entry me = (Map.Entry)itr.next();
    show(me.getKey() + "\t" + me.getValue());
}
}

OUTPUT:
The elements of the Map are
KEY          VALUES
Hari Mohan   101
Anjana       104
Ranjana      103
Man Mohan    102

```

Explanation: This program creates an instance of HashMap and using put method puts the key and values into the map instance HM. The values are the String objects, that is, names and keys are points obtained during a game. The collection of all the map entries as a Set is returned using entrySet method. As stated earlier in the description of methods, each entry of this set is an object of type Map.Entry. Next elements of set s are displayed using an iterator itr. The while loop runs till there is an element in the set s. The reference of the object returned through next method is typecasted by Map.Entry interface and stored in me of type Map.Entry. Then using getValue and getKey methods of Map.Entry interface the elements are displayed.

```

/*PROG 18.16 DEMO OF HASHMAP CLASS VER 2 */

import java.util.*;
class JPS16
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        HashMap HM = new HashMap();
        HM.put(" Hari Mohan", new Integer(101));
        HM.put(" Man Mohan ", new Integer(102));
        HM.put(" Ranjana ", new Integer(103));
        HM.put(" Anajana ", new Integer(104));

        Set s = HM.keySet();
    }
}
```

```

Iterator itr;
show("\n Keys of Map\n");
for (itr = s.iterator(); itr.hasNext(); )
    show(itr.next() + " ");
Collection C = HM.values();

show("\n Values of Map\n");
for (itr = C.iterator(); itr.hasNext(); )
    show(itr.next() + " ");
}
}

OUTPUT:

Keys of Map
Hari Mohan
Ranjana
Man Mohan
Anajana
Values of Map

101
103
102
104

```

Explanation: The elements of HashMap HM are the same as in the previous program. Here keys and values are obtained separately using the keySet and values method, respectively. The keySet method returns all the keys as a set. These are then displayed using an iterator to the set s. The values method returns a reference of Collection type. This is stored in C and then displayed using iterator values.

3. The TreeMap class

The base class of the TreeMap is the AbstractMap class and it implements SortedMap, Cloneable and Serializable interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class. This implementation provides guaranteed $\log(n)$ time cost for the containsKey, get, put and remove operations.

The class defines the following constructors:

1. **public TreeMap()**

This form of constructor creates a new, empty **map**, sorted according to the key's natural order. All keys inserted into the map must implement the **Comparable** interface. Furthermore, all such keys must be mutually comparable: **k1.compareTo(k2)** must not throw a **ClassCastException** for any elements **k1** and **k2** in the map. If the user attempts to put a key into the map that violates this constraint (e.g., the user attempts to put a string key into a map whose keys are integers), the **put(Object key, Object value)** call will throw a **ClassCastException**.

2. **public TreeMap(Comparator c)**

This form of constructor creates a new empty map, sorted according to the given comparator. All keys inserted into the map must be mutually comparable by the given comparator: **comparator.compare(k1, k2)** must not throw a **ClassCastException** for any keys **k1** and **k2** in the map. If the user attempts to put a key into the map that violates this constraint, the **put(Object key, Object value)** call will throw a **ClassCastException**.

3. **public TreeMap(Map m)**

This form of constructor creates a new map containing the same mappings as the given map, sorted according to the key's natural order.

4. **public TreeMap(SortedMap m)**

This form of constructor creates a new map containing the same mappings as the given SortedMap, sorted according to the same ordering.

```
/*PROG 18.17 DEMO OF TREEMAP CLASS VER 1 */

import java.util.*;
class JPS17
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        TreeMap TM = new TreeMap();
        TM.put("CS02", "Hari");
        TM.put("CS01", "Man");
        TM.put("CS04", "Ranjana");
        TM.put("CS03", "Anjana");

        Set s = TM.entrySet();
        show("\nThe elements of the Map are \n");
        Iterator itr = s.iterator();
        show("KEY\t VALUES");
        while (itr.hasNext())
        {
            Map.Entry me = (Map.Entry)itr.next();
            show(me.getKey() + "\t" + me.getValue());
        }
    }
}

OUTPUT:
The elements of the Map are

KEY      VALUES
CS01      Man
CS02      Hari
CS03      Anjana
CS04      Ranjana
```

Explanation: This program is similar to the earlier programs of HashMap. Note that output of the program is in sorted order.

```
/*PROG 18.18 DEMO OF TREEMAP CLASS VER 2*/
import java.util.*;
class JPS18
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        TreeMap TM = new TreeMap();
        for(int i=101;i<105;i++)
            TM.put(new Integer(i), new Integer(100));

        Set s = TM.entrySet();
        Iterator itr = s.iterator();
        Map.Entry me = (Map.Entry)itr.next();
        me.setValue(new Integer(95));
        Object obj = TM.get(new Integer(101));
        show("\nValue is "+obj);

        boolean k = TM.containsKey(new Integer(106));
        boolean v = TM.containsKey(new Integer(100));

        show("\nk = "+k+"\tv= "+v);
        TM.remove(new Integer(101));
        show("\nSize of TreeMap instance TM is
                "+TM.size());
    }
}
OUTPUT:
Value is 95
k = false v= false
Size of TreeMap instance TM is 3
```

Explanation: This program shows some of the methods of TreeMap class. Initially some elements are put in the TreeMap instance TM. Both key and value are Integer objects. Assume key is the roll number and values are marks; the first element of the TM is obtained using iterator. Its values are changed using setValue method of Map.Entry interface. The same value is then obtained using get method. The method containsKey and containsValue checks whether a key and value supplied as argument are within the map. If they are within the map, the method returns true else, returns false. The remove method removes the given object from the map and size method gives number of elements in the map.

```
/*PROG 18.19 DEMO OF TREEMAP CLASS VER 3 */
```

```
import java.util.*;
class JPS19
{
```

```

static void show(String s)
{
    System.out.println(s);
}
    public static void main(String args[])
{
    TreeMap TM = new TreeMap();
    TM.put(new Integer(101), new Double(34.56));
    TM.put(new Integer(102), new Double(44.56));
    TM.put(new Integer(105), new Double(35.77));
    TM.put(new Integer(106), new Double(45.78));
    TM.put(new Integer(104), new Double(30.59));
    TM.put(new Integer(107), new Double(28.75));
    TM.put(new Integer(103), new Double(25.90));

    show("\nFirst key:" + TM.firstKey());
    show("Last key:" + TM.lastKey());

    SortedMap SM = TM.headMap(new Integer(104));
    Iterator itr;
    Set s = SM.entrySet();
    show("\nKEY\tVALUES for key <104");
    for (itr = s.iterator(); itr.hasNext(); )
    {
        Map.Entry me = (Map.Entry)itr.next();
        show(me.getKey() + "\t" + me.getValue());
    }
    SM = TM.tailMap(new Integer(104));
    s = SM.entrySet();
    show("\nKEY\tVALUES for key >104");
    for (itr = s.iterator(); itr.hasNext(); )
    {
        Map.Entry me = (Map.Entry)itr.next();
        show(me.getKey() + "\t" + me.getValue());
    }
}
}

OUTPUT:
First key:101
Last key:107

KEY      VALUES for key <104
101      34.56
102      44.56
103      25.9

KEY      VALUES for key >104
104      30.59
105      35.77
106      45.78
107      28.75

```

Explanation: This program shows few methods of `SortedMap` interface. The `firstKey` and `lastKey` method returns the first and last key of the specified map. The `headMap` returns a map where all the keys of map entries are less than the specified key. The `tailMap` does the reverse of `headMap`.

18.9 THE COMPARATOR INTERFACE

The `Comparator` interface is a type of comparison function, which imposes a total ordering on some collection of objects. By default, Java uses the natural ordering of the elements for the comparison purpose, that is, 8 comes before 9, C comes before D and so on. The natural order can be altered for comparison purpose. Comparators can be passed to a sort method to allow precise control over the sort order. Comparator can also be used to control the order of certain data structures (such as `TreeSet` or `TreeMap`). The interface defines two methods: `compare` and `equals`.

The signature of `compare` method is given as:

```
int compare(Object O1, Object O2)
```

The method compares its two arguments for order. It returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. The method can throw a `ClassCastException` if the types of the objects are not compatible for comparison. Overriding `compare` method lets one alter the natural ordering, for example, printing elements in reverse order.

The signature of `equals` method is given as:

```
boolean equals(Object Obj)
```

The method indicates whether some other object is “equal to” this `Comparators`. If it is, it returns true else, returns false. This method can return true only if the specified `Object` is also a comparator and it imposes the same ordering as this comparator. This method is usually not overridden.

The following example helps to better understand the usage of `Comparator` interface.

```
/*PROG 18.20 DEMO OF COMPARATOR INTERFACE VER 1 */
```

```
import java.util.*;
class MyComp implements Comparator
{
    public int compare(Object a, Object b)
    {
        Integer ia, ib;
        ia = (Integer)a;
        ib = (Integer)b;
        return ib.compareTo(ia);
    }
}
class JPS20
{
    public static void main(String args[])
    {
        TreeSet TS = new TreeSet(new MyComp());
        TS.add(new Integer(10));
        TS.add(new Integer(18));
        TS.add(new Integer(105));
        TS.add(new Integer(210));
```

```

        TS.add(new Integer(149));
        TS.add(new Integer(330));
        TS.add(new Integer(75));

        System.out.println("\nThe sorted elements are\n");
        Iterator itr;
        for (itr = TS.iterator(); itr.hasNext(); )
            System.out.print(itr.next() + " ");
        System.out.println();
    }
}

OUTPUT:

The sorted elements are

330 210 149 105 75 18 10

```

Explanation: To make use of Comparator, a class must be created and Comparator interface implemented. In this program, a class MyComp is created that implements Comparator interface. In the class, the compare method is overridden. The default compare method uses compareTo method for comparison of two objects. As stated earlier, the default method uses natural ordering. But in this program, the ordering for comparing of two Integer objects is reversed. Note in the default form first object calls the compareTo method and passes second object as argument but in this case the order is reversed. Furthermore, depending on the type of object the compare method typecasting has to be done. Here the Object is typecasted to Integer.

In the main, when creating an instance of TreeSet, in the constructor an object of MyComp class is passed. This causes to use own comparator for comparison purpose. Note the output is in reverse order and the default order is ascending.

```
/*PROG 18.21 DEMO OF COMPARATOR INTERFACE VER 2 */
```

```

import java.util.*;
class MyComp implements Comparator
{
    public int compare(Object a, Object b)
    {
        String ia, ib;
        ia = ((String)a).toLowerCase();
        ib = ((String)b).toLowerCase();
        return ib.compareTo(ia);
    }
}
class JPS21
{
    public static void main(String args[])
    {
        TreeSet TS = new TreeSet(new MyComp());
        TS.add("Hari");
        TS.add("Vijay");
        TS.add("Madhuri");
    }
}

```

```

        TS.add("Man");
        TS.add("Vikas");
        System.out.println("\nSorted names are\n");

        Iterator itr;
        for (itr = TS.iterator(); itr.hasNext(); )
            System.out.print(itr.next() + " ");
        System.out.println();
    }
}

OUTPUT:

Sorted names are
Vikas Vijay Man Madhuri Hari

```

Explanation: In this program, the compare method is overridden so two String objects are compared irrespective of their case, that is, ‘Hari’ and ‘hari’ will be treated as same. For that, both the strings are converted to lower case and compareTo method is applied. Note the output without using own comparator will be:

Vikas Vijay Man Madhuri Hari

```

/*PROG 18.22 DEMO OF COMPARATOR INTERFACE 3 */

import java.util.*;
class MyComp implements Comparator
{
    public int compare(Object a, Object b)
    {
        Integer ia, ib;
        ia = (Integer)a;
        ib = (Integer)b;
        return ib.compareTo(ia);
    }
}
class JPS22
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        TreeMap TM = new TreeMap(new MyComp());
        TM.put(new Integer(101), "Hari");
        TM.put(new Integer(104), "Man");
        TM.put(new Integer(210), "Ravi");
        TM.put(new Integer(150), "Vijay");
        TM.put(new Integer(340), "Hemangi");

        Iterator itr;

```

```

        Set S = TM.entrySet();
        show("\nKEY\tVALUES");
        for (itr = S.iterator(); itr.hasNext(); )
        {
            Map.Entry me = (Map.Entry)itr.next();
            show(me.getKey() + "\t" + me.getValue());
        }
        show(" ");
    }

OUTPUT:
KEY      VALUES
340      Hemangi
210      Ravi
150      Vijay
104      Man
101      Hari

```

Explanation: This program is similar to the previous one but in this case a comparator for the TreeMap instance is written. The default comparator does the sorting on the basis of natural ordering but the natural ordering is reversed. This is quite clear from the output of the program.

18.10 HISTORICAL COLLECTION CLASSES

The historical collection classes are those classes supplied with the earlier version of Java. Though the Collection Framework provides a number of classes which can be used in different programming situations, at times one still needs to use some of the original collections capabilities. This section reviews some of the capabilities of working with arrays, vectors, hashtables, enumerations and other historical capabilities.

1. The Enumeration interface

The Enumeration interface allows one to iterate through all the elements of a collection. In the Collection Framework, this interface has been superseded by the Iterator interface. However, not all libraries support the newer interface, so Enumeration may have to be used from time to time.

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the `nextElement` method return successive elements of the series.

For example, to print all elements of a vector `V` the following can be written:

```
for(Enumeration e = V.elements();e.hasMoreElements())
    System.out.println(e.nextElement());
```

The interface defines only two methods:

1. **boolean hasMoreElements()**

The method tests if this enumeration contains more elements. It returns true if enumeration contains more elements else, false.

2. **Object nextElement()**

The method returns the next element of this enumeration if this enumeration object has at least one more element to provide.

2. The Vector class

The vector class implements a growable array of objects, but can store heterogeneous data elements. Like an array, it contains components that can be accessed using an integer index. However, the size of a vector

can grow or shrink as needed to accommodate adding and removing items after the vector has been created. With the Java 2 SDK, version 2, the Vector class has been retrofitted into the Collections Framework hierarchy to implement the List interface.

When transitioning from `Vector` to `ArrayList`, one key difference is that the arguments have been reversed to positionally change an element's value:

- (i) From original `Vector` class `void setElementAt(Object element, int index)`
- (ii) From `List` interface `void set(int index, Object element)`

One more difference to note is that `Vector` is synchronized but `ArrayList` is not.

Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size; it is usually large because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

The `Vector` class defines the following constructors:

1. **`public Vector()`**

This form of constructor creates an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.

2. **`public Vector(int initialCapacity)`**

This form of constructor creates an empty vector with the specified initial capacity and with its capacity increment equal to zero.

3. **`public Vector(int initialCapacity, int CapacityIncrement)`**

This form of constructor creates an empty vector with the specified initial capacity and capacity increment.

4. **`public Vector(Collection c)`**

This form of constructor creates a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Apart from the above four constructors, the `Vector` class defines the following data members.

Table 18.11

- **`protected int capacityIncrement`**

It is the amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity. If the capacity increment is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

- **`protected Object[] elementData`**

It is the array buffer into which the components of the vector are stored. The capacity of the vector is the length of this array buffer, and is at least large enough to contain all the vector's elements. Any array elements following the last element in the `Vector` are null.

- **`protected int elementCount`**

It is the number of valid components in this `Vector` object. Components `elementData[0]` through `elementData[elementCount-1]` are the actual items.

| Method Signature | Description |
|--|--|
| <code>final void addElement(Object element)</code> | The object specified by <code>element</code> is added to the vector. |
| <code>final int capacity()</code> | Returns the capacity of the vector. |
| <code>Object clone()</code> | Returns a duplicate of the invoking vector. |

(Continued)

| | |
|--|--|
| <code>final Boolean contains(Object element)</code> | Returns true if element is contained by the vector, returns false otherwise. |
| <code>final void copyInto(object array[])</code> | The elements contained in the invoking vector are copied into the array specified by array. |
| <code>final Object elementAt(int index)</code> | Returns the element at the location specified by index. |
| <code>final Enumeration elements()</code> | Returns an enumeration of the elements in the vector. |
| <code>final void ensureCapacity(int size)</code> | Sets the minimum capacity of the vector to size. |
| <code>final Object firstElement()</code> | Returns the first element in the vector. |
| <code>final int indexOf(Object element)</code> | Returns the index of the occurrence of element. If the object is not in the vector, -1 is returned. |
| <code>final int indexOf(Object element, int start)</code> | Returns the index of the first occurrence of element at or after start. If the object is not in that portion of the vector, -1 is returned. |
| <code>final void insertElementAt(Object element, int index)</code> | Adds element to the vector at the location specified by index. |
| <code>final Boolean isEmpty()</code> | Returns true if the vector is empty and returns false if it contains one or more elements. |
| <code>final Object lastElement()</code> | Returns the last element in the vector. |
| <code>final int lastIndexOf(Object element)</code> | Returns the index of the last occurrence of element. If the object is not in the vector, -1 is returned. |
| <code>final int lastIndexOf(Object element, int start)</code> | Returns the index of the last occurrence of element before start. If the object is not in that portion of the vector, -1 is returned. |
| <code>final void removeAllElements()</code> | Empties the vector. After this method executes, the size of the vector is zero. |
| <code>final Boolean removeElement(Object element)</code> | Removes element from the vector. If more than one instance of the specified object exist in the vector it is the first one that is removed. Returns true if successful and false if the object is not found. |
| <code>final void removeElementAt(int index)</code> | Removes the element at the location specified by index. |
| <code>final void setElementAt(Object element, int index)</code> | The location specified by index is assigned element. |
| <code>final void setSize(int size)</code> | Sets the number of elements in the vectors to size. If the new size is less than the old, elements are lost. If the new size is larger than the old, null elements are added. |
| <code>final int size()</code> | Returns the number of elements currently in the vector. |
| <code>String toString()</code> | Returns the string equivalent of the vector. |
| <code>final void trimToSize()</code> | Sets the vector's capacity equal to the number of elements that it currently holds. |

Table 18.11 Methods of Vector class

```
/*PROG 18.23 DEMO OF VECTOR CLASS VER 1 */
```

```
import java.util.*;
class JPS23{
    static void show(String s){
        System.out.println(s);
    }
    public static void main(String args[])
    {
        Vector V = new Vector();
        show("\nVector size := " + V.size());
        show("Vector capacity := " + V.capacity());

        V.addElement(new Integer(12));
        V.addElement("Vector_Demo");
        V.addElement(new Double(14.54));
        V.addElement(new Boolean(true));
        V.addElement("Element");
        V.addElement(new Character('V'));

        show("\nFirst Element := " + V.firstElement());
        show("Last Element := " + V.lastElement());

        show("Vector size now := " + V.size());
        show("Vector Capacity now := " + V.capacity());

        Enumeration E;
        show("\nElements of Vector V");
        for (E = V.elements(); E.hasMoreElements(); )
            show(E.nextElement() + " ");
        show(" ");
    }
}
```

OUTPUT:

```
Vector size          := 0
Vector capacity     := 10
First Element       := 12
Last Element        := V
Vector size now     := 6
Vector Capacity now := 10
Elements of Vector V
12
Vector_Demo
14.54
true
Element
V
```

Explanation: This program creates an instance V of the Vector class. The initial size and capacity of the V is **0** and **10**, respectively. Elements are added to the V. using addElement method. The method firstElement and lastElement returns the first and last element of the vector V.

For listing all the elements Enumeration is used. A reference E of this interface is created and initialized to V.elements() that returns a reference to Enumeration type. Till the vector V has got more elements (which is checked using method hasMoreElements) these elements are displayed using nextElement() method. This is similar to iterating all the elements of a collection using iterator interface and iterator method.

```
/*PROG 18.24 DEMO OF VECTOR CLASS VER 2 */

import java.util.*;
class JPS24
{
    static void show(String s)
    {
        System.out.print(s);
    }
    public static void main(String args[])
    {
        Vector V = new Vector(10);
        for(int i =1; i<=5;i++)
        V.addElement(new Integer(i));
        V.addElement("Joy");
        V.addElement("Fun");
        V.addElement("cool");
        V.addElement("Laugh");
        show("\nVector size: "+V.size()+"\n");
        show("\nVector capacity: "+V.capacity()+"\n");
        Object obj[] = new Object[V.size()];
        V.copyInto(obj);
        show("\nContents of array\n");
        for(int i = 0;i<obj.length;i++)
        show(obj[i]+ " ");
        show("\nV contains \"cool\":"
            +V.contains("cool")+"\n");
        show("\nElements at location 6 is "
            +V.elementAt(6)+"\n");

        V.removeElementAt(2);
        V.remove(new Integer(1));
        V.remove("cool");
        V.insertElementAt("Added",2);

        obj = new Object[V.size()];
        V.copyInto(obj);
        show("\nContents of array now\n");
        for(int i=0;i<obj.length;i++)
        show(obj[i]+ " ");
    }
}
```

```

        show("\n");

        V.removeAllElements();
        show("\nAfter removing all elements \n");
        show("V.isEmpty() returns :" + V.isEmpty() + "\n");
    }

}

OUTPUT:

Vector size: 9
Vector capacity: 10
Contents of array
1 2 3 4 5 Joy Fun cool Laugh
V contains "cool": true

Elements at location 6 is Fun

Contents of array now
2 4 Added 5 Joy Fun Laugh

After removing all elements
V.isEmpty() returns :true

```

Explanation: This program makes use of various methods of `Vector` class. For copying a vector into an Object array `copyInto` method can be used. In this program, first an Object array of vector size is created, then `copyInto` method is used for copying all elements into Object array `obj`. The `contains` method checks whether specified element is in the vector or not. The `elementAt` method returns the element at the specified location. The `remove` method removes the specified object from the vector and `removeElementAt` removes object at the specified location. The `insertElementAt` inserts the element at the specified location. The `removeAllElements` is equivalent to `clear` method which removes all the elements from the vector. The method `isEmpty` returns true if the vector is empty.

3. The Stack class

The stack class represents a last-in-first-out (LIFO) stack of objects. It extends Vector class with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top. They are shown in the Table 18.12.

| Method Signature | Description |
|--|---|
| <code>boolean empty()</code> | Returns true if the stack is empty, and returns false if the stack contains elements |
| <code>Object peek()</code> | Returns the element on the top of the stack, but does not remove it. |
| <code>Object pop()</code> | Returns the element on the top of stack, removing it in the process. |
| <code>Object push(Object element)</code> | Pushes element onto the stack, element is also returned. |
| <code>int search(Object element)</code> | Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned. |

Table 18.12 Methods of Stack class

When a stack is first created, it contains no items.

An `EmptyStackException` is thrown if `pop()` is called when the invoking stack is empty. Because the `Stack` class extends the `Vector` class, a `Stack` can still be accessed or modified with the inherited `Vector` methods.

```
/*PROG 18.25 DEMO OF STACK CLASS VER 1 */

import java.util.*;
class JPS25
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        Stack S = new Stack();
        S.push(new Integer(5));
        S.push(new Integer(7));
        S.push(new Integer(3));

        show("\nStack contents");
        show(S + " ");
        show("\nFirst item popped: " + S.pop());

        S.push("Two");
        S.push("One");
        S.push("Zero");
        show("\nAfter adding some elements");
        show("Stack contents\n");
        show(S + " ");

        show("\nFirst item popped:" + S.pop());
        show("\nFirst item peeked:" + S.peek());

        show("\nStack contents");
        show(S + " ");
        int pos = S.search(new Integer(6));
        if (pos != -1)
            show("\nFrom top offset of 6 is:" + pos);
        else
            show("\nElement is not in stack");
    }
}

OUTPUT:

Stack contents
[5, 7, 3]
```

```

First item popped: 3
After adding some elements
Stack contents
[5, 7, Two, One, Zero]
First item popped: Zero
First item peeked: One
Stack contents
[5, 7, Two, One]
Element is not in stack

```

Explanation: In this program, Stack instance S is created. In the beginning, some Integer objects are pushed and then the contents of the Stack instance S displayed. The top element is then popped from the stack. Further elements are added to the Stack S and then first item is popped. Next first item is only peeked and displayed. The Integer object 6 is searched in the Stack S using search method. The method returns the offset of the element from the top in case the element exists in the Stack S else, it returns -1.

4. The Dictionary class

The dictionary class is the abstract parent of any class, such as HashTable, which maps keys to values. Every key and every value is an object. In any one Dictionary object, every key is associated with at most one value. Given a Dictionary and a key, the associated element can be looked up. Thus, like a map, a dictionary can be considered as a list of key-value pairs. Any non-null object can be used as a key and as a value. The Dictionary class is completely full of abstract methods. In other words, it should have been an interface. It forms the basis for a key-value pair collections in the historical collection classes, with its replacement being Map in the new framework. The class is now obsolete so it will not be discussed in detail.

5. The HashTable class

This class implements a hashTable, which maps keys to values. Any non-null object can be used as a key or as a value. The Hashtable implementation is a generic dictionary that permits storing any object as its key or value (besides null). With the Java 2 SDK, version 1.2, the class has been reworked into the Collections Framework to implement the Map interface. So the original HashTable methods or the newer Map method can be used. For a synchronized Map, using Hashtable is slightly faster than using a synchronized HashMap.

In Hashtable, hashing is applied onto the key and a hash code is returned. The hash code is used as the index at which the value is stored within the table. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method. Fortunately, many of Java's built-in classes already implement the hashCode () method. For example, the most common type of Hashtable uses a String object as the key. String implements both hashCode () and equals ().

The Hashtable class defines the following constructors:

1. **public Hashtable()**

This form of constructor creates a new, empty hashtable with a default initial capacity(11) and load factor, which is 0.75.

2. **public Hashtable(int intialCapacity)**

This form of constructor creates a new, empty hashtable with the specified initial capacity and default load factor, which is 0.75.

3. **public Hashtable(int initialCapacity, float loadFactor)**

This form of constructor creates a new, empty hashtable with the specified initial capacity and the specified load factor.

4. **public Hashtable(Map t)**

This form of constructor creates a new hashtable with the same mappings as the given Map. The hashtable is created with an initial capacity sufficient to hold the mappings in the given Map and a default load factor, which is 0.75.

The historical methods of the HashTable class are shown in the Table 18.13.

| Method Signature | Description |
|---|---|
| <code>void clear()</code> | Resets and empties the hash table. |
| <code>Object clone()</code> | Returns a duplicate of the invoking object. |
| <code>boolean contains(Object value)</code> | Returns true if some value equal to value exists within the hash table.
Returns false if the value is not found. |
| <code>boolean containsKey(Object key)</code> | Returns true if some key equal to key exists within the hash table.
Returns false if the key is not found. |
| <code>boolean containsValue(Object value)</code> | Returns true if some value equal to value exists within the hashtable.
Returns false if the value is not found. (A non-Map method added by Java 2, for consistency) |
| <code>Enumeration elements()</code> | Returns an enumeration of the values contained in the hash table. |
| <code>Object get(Object key)</code> | Returns the object that contains the value associated with the key. If key is not in the hash table, a null object is returned. |
| <code>boolean isEmpty()</code> | Returns true if the hash table is empty; returns false if it contains at least one key. |
| <code>Enumeration keys()</code> | Returns an enumeration of the keys contained in the hash table. |
| <code>Object put(Object key, Object value)</code> | Inserts a key, and a value into the hash table. Returns null if key is not already in the hash table; returns the previous value associated with key if key is already in the hash table. |
| <code>void rehash()</code> | Increases the size of the hash table and rehashes all of its keys. |
| <code>Object remove(Object key)</code> | Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned. |
| <code>int size()</code> | Returns the number of entries in the hash table. |
| <code>String toString()</code> | Returns the string equivalent of a hash table. |

Table 18.13 Methods of Hashtable class

```
/*PROG 18.26 DEMO OF HASHTABLE CLASS VER 1 */
```

```
import java.util.*;
class JPS26
{
    public static void main(String args[])
    {
        Hashtable HT = new Hashtable();
        HT.put(new Integer(101), "Hari");
        HT.put(new Integer(104), "Man");
        HT.put(new Integer(310), "Hemangi");
        HT.put(new Integer(140), "Ruchika");
```

```

HT.put(new Integer(320), "Veeneta");

Enumeration E;
System.out.println("\nKEYS\tVALUES");
for (E = HT.keys(); E.hasMoreElements(); )
{
    Object obj = E.nextElement();
    System.out.println(obj + "\t" + HT.get(obj));
}
}

OUTPUT:
KEYS      VALUES
140      Ruchika
104      Man
310      Hemangi
101      Hari
320      Veeneta

```

Explanation: An Enumeration for the keys is obtained using keys method of Hashtable. For each key the corresponding value is obtained using get method. Rest is simple to understand.

```

/*PROG 18.27 DEMO OF HASHTABLE CLASS VER 2 */

import java.util.*;
class JPS27
{
    static void show(String s){
        System.out.println(s);
    }
    public static void main(String args[])
    {
        Hashtable HT = new Hashtable();
        HT.put(new Integer(101), "Hari");
        HT.put(new Integer(104), "Ranjana");
        HT.put(new Integer(210), "Anjana");
        HT.put(new Integer(140), "Vijay");
        HT.put(new Integer(310), "Madhuri");

        Iterator itr;
        System.out.println("\nKEYS\tVALUES");
        Set S = HT.entrySet();
        for (itr = S.iterator(); itr.hasNext(); )
        {
            Map.Entry me = (Map.Entry)itr.next();
            show(me.getKey() + "\t" + me.getValue());
        }
    }
}
```

OUTPUT:

| KEYS | VALUES |
|------|---------|
| 140 | Vijay |
| 104 | Ranjana |
| 310 | Madhuri |
| 101 | Hari |
| 210 | Anjana |

Explanation: As the class implements the Map interface the set-view of the Hashtable can be obtained using the entrySet method of the Map interface. Once obtained, then using iterator and MapEntry interface all the keys and values of the Hashtable can be displayed. One more method using the keySet method and iterator is given as:

```
Iterator itr;
System.out.println("\nKEYS\tVALUES");
Set S = HT.entrySet();
for (itr = S.iterator(); itr.hasNext(); )
{
    Object obj = itr.next();
    System.out.println(obj+ "\t"+HT.get(obj));
}
```

Here using the keySet method a set of keys is obtained, and using Iterator and get method all the elements are displayed.

6. The Properties class

The Properties implementation is a specialized Hashtable for working with text strings. While values retrieved from a Hashtable have to be cast, the Properties class allows to get text values without casting. In Properties the key is a String and the value is also a String. The class also supports loading and saving property settings from an input stream or to an output stream.

The most commonly used set of properties is the system properties list, retrieved by

```
System.getProperties()
```

The class defines the following constructors:

1. **public Properties()**

This form of constructor creates an empty property list with no default values.

2. **public Properties(Properties defaults)**

This form of constructor creates an empty property list with the specified defaults.

Apart from the above two constructors, the class defines one protected member defaults whose declaration is given as:

```
protected Properties defaults
```

It is a property list that contains default values for any keys not found in this property list.

The various methods of Properties class are given in the Table 18.14.

| Method Signature | Description |
|--|--|
| String getProperty(String key) | Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list. |
| String getProperty(String key, String defaultProperty) | Returns the values associated with key. A default Property is returned if key is neither in the list nor in the default property list. |

| | |
|--|--|
| void list(PrintStream streamOut) | Sends the property list to the output stream linked to streamOut. |
| void list(PrintWriter streamOut) | Sends the property list to the output stream linked to streamOut. |
| void load(InputStream streamIn) throws IOException | Inputs a property list from the input stream linked to streamIn. |
| Enumeration propertyNames() | Returns an enumeration of the keys. This includes those keys found in the default property list, too. |
| Object setProperty(String key, String value) | Associates value with key. Returns the previous value associated with key, or returns null if no such association exists. (Added by Java 2, for consistency) |
| void store(OutputStream streamOut, String description) | After writing the string specified by description, the property list is written to the output stream linked to streamOut. (Added by Java 2). |

Table 18.14 Methods of Properties class

Following is a small example of Properties class.

```
/*PROG 18.28 DEMO OF PROPERTIES CLASS VER 1*/
import java.util.*;
class JPS28
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        Properties P = new Properties();
        P.put("LINUX", "OS1");
        P.put("UNIX", "OS2");
        P.put("Java", "Prog.Lanugage");
        P.put("DDK", "System Software");
        Iterator itr;
        System.out.println("\nKEYS\t\tVALUES");
        Set S = P.keySet();
        for (itr = S.iterator(); itr.hasNext(); )
        {
            String obj = (String)itr.next();
            System.out.println(obj+"\t\t"+P.getProperty(obj));
        }
    }
}
OUTPUT:
KEYS      VALUES
Java      Prog.Lanugage
LINUX     OS1
UNIX      OS2
DDK       System Software
```

Explanation: This program is simple to understand. In this program, keys have been used as some computer system tools and values are their brief description. Using keySet method the set-view of the keys is taken and stored in the Set instance S. Then using iterator and for loop each key in turn is extracted using next method and the corresponding value is obtained using getProperty method of Properties class.

```
/*PROG 18.29 DEMO OF PROPERTIES CLASS VER 2 */

import java.util.*;
class JPS29
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        Properties def = new Properties();
        def.put("XYZ", "don't know");
        Properties P = new Properties(def);

        P.put("LINUX", "OS");
        P.put("Java", "Prog.Language");

        show(P.getProperty("LINUX"));
        show(P.getProperty("XYZ"));

        String temp = P.getProperty("pqr", "What is this");
        show(temp);
    }
}
OUTPUT:
OS
don't know
What is this
```

Explanation: When a value is not associated with any of the key specified then a default value can be set for a Properties instance. There are two methods to do this. In the first method, default values can be stored for a Properties instance and the instance is passed as a parameter when creating new Properties instance. This is shown in the program. A Properties instance def is created and just one key 'xyz' with value 'don't know' is stored in it. This def is passed as an argument to the Properties constructor when creating Properties instance P. Now when line

```
show(P.getProperty("xyz"));
```

executes, JVM looks for a value associated with 'xyz'. As there is no key named 'xyz' in the P, default Properties instance def is searched. There key 'xyz' is found and value is returned.

The second method is to use getProperty method but with two String arguments. First argument is the key and second is its default value. In this program, the key 'pqr' is not found in P, so its default value 'What's this' is returned.

```
/*PROG 18.30 DEMO OF PROPERTIES CLASS VER 3 */
```

```
import java.util.*;
import java.io.*;
class JPS30
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        try
        {
            String name, mnum;
            Properties P = new Properties();
            FileOutputStream fout = null;
            Scanner sc = new Scanner(System.in);
            fout = new FileOutputStream("mbook.txt");
            while(true)
            {
                show("Enter the name");
                name = sc.next();
                if (name.equals("exit") == true)
                    break;
                show("Enter the mobile number");
                mnum = sc.next();
                P.put(name, mnum);
            }
            P.store(fout, "Mobile Book");
            fout.close();
        }
        catch (Exception e)
        {
            System.out.println("Error in file handling");
        }
    }
}
```

OUTPUT:

```
Enter the name
H.M.Pandey
Enter the mobile number
9923598540
Enter the name
Lala
Enter the mobile number
9423347619
Enter the name
exit
```

Explanation: The store method of the Properties class can be used to store the contents of the Properties instance to a file. The program is having a Properties instance P in which name and mobile number of few persons are stored. Both the name and mobile number are taken from console. After storing few entries into the Properties instance all entries of P are written to the file 'mbook.txt' which is opened earlier using FileOutputStream. The second argument in the store method is a String object. It is like giving description of the file.

The store 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 method. If the second argument is not null, an ASCII# character, the string (second argument), and a line separator are first written to the output stream. Next, a comment line is always written, consisting of an ASCII# character, the current date and time and a line separator. Then every entry in this Properties table is written out, one per line. For each entry the key string is written, then an ASCII=, then the associated element string.

```
#Mobile Book
#Sat Jan 10 17:02:24 PST 2009
H.M.Pandey=9923598540
Lala=9423347619
```

```
/*PROG 18.31 DEMO OF PROPERTIES CLASS VER 4 */

import java.io.*;
import java.util.*;
class JPS31
{
    public static void main(String args[])
    {
        try
        {
            Properties P = new Properties();
            FileInputStream fin = null;
            fin = new FileInputStream("mbook.txt");
            if (fin != null)
            {
                P.load(fin);
                Iterator itr;
                System.out.println("\nContact details are");
                System.out.println("\nNAME\t\tMOBILE NO.");
                Set S = P.keySet();
                for (itr = S.iterator(); itr.hasNext(); )
                {
                    String obj = (String)itr.next();
                    System.out.println(obj+"\t\t"+P.
                        getProperty(obj));
                }
            }
            fin.close();
        }
        catch (Exception e)
        {
            System.out.println("Error in file handling ");
        }
    }
}
```

```

        }
    }
}

OUTPUT:

Contact details are
NAME      MOBILE NO.
Hari      9923598540
Man       9745659019
Lala      9423347619
Ritu      9923781210

```

Explanation: In this program, the load method of Properties class is used for loading of entries from the file into an instance of Properties class. Here the file ‘mbook.txt’ is first opened using FileInputStream class. If the file successfully opened, the entries are to be loaded into P using load method. Once loaded, rest of the code simply displays the name and mobile numbers of the person. This has been explained earlier.

18.11 ALGORITHM SUPPORT

The Collection class available as part of the Collection Framework, provides support for various algorithms with the collection classes, both new and old. This class exclusively consists of static methods that operate on or returns collections. It contains polymorphic algorithms that operate on collections, ‘wrappers’, which returns a new collection backed by a specified collection and a few others. The methods of this class all throws a NullPointerException if the collection or class objects provided to them are null. One more execution, UnsupportedOperationException, occurs when an attempt is made to modify an unmodified collection.

The various methods of this class are shown in the Table 18.15.

| Method Signature | Description |
|---|---|
| static int binarySearch
(List list, Object value,
Comparator c) | Searches for values in list ordered according to c. Returns the position of values in list, or 'l' if value is not found. |
| static int binarySearch(List
list, Object value) | Searches for value in list. The list must be sorted. Returns the position of value in list, or l if value is not found. |
| Static void copy(List list1, List
list2) | Copies the elements of list1 to list2. |
| Static Enumeration
enumeration(Collection c) | Returns an enumeration over c. |
| Static void fill(List list,
Object obj) | Assigns obj to each element of list. |
| Static int frequency
(Collection c, Object o) | Returns the number of elements in the specified collection equal to the specified object. |
| Static Object max(Collection c,
Comparator comp) | Returns the maximum element in c as determined by comp. |

(Continued)

| | |
|--|--|
| Static object max(Collection c) | Returns the maximum element in c as determined by natural ordering. The collection need not be sorted. |
| Static Object min(Collection c, Comparator comp) | Returns the minimum element in c as determined by comp. The collection need not be sorted. |
| Static Object min(Collection c) | Returns the minimum element in c as determined by natural ordering. |
| Static List nCopies(int num, Object obj) | Returns num copies of obj contained in an immutable list. num that must be greater than or equal to zero. |
| Static void reverse(List list) | Reverses the sequence in list. |
| Static Comparator reverseOrder() | Returns a reverse comparator (a comparator that reverses the outcomes of a comparison between two elements). |
| Static void shuffle(List list, Random r) | Shuffles (i.e., randomizes) the elements in list by using r as a source of random numbers. |
| Static void shuffle(List list) | Shuffles (i.e., randomizes) the elements in list. |
| Static Set singleton(Object obj) | Returns obj as an immutable set. This is an easy way to convert a single object into a set. |
| Static void sort(List list, Comparator comp) | Sorts the elements of list as determined by comp. |
| Static void sort(List list) | Sorts the elements of list as determined by their natural ordering. |
| Static Collection synchronizedCollection(Collection c) | Returns a thread-safe collection backed by c. |
| Static List synchronizedList(List list) | Returns a thread-safe list backed by list. |
| Static Map synchronizedMap(Map m) | Returns a thread-safe map backed by m. |
| Static Set synchronizedSet(Set s) | Returns a thread-safe set backed by s. |
| static SortedMapsynchronizedSortedMap(SortedMap m) | Returns a thread-safe sorted set backed by sm. |
| static SortedSetsynchronizedSortedSet(SortedSet ss) | Returns a thread-safe set backed by ss. |
| staticCollectionunmodifiableCollection(Collection c) | Returns an unmodifiable collection backed by c. |
| Static List unmodifiableList(list list) | Returns an unmodifiable list backed by list. |
| Static Map unmodifiableMap (Map m) | Returns an unmodifiable map backed by m. |
| Static Set unmodifiableSet(Set s) | Returns an unmodifiable set backed by s |
| Static sortedMapunmodifiableSortedMap(SortedMap sm) | Returns an unmodifiable sorted map backed by sm. |
| Static SortedSetunmodifiableSortedSet(SortedSet ss) | Returns an unmodifiable sorted set backed by ss. |

Table 18.15 Algorithms supported by Collection framework

All the methods which start with `synchronized` are thread-safe methods. Recall the Collections framework does not provide any synchronized collections. Using the `synchronized` method the synchronization can be achieved among multiple threads accessing the collections. It is imperative that the user manually synchronize on the returned collection when iterating over it, that is, while iterating a collection it must be written inside a synchronized block.

All the methods which start with `unmodifiable` return immutable collections, that is, they cannot be modified once created.

Apart from the above methods, the class `Collections` provides three static fields:

- (i) `public static final List EMPTY_LIST`
The `EMPTY_LIST` represents the empty list (immutable).
- (ii) `public static final Set EMPTY_SET`
The `EMPTY_SET` represents the empty set (immutable).
- (iii) `public static final Map EMPTY_MAP`
The `EMPTY_MAP` represents the empty map (immutable).

The following programs make use of these methods:

```
/*PROG 18.32 DEMO OF FEW ALGORITHMS OF COLLECTION VER 1 */
```

```
import java.util.*;
import java.io.*;
class JPS32
{
    static void show(String s)
    {
        System.out.print(s);
    }
    public static void main(String args[])
    {
        ArrayList AL = new ArrayList();
        AL.add(new Integer(20));
        AL.add(new Integer(35));
        AL.add(new Integer(5));
        AL.add(new Integer(135));
        AL.add(new Integer(467));
        AL.add(new Integer(8));

        show("\nMin element of list:"
             + Collections.min(AL) + "\n");
        show("\nMax element of list:"
             + Collections.max(AL) + "\n");

        Collections.sort(AL);

        Iterator itr;

        show("\nList in sorted order\n\n");
        for (itr = AL.iterator(); itr.hasNext(); )
            show(itr.next() + " ");
        show("\n");
```

```

        Collections.reverse(AL);
        show("\nList in reverse order\n\n");
        for (itr = AL.iterator(); itr.hasNext(); )
            show(itr.next() + " ");
        show("\n");
        Collections.shuffle(AL);
        show("\nList after shuffling\n\n");
        for (itr = AL.iterator(); itr.hasNext(); )
            show(itr.next() + " ");
        show("\n");

        Collections.fill(AL, "Filled");
        show("\nList after fill \n\n");
        for (itr = AL.iterator(); itr.hasNext(); )
            show(itr.next() + " ");
        show("\n");
    }
}

OUTPUT:

Min element of list: 5
Max element of list: 467
List in sorted order
5 8 20 35 135 467
List in reverse order
467 135 35 20 8 5
List after shuffling
8 35 467 20 135 5
List after fill
Filled Filled Filled Filled Filled Filled

```

Explanation: This program is simple to understand. Simply observe how various Collections methods are used.

```
/*PROG 18.33 DEMO OF FEW ALGORITHMS OF COLLECTION VER 2 */
```

```

import java.io.*;
import java.util.*;
class JPS33
{
    static void show(String s)
    {
        System.out.print(s);
    }
    public static void main(String args[])
    {
        List LL =Collections.nCopies(5,new Integer(10));
        //LL.set(0, new Integer(20));line generates error;
        List UML = Collections.unmodifiableList(LL);
    }
}

```

```

//LL.set(2, "Changed");line generates error;
Comparator cm = Collections.reverseOrder();
LinkedList LL1 = new LinkedList();

LL1.add(0, "one");
LL1.add(1, "two");
LL1.add(2, "three");
LL1.add(3, "four");

Collections.sort(LL1, cm);
Iterator itr;

show("\nList in reverse sorted order using
comparator\n");

for (itr = LL1.iterator(); itr.hasNext(); )
show(itr.next() + " ");
show("\n");

Collections.fill(LL1, new Integer(5));
int freq = Collections.frequency(LL1, new
Integer(5));
show("\nFrequency of Integer object 5 is:"
+freq"\n");
}

}

OUTPUT:

List in reverse sorted order using comparator
two three one four
Frequency of Integer object 5 is: 4

```

Explanation: The method `nCopies` returns an immutable list after filling the list with 5 integer objects of value 10 each. Note the list cannot be modified so the line in comment generates run time error. Similarly, the method `unmodifiableList` creates an immutable list that cannot be modified.

The method `reverseOrder` returns a comparator that is sorted in `cm`. This `cm` is used for sorting the contents of list `LL1` in reverse order.

```
/*PROG 18.34 DEMO OF FEW CONSTANTS OF COLLECTION VER 3 */
```

```

import java.util.*;
import java.io.*;
class JPS34
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {

```

```

List LL = Collections.EMPTY_LIST;
//LL.add(0,"hello");line generates error
show("\nLL.isEmpty():" + LL.isEmpty());

Set S = Collections.EMPTY_SET;
//S.add("hello");line generates error
show("\nS.isEmpty():" + S.isEmpty());
Map M = Collections.EMPTY_MAP;
//M.put("hello","bye");line generates error
show("\nM.isEmpty():" + M.isEmpty());
}

OUTPUT:
LL.isEmpty():true
S.isEmpty():true
M.isEmpty():true

```

Explanation: The static constants EMPTY_LIST, EMPTY_SET and EMPTY_MAP represent empty immutable list, set and map, respectively. Rest is simple to understand.

18.12 PONDERABLE POINTS

1. The Collection Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection. The framework provides a convenient API to many of the abstract data types used in data structure curriculum: maps, sets, lists, trees, arrays, hashtables and so on.
2. The Collections Framework is made up of a set of interfaces for working with groups of objects. The different interfaces describe different types of groups.
3. The Collection interface is a group of objects with duplicates allowed.
4. The Set interface extends Collection but forbids duplicates.
5. The List interface extends Collection, allows duplicates, and introduces positional indexing.
6. The Map interface extends neither Set nor Collection.
7. There are two general-purpose List implementations in the Collection Framework: ArrayList and LinkedList.
8. Iterating means accessing each element of the collection one by one, that is, using an iterator each of the elements of the collection can be cycled through. The iterator() method of the Collection interface returns an Iterator. With the Iterator interface methods, a collection can be traversed from start to finish.
9. The Collections Framework provides two general-purpose implementations of the Set interface: HashSet and TreeSet.
10. A Map is used to store key-value pairs, with the values retrieved using the key. For each given key there will be a corresponding value. In essence, each element of the map is stored as a (key, value) pair. Given a key, a value can be retrieved. This is the most powerful feature of map. Both key and value are objects. The keys must be unique but values may be duplicates.
11. The Comparator interface is a type of comparison function, which imposes a total ordering on some collection of objects.

12. The historical collection classes are those classes which were supplied with the earlier version of Java. They are Vector, Stack, etc.
13. The Vector class implements a growable array of objects, but can store heterogeneous data elements. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
14. The Enumeration interface allows one to iterate through all the elements of a collection.
15. The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. They are push, pop empty, search and peek.
16. The class Hashtable implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value. The Hashtable implementation is a generic dictionary that permits storing any object as its key or value (besides null).
17. The Properties implementation is a specialized Hashtable for working with text strings. In Properties the key is a String and the value is also a String. The class also supports loading and saving property settings from an input stream or to an output stream.
18. The Collection class available as part of the Collections Framework provides support for various algorithms with the collection classes, both new and old. This class consists, exclusively, of static methods that operate on or returns collections.

REVIEW QUESTIONS

1. What is Java Collection Framework?
2. What are the four main types defined in the Java Collection Framework?
3. What is legacy class?
4. Which collection classes are implemented with an indexed structure (i.e., a simple array)?
5. Which collection classes are implemented with a linked structure?
6. What is an ArrayList object?
7. What is a LinkedList object?
8. What is the Collection interface?
9. What set-theoretic methods are supported by the Java Collection Framework?
10. What is an iterator?


```
int chars(List<String>strings)
//returns the total number of
characters in all the strings
```
11. How are iterators similar to array indexes?
12. How are iterators different from array indexes?
13. How can the Array class be used to initialize a collection?
14. How can the Collections class be used to initialize a collection?
15. What is a generic class?
16. What is a generic method?
17. What is generic type parameter?
18. What is a constrained generic type parameter?
19. What is generic type argument?
20. What is auto boxing?
21. Write and test this method:
22. Write and test this method using an iterator:


```
void print(Collection C)
//prints all the elements of the
collection c
```
23. Write and test this generic method:


```
<E>int frequency(Collection<E>c, E
e) {
//returns the number of occur-
rences of e in c
```

 - (a) Using an iterator.
 - (b) Using an enhanced for loop
24. Write and test this generic method:


```
<E>E getElementAt(List<E>list,
int index) {
//returns the list element at the
specified index
```

 - (a) Using an iterator.
 - (b) Using an enhanced for loop.

Multiple Choice Questions

1. A hash table can store a maximum of 10 records. Currently there are records in location 1, 3, 4, 7, 8, 9, and 10. The probability of a new record going into location 2, with a hash function resolving collisions by linear probing is:
 - (a) 0.6
 - (c) 0.2
 - (b) 0.1
 - (d) 0.5
2. The HashMap class automatically resizes its hash table when the load factor reaches a specific threshold. This threshold value can be set by:
 - (a) public HashMap (int initialCapacity, float loadFactor)
 - (b) public HashMap (int initial size, float loadFactor)
 - (c) public HashMap (int initial size, float Capacity)
 - (d) None of the above.
3. Which of the following statements is true:
 - I. The size of the hash table is the actual number of elements in the table.
 - II. The capacity of the table is the number of components that hash table has.
 - III. The HashMap class automatically resizes its hash table when the load factor reaches a specific threshold.
 - IV. The ratio of the size of the hash table to the capacity of the table is called as load factor.
 - (a) I and II
 - (c) III and IV
 - (b) II and IV
 - (d) All are correct.
4. Java defines Hashtable class in _____ package.
 - (a) java.awt
 - (c) java.io
 - (b) java.util
 - (d) java.lang
5. A hash function randomly distributes records one by one in a space that can hold x number of records. The probability that the m th record is the first record to result in collision is:
 - (a) $(x-1)(x-2) \dots (x-(m-2))(m-1)/x^{m-1}$
 - (b) $(x-1)(x-2) \dots (x-(m-1))(m-1)/x^{m-1}$
 - (c) $(x-1)(x-2) \dots (x-(m-2))(m-1)/x^m$
 - (d) $(x-1)(x-2) \dots (x-(m-1))(m-1)/x^m$
6. If the hashing function is the remainder on division, the clustering is more likely to occur if the storage space is divided into 40 sectors rather than 41. This conclusion is:
 - (a) More likely to be false
 - (b) More likely to be true
 - (c) Is always false
 - (d) None of the above
7. A hash function f defined as $f(\text{key}) = \text{key} \bmod 7$, with linear probing, is used to insert the keys 37, 72, 48, 98, 11, 56, into a table indexed from 0 to 6. 11 will be stored at the location:
 - (a) 3
 - (c) 5
 - (b) 4
 - (d) 6
8. Consider a hashing function that resolves collision by quadratic probing. Assume the address space is indexed from 1 to 8. If a collision occurs at position 4, which of the following locations will never be adopted?
 - (a) 4
 - (c) 8
 - (b) 5
 - (d) 2
9. Java defines four implementations of its Map interface: the AbstractMap class, the HashMap class, the TreeMap class and the WeakHashMap class. Here, AbstractMap class works as:
 - (a) Inherited class for all the other three classes
 - (b) Base class for all the other three classes
 - (c) Inner class for all the other three classes
 - (d) None of the above
10. Trace the following code


```

1. import java.util.*;
2. class JPS
3. {
4.     static void show(String s)
5.     {
6.         System.out.println(s);
7.     }
8.     public static void main(String args[])
9.     {
10.        HashMap HM=new HashMap();
11.        HM.put("PEARSON", new Integer(201));
12.        HM.put("Noida", new Integer(202));
      
```

```
13. Set s=HM.entrySet();  
14. show("Map elements\n");  
15. Iterator itr=s.iterator();  
16. show("Key\t Value");  
17. while(itr.hasNext())  
18. {  
19. Map.Entry me=(Map.Entry) itr.next();  
20. show(me.getKey()+"\t"+me.getValue());  
21. }  
22. }  
23. }
```

(a) Map elements

| Key | Value |
|---------|-------|
| PEARSON | 201 |
| Noida | 202 |

(b) Map elements

| Key | Value |
|---------|-------|
| Noida | 201 |
| PEARSON | 202 |

(c) Map elements

| Key | Value |
|---------|-------|
| PEARSON | 202 |
| Noida | 201 |

(d) None of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. a 3. d 4. b 5. a 6. b 7. c 8. d 9. b 10. a

19

Basic Utility Classes

19.1 INTRODUCTION

This chapter discusses a number of classes defined within various Java packages. These are the classes that are used frequently and are of importance. A number of methods of the classes are discussed and examples provided wherever necessary.

19.2 THE STRINGTOKENIZER CLASS

This class is defined within `java.util` package. The `StringTokenizer` class allows an application to break a string into tokens. The `StringTokenizer` methods do not distinguish among identifiers, numbers and quoted strings, neither do they recognize and skip comments. Tokens are basically division of text into discrete parts as given by the user. `StringTokenizer` implements the `Enumeration` interface. Therefore, given an input string, the individual tokens contained in it can be enumerated using `StringTokenizer`.

To use `StringTokenizer`, an input string has to be specified and a string that contains delimiter(s). The delimiter(s) are characters that separate tokens. Each character in the delimiter(s) string is considered a valid delimiter, for example, `“.”` sets the delimiter to colon. The default set of delimiter(s) consists of the whitespace characters: space, tab, newline and carriage returns.

The various constructors of `StringTokenizer` class are as shown below:

1. **`public StringTokenizer(String str);`**

This form of constructor creates a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is `“\t\n\r\f”`: the space character, the tab character, the newline character, the carriage-return character, and the form-feed character. Delimiter characters themselves will not be treated as tokens.

2. **`public StringTokenizer(String str, String delim);`**

This form of constructor creates a string tokenizer for the specified string. The characters in the `delim` argument are the delimiters for separating tokens. Delimiter characters themselves will not be treated as tokens. This form of constructor is most frequently used.

3. **`public StringTokenizer(String str, String delim, boolean ret_Delims);`**

This form of constructor is similar to the second form of constructor with the difference that it takes one more parameter of `boolean` type. If the third parameter is true, delimiters are returned as tokens else, they are not.

The two methods `nextToken` and `hasMoreTokens` are used for extracting consecutive tokens from a string which is StringTokenized using `StringTokenizer` class. The method `nextToken` returns the next token in the string as `String` object. The method `hasMoreTokens` returns true if any more token is there in the string. For example, consider the following code:

```
/*PROG 19.1 DEMO OF STRING TOKENIZER VER 1 */

import java.util.StringTokenizer;
class JPS1
{
    public static void main(String args[])
    {
        StringTokenizer st=new StringTokenizer("This is a
            demo");
        while (st.hasMoreTokens())
        {
            System.out.println(st.nextToken());
        }
    }
}

OUTPUT:
This
is
a
demo
```

Explanation: The code uses the first form of the StringTokenizer class so the default delimiter in the above code is space character. The first token returned is ‘this’ which is printed. After that, various tokens return as ‘are’, ‘a’ and ‘demo’. For extracting all the tokens hasToken() method is used. For checking whether any more token is remaining hasMoreTokens() method is used as condition in the while loop.

```
/*PROG 19.2 DEMO OF STRING TOKENIZER VER 2*/

import java.util.StringTokenizer;
class JPS2
{
    static String minfo = "Company:Sony Erricson;" +
    "Modal:z550i;" + "Price:7600;";
    public static void main(String args[])
    {
        StringTokenizer st = new StringTokenizer(minfo,
            ":");

        while (st.hasMoreTokens())
        {
            String key = st.nextToken();
            String val = st.nextToken();
            System.out.println(key + "\t" + val);
        }
    }
}

OUTPUT:
Company Sony Erricson
Modal z550i
Price 7600
```

Explanation: A static String object minfo is taken which contains information about mobile. The ‘:’ and ‘;’ are intentionally put into the string so that they can be used as delimiter. The first token extracted from string is ‘Company’ which is stored into the String variable key due to delimiter “:”. The second token extracted is ‘Sony Erricson’ due to delimiter “;”. Both are displayed so the first line of output is obtained. On the similar basis, the remaining output is obtained. Note the above code can be written with the use of single delimiter, but the string has to be modified. See the next program.

```
/*PROG 19.3 DEMO OF STRING TOKENIZER VER 3 */

import java.util.StringTokenizer;
class JPS3
{
    static String minfo = "Company:Sony Erricson:" +
                        "Modal:z550i;" + "Price:8000";
    public static void main(String args[])
    {
        StringTokenizer st = new
                            StringTokenizer(minfo,":");
        while (st.hasMoreTokens())
        {
            String key = st.nextToken();
            String val = st.nextToken();
            System.out.println(key + "\t" + val);
        }
    }
}

OUTPUT:
Company Sony Erricson
Modal      z550i
Price      8000
```

Explanation: Read the explanation of previous program.

19.3 THE DATE CLASS

This class is defined within `java.util` package. The Date class encapsulates the current date and time. Prior to JDK 1.1, the class Date had two additional functions. It allowed the interpretation of dates as year, month, hour, minute and second values. It also allowed the formatting and parsing of date strings. Unfortunately, the API for these functions was not amenable to internationalization. As of JDK 1.1, the Calendar class should be used to convert between dates and time fields and the DateFormat class should be used to format and parse date strings. The corresponding methods in Date are deprecated.

The Date class defines two constructors:

1. **public Date()**

This constructor form allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond. In short, this constructor returns the current date and time.

2. **public Date(long date)**

This constructor form allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as ‘the epoch’, namely January 1, 1970, 00:00:00 GMT.

Apart from the above constructors, the various frequently used methods defined by the Date class are shown in the Table 19.1.

| Method Signature | Description |
|-----------------------------|---|
| boolean after(Date date) | Returns true if the invoking Date object contains a date that is later than the one specified by date. Otherwise, it returns false. |
| boolean before(Date date) | Returns true if the invoking Date object contains a date that is earlier than the one specified by date. Otherwise, it returns false. |
| int compareTo(Date date) | Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking objects is earlier than date. Returns a positive value if the invoking objects is later than date. |
| boolean equals(Object date) | Returns true if the invoking Date object contains the same time and date as the one specified by date. |
| long getTime() | Returns the number of milliseconds that have elapsed since January 1, 1970. |
| Void setTime(long time) | Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1970. |
| String toString() | Converts the invoking Date object into a string and returns the result. |

Table 19.1 Methods of Date class

```
/*PROG 19.4 DISPLAYING CURRENT DATE AND TIME */

import java.util.Date;
class JPS4
{
    public static void main(String args[])
    {
        Date d = new Date();
        System.out.println("\nToday is " + d);
    }
}

OUTPUT:

Today is Wed Jan 14 06:04:05 PST 2009
```

Explanation: The Date class is defined within the `java.util` package. The current date and time can be obtained by using the default constructor of this Date class as `Date d = new Date();`; the same can be printed directly using `println`.

```
/*PROG 19.5 DISPLAYING CURRENT DATE AND TIME VER 2 */

import java.util.*;
class JPS5
{
    public static void main(String args[])
    {
        Date d = new Date();
        long time = d.getTime();
        System.out.println("\n Number of seconds");
        System.out.println(" using gettime:" + time);
        d = new Date(time);
        System.out.println(" Today is " + d);
    }
}
OUTPUT:
Number of seconds
using gettime:1231942545765
Today is Wed Jan 14 06:15:45 PST 2009
```

Explanation: The method `getTime` returns the number of seconds that have elapsed since January 1, 1970. The same is passed as argument to `Date` constructor. The output of the program is same as of the previous one.

```
/*PROG 19.6 DEMO OF AFTER AND BEFORE METHOD */

import java.util.*;
class JPS6
{
    public static void main(String args[])
    {
        Date d1 = new Date(98, 3, 20);
        Date d2 = new Date(98, 2, 20);
        System.out.println("\nDate 1 =" + d1);
        System.out.println("\nDate 2 =" + d2);
        System.out.println("\ndl.before(d2):" + d1.
            before(d2));
        System.out.println("\ndl.before(d2):" + d1.
            after(d2));
    }
}
OUTPUT:
Date 1 =Mon Apr 20 00:00:00 PDT 1998
Date 2 =Fri Mar 20 00:00:00 PST 1998
d1.before(d2):false
d1.before(d2):true
```

Explanation: The constructor form of the method shown in this program is deprecated yet it has been used to show the usage of after and before method of Date class. The constructor of Date shown in this program takes three arguments: year, month and date. In the before method if invoking date is greater than the argument passed, false is returned else, true is returned. The date 20/03/98 (January is 0) is greater than 20/2/98 so before returns true and after returns true.

19.4 THE CALENDAR CLASS

This class is defined within java.util class package. The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR and so on, and for manipulating the calendar fields, such as getting the date of the next week. An instance in time can be represented by a millisecond value that is an offset from the January 1, 1970 00:00:00:000 GMT.

The class provides no public constructor but an instance can be obtained of the Calendar class using static member getInstance() as:

```
Calendar cal = Calendar.getInstance();
```

Calendar's getInstance method returns a Calendar object whose calendar fields have been initialized with the current date and time.

The Calendar class defines various int constants which can be used in the methods of Calendar class. They are given in the Table 19.2.

| | | | |
|-------------|---------------|--------------|-----------------------|
| AM | AM_PM | APRIL | AUGUST |
| DATE | DAY_OF_MONTH | DAY_OF_WEEK | DAY_OF_MONTH_IN_MONTH |
| DAY_OF_YEAR | DECEMBER | FEBUARY | FRIDAY |
| HOUR | HOUR_OF_DAY | JANUARY | JULY |
| JUNE | MARCH | MAY | MILLISECOND |
| MINUTE | MONDAY | MONTH | NOVEMBER |
| OCTOBER | PM | SATURDAY | SECOND |
| SEPTEMBER | SUNDAY | THRUSDAY | TUESDAY |
| WEDNESDAY | WEEK_OF_MONTH | WEEK_OF_YEAR | YEAR |

Table 19.2 Various int constants defined by Calendar class

Usage of these constant along with the methods of the Calendar class is illustrated using various programs given later.

The various frequently used methods of Calendar class are shown in the Table 19.3.

| Method Signature | Description |
|--------------------------------------|---|
| abstract void add(int which int val) | Adds val to the time or date component specified by which. To subtract, add a negative value, which must be one of the fields defined by Calendar, such as Calendar.HOUR. |
| boolean after(Object calendarObj) | Returns true if the invoking Calendar object contains a date that is later than the one specified by calendarObj. Otherwise, it returns false. |

(Continued)

| | |
|--|--|
| boolean before(Object calendarObj) | Returns true if the invoking Calendar object contains a date that is earlier than the one specified by calendarObj. Otherwise, it returns false. |
| boolean equals(Object calendarObj) | Returns true if the invoking Calendar object contains a date that is equal to the one specified by calendarObj. Otherwise, it returns false. |
| final int get(int calendarField) | Returns the value of one component in the invoking objects. The component is indicated by calendarField. Examples of the components that can be requested are Calendar.YEAR, Calendar.MONTH, Calendar.MINUTE and so forth. |
| static Calendar getInstance() | Returns a Calendar object for the default locale and time zone. |
| final Date getTime() | Returns a Date object equivalent to the time of the invoking object. |
| final boolean isSet(int which) | Returns true if the specified time component is set. Otherwise, it returns false. |
| final void set(int which, int val) | Sets the date or time component specified by which to the value specified by val in the invoking object, which must be one of the fields defined by Calendar, such as Calendar.HOUR. |
| final void set(int year, int month, int dayOfMonth) | Sets various date and time components of the invoking object. |
| final void set(int year, int month, int dayOfMonth, int hours, int minutes) | Sets various date and time components of the invoking object. |
| final void set(int year, int month, int dayOfMonth, int hours, int minutes, int seconds) | Sets various date and time components of the invoking object. |
| final void setTime(Date d) | Sets various date and time components of the invoking object. This information is obtained from the Date object d. |

Table 19.3 Few methods of Calendar class

```
/*PROG 19.7 GETTING CURRENT TIME & DATE USING CALENDAR CLASS */

import java.util.Calendar;
class JPS7
{
    public static void main(String args[])
    {
        Calendar cal = Calendar.getInstance();
        int h = cal.get(Calendar.HOUR);
        int m = cal.get(Calendar.MINUTE);
        int s = cal.get(Calendar.SECOND);
        System.out.println("\nTime is "+h+":"+m+":"+s);
        int day = cal.get(Calendar.DAY_OF_MONTH);
        int mon = cal.get(Calendar.MONTH);
        mon++;
    }
}
```

```

        int yr = cal.get(Calendar.YEAR);
        System.out.println("\nDate is "+day+"/"+mon+"/"+yr);
    }
}

OUTPUT:

Time is 9:41:28
Date is 14/1/2009

```

Explanation: The `Calendar` class is defined within the `java.util` package. Objects of this class cannot be created but an instance of this class can be obtained using static method `getInstance`. The returned instance can be stored in a reference of `Calendar` class. This is what was done in the line `Calendar cal = Calendar.getInstance();`

There are a number of static fields defined within the `Calendar` class which are given in the table above. Using these fields and `get` method, their value can be obtained. In this program, the current **hour**, **minute** and **second** have been obtained as:

```

int h = cal.get(Calendar.HOUR);
int m = cal.get(Calendar.MINUTE);
int s = cal.get(Calendar.SECOND);

```

Similarly, day of month, current month and year are obtained as:

```

int day = cal.get(Calendar.DAY_OF_MONTH);
int mon = cal.get(Calendar.MONTH);
int yr = cal.get(Calendar.YEAR);

```

Note the value of month **January is 0** so `mon++` is written before displaying the output.

```

/*PROG 19.8 DISPLAYING DAY OF WEEK */

import java.util.*;
class JPS8
{
    public static void main(String args[])
    {
        Calendar cal = Calendar.getInstance();
        String day = null;
        int s = cal.get(Calendar.DAY_OF_WEEK);
        switch (s)
        {
            case 1: day = "Sunday";
                      break;
            case 2: day = "Monday";
                      break;
            case 3: day = "Tuesday";
                      break;
            case 4: day = "Wednesday";
                      break;
            case 5: day = "Thursday";
                      break;
            case 6: day = "Friday";
                      break;
            case 7: day = "Saturday";
                      break;
        }
        System.out.println("Day is "+day);
    }
}

```

```

        case 5: day = "Thrusday";
                   break;
        case 6: day = "Friday";
                   break;
        case 7: day = "Saturday";
                   break;
    }
    System.out.println("\nToday is " + day);
}
}

OUTPUT:

```

Today is Wednesday

Explanation: The line `cal.get(Calendar.DAY_OF_WEEK);` gives current day of the week as numeric value between 1 and 7 (inclusive both) **Sunday is 1**. Rest is self-explanatory.

```

/*PROG 19.10 USING SET AND GET METHOD */

import java.util.*;
class JPS10
{
    public static void main(String[] args)
    {
        Calendar cal = Calendar.getInstance();
        cal.set(2007, 9, 01, 6, 25, 15);
        int h = cal.get(Calendar.HOUR);
        int m = cal.get(Calendar.MINUTE);
        int s = cal.get(Calendar.SECOND);
        System.out.println("\nTime os "+h+":"+m+":"+s);
        int day = cal.get(Calendar.DAY_OF_MONTH);
        int mon = cal.get(Calendar.MONTH);
        mon++;
        int yr = cal.get(Calendar.YEAR);
        System.out.println("\nDate is
                           "+day+"/"+mon+"/"+yr);
    }
}

OUTPUT:

```

Time os 6:25:15
Date is 1/10/2007

Explanation: This program demonstrates the usage of set method. There are a number of overloaded form of the set method but only those have been used that set year, month, day, hour, minutes and seconds. The same are displayed back using the get method and int constants of Calendar class.

19.5 THE RANDOM CLASS

The class is defined within `java.util` package. An instance of this class is used to generate a stream of pseudorandom numbers. These numbers are called so, as they are uniformly distributed over a range of numbers. The numbers are generated randomly on the basis of some initial value known as seed value. There are two different constructor forms of this class:

1. **public Random()**

This form of constructor creates a new random number generator. The default seed for this is the current time.

2. **public Random(long seed);**

This form of constructor creates a new random number generator using a single long seed supplied as argument. The method internally uses `setSeed` method for setting the argument seed as:

```
public Random(long seed)
{
    setSeed(seed);
}
```

Initializing the random number generator with initial seed means providing starting value for the random sequence and if the same seed value is to be kept for another random number generator, the same random sequence is obtained. For example:

```
Random rand1 = new Random(20);
Random rand2 = new Random(20);
for(int i = 0; i<5; i++)
{
    System.out.print(Math.abs(rand1.nextInt()%100)+"\t");
    System.out.print(Math.abs(rand2.nextInt()%100));
    System.out.println();
}
```

In this program, the starting seed for the object `rand1` and `rand2` is same. So they both print the same random sequence. The method `nextInt` returns a random number uniformly distributed over the range of int data type, that is, a 32 bit number between the ranges -2^{31} to $-2^{31}-1$. The remainder of number by 100 guarantees that the number will not go beyond 99 but it may be negative so it is converted to positive value with the help of `abs` function of `Math` class. The output of the above code will be:

| | |
|----|----|
| 90 | 90 |
| 23 | 23 |
| 3 | 3 |
| 73 | 73 |
| 85 | 85 |

In order to have different sequence of random numbers for both the objects `rand1` and `rand2`, the seed value has to be changed. This is fine but what one wants is a different random pattern within range. Whenever the above code is run, the same sequence of random numbers is obtained. In order to get the different pattern of random sequence, seed will have to be provided which changes on each run of the program and the best possible option is current time or some value which changes during every run of the program.

Apart from returning the random integers using `nextInt()`, the various other methods for returning random `long`, `float`, `double`, `boolean`, `bytes`, etc. are given in the Table 19.4.

| Method Signature | Description |
|----------------------------|--|
| boolean nextBoolean() | Returns the next Boolean random number |
| void nextByte(byte vals[]) | Fills vals with randomly generated values. |
| Double nextDouble() | Returns the next double random number |
| Float nextFloat() | Returns the next float random number. |
| Int nextInt() | Returns the next int random number. |
| Int nextInt(int n) | Returns the next int random number within the range zero to n. |
| Long nextLong() | Returns the next long random number. |
| Void setSeed(long newSeed) | Sets the seed value (i.e., the starting point for the random number generator) to that |

Table 19.4 Few methods of Random class

```
/*PROG 19.11 DEMO OF RANDOM CLASS */

import java.util.*;
class JPS11
{
    public static void main(String args[])
    {
        Random rand = new Random(new Date().getTime());
        System.out.println("\nFive random integers");
        for (int i = 0; i < 5; i++)
            System.out.print(rand.nextInt() + " ");
        System.out.print("\nFive random floats\n");
        for (int i = 0; i < 5; i++)
            System.out.print(rand.nextFloat() + " ");
        System.out.print("\nFive random doubles\n");
        for (int i = 0; i < 5; i++)
            System.out.print(rand.nextDouble() + " ");
        System.out.print("\nFive random booleans\n");
        for (int i = 0; i < 5; i++)
            System.out.print(rand.nextBoolean() + " ");
    }
}

OUTPUT:

Five random integers
589206553 -1239907088 -110317826 -1698306371 -254853104
Five random floats
0.13498777 0.29409176 0.066607 0.3974718 0.8770754
Five random doubles
0.8618522631398176 0.8907570072174186 0.9432607246587728
0.33741845895889677 0.788368640427357
Five random booleans
false true false true true
```

Explanation: As clear from the output of the above program, the numbers are purely in range and may not be initialized same in various programming situations. Therefore, abs function and arithmetic operators like +, -, %, * etc. have to be used to bring the random numbers in the desired range. For doing this, one example was presented in the beginning.

```
/*PROG 19.12 RANDOM VALUES USING RANDOM CLASS */

import java.util.Random;
import java.util.Date;
class JPS12
{
    public static void main(String[] args)
    {
        Date d = new Date();
        Random rand = new Random(d.getTime());
        int ran;
        System.out.println("\n10 Random values between 90
                           and 100");
        for (int i = 0; i < 10; i++)
        {
            ran = (rand.nextInt()) % 10;
            ran = (int)(Math.abs(ran));
            ran = ran + 90;
            System.out.print(ran + " ");
        }
    }
}
OUTPUT:
10 Random values between 90 and 100
97 95 91 92 98 93 97 92 95 91
```

Explanation: The program is simple to understand.

19.6 OBSERVABLE CLASS AND OBSERVER INTERFACE

The class & interface is defined within `java.util` package. The `Observable` class represents an observable object. It can be subclassed to represent an object that the application wants to have observed. When an object of such a subclass undergoes a change, observing classes are notified. The observing classes must implement observer interface. A class can implement the `Observer` interface when it wants to be informed of changes in observable objects. The observer interface defines just one method `update` whose signature is as shown:

```
void update(Observable o, Object arg);
```

The method takes two parameters: first is the object of `Observable` class which is being observed, and second is the value passed by `notifyObservers()`, a method of `Observable` class. An application calls an `Observable` object's `notifyObservers` method to have all the object's observers notified of the change.

The concept can be understood with an example.

```
/*PROG 19.13 DEMONSTRATE THE OBSERVABLE CLASS AND THE
OBSERVER INTERFACE VER 1 */
```

```
import java.util.*;
class Spy implements Observer
{
    public void update(Observable obj, Object arg)
    {
        System.out.print("\nupdate() called string ");
        System.out.println(" is : " + (String)arg.
            toString());
    }
}
class BeingSpied extends Observable
{
    String str = "Hello Observer";
    void demo()
    {
        setChanged();
        notifyObservers(new String(str));
    }
}
class JPS13
{
    public static void main(String args[])
    {
        BeingSpied observed = new BeingSpied();
        Spy observing = new Spy();
        observed.addObserver(observing);
        observed.demo();
    }
}
OUTPUT:
update () called string is : Hello Observer
```

Explanation: The class Spy implements Observer interface and implements the update method. The class BeingSpied extends the Observer class. The class Spy becomes the observer and the class BeingSpied becomes class to be observed by Spy class. Whenever the class is being observed it must call the method setChanged() and notifyObservers in order. The method setChanged marks this Observable object as having been changed. The method notifyObservers notifies the observer object about the change and calls the update method. Note when update method is called the second argument receives the argument sent by notifyObservers method.

This program has a demo method in the class BeingSpied. The first line in the class changes the default value of the String str to '**Hello Observer**'. The second line in the method demo is the setChanged() method which means some change has occurred in the class. This is informed to the observers using notifyObservers method. In the method the String str is passed. This notifyObservers method calls update method in the Spy class. As the receiving argument is of Object class type, typecasting by String is done and toString method is applied to get the String form of the argument arg.

In the main, note an object of Spy class is added as an observer using the addObserver method of Observable class, then demo method is called.

There are two forms of notifyObservers method. One form is seen with one argument. The second form does not take any argument and a null is passed to second argument of update method. An example of this is given below:

```
/*PROG 19.14 DEMONSTRATE THE OBSERVABLE CLASS AND THE OBSERVER
INTERFACE VER 2 */
```

```
import java.util.*;
class Spy implements Observer
{
    public void update(Observable obj, Object arg)
    {
        System.out.println("\nupdate() is called ");
    }
}
class BeingSpied extends Observable
{
    void demo()
    {
        setChanged();
        notifyObservers();
    }
}
class JPS14
{
    public static void main(String args[])
    {
        BeingSpied observed = new BeingSpied();
        Spy observing = new Spy();
        observed.addObserver(observing);
        observed.demo();
    }
}
OUTPUT:
update() is called
```

Explanation: This program is simple to understand. As soon as demo is called, the notifyObservers method in its body calls update method for its observer.

```
/*PROG 19.15 DEMONSTRATE THE OBSERVABLE CLASS AND THE OBSERVER
INTERFACE VER 3 */
```

```
import java.util.*;
class Spy implements Observer
{
    public void update(Observable obj, Object arg)
    {
```

```

        System.out.println("\nFrom Update count is := " +
                           ((Integer)arg).intValue());
    }
}
class BeingSpied extends Observable
{
    void counter(int count)
    {
        for (; count >= 1; count--)
        {
            setChanged();
            notifyObservers(new Integer(count));
        }
    }
}
class JPS15
{
    public static void main(String args[])
    {
        BeingSpied observed = new BeingSpied();
        Spy observing = new Spy();
        observed.addObserver(observing);
        observed.counter(5);
    }
}
OUTPUT:
From Update count is := 5
From Update count is := 4
From Update count is := 3
From Update count is := 2
From Update count is := 1

```

Explanation: This time there is a `counter` method inside the `BeingSpied` class. The method takes one argument `count` of integer type and runs a `for` from `count` to 1. In each iteration, it changes the value of `count` and call `setChanged` and `notifyObservers` method. In this method, a new `Integer` object is constructed with value of `count`. Now the update method is called for each iteration of `for` loop and it displays the value of `count` from 5 to 1. This is possible as `Object` class is a parent of every other class in Java. Integer value from an `Integer` object is extracted using `intValue()` method.

The various methods defined by `Observable` class are shown in the Table 19.5.

| Method Signature | Description |
|--|--|
| <code>void addObserver(Observer obj)</code> | Add <code>obj</code> to the list of objects observing the invoking object. |
| <code>int countObservers(Observer obj)</code> | Returns the number of objects observing the invoking object. |
| <code>void deleteObserver(Observer Obj)</code> | Removes <code>obj</code> from the list of objects observing the invoking object. |
| <code>void deleteObservers()</code> | Removes all observers for the invoking object |

| | |
|----------------------------------|---|
| Void notifyObservers() | Notifies all observers of the invoking object that it has changed by calling update(). A null is passed as the second argument to update(). |
| Void notifyObservers(Object obj) | Notifies all observers of the invoking object that it has changed by calling update(). obj is passed as an argument to update(). |

Table 19.5 Methods of Observable class

An observable object can have one or more observers. After an observable instance changes, an application calling the Observable's `notifyObservers` method causes all of its observers to be notified of the change by a call to their `update` method. Consider the next program.

```
/*PROG 19.16 MORE THAN ONE OBSERVER */

import java.util.*;
class Spy1 implements Observer
{
    public void update(Observable obj, Object arg)
    {
        System.out.println("From Spy1 count is " +
                           ((Integer)arg).intValue());
    }
}
class Spy2 implements Observer
{
    public void update(Observable obj, Object arg)
    {
        System.out.println("From Spy2 count is " +
                           ((Integer)arg).intValue());
    }
}
class BeingSpied extends Observable
{
    void counter(int count){
        for (; count >= 1; count--)
        {
            setChanged();
            notifyObservers(new Integer(count));
        }
    }
}
class JPS16
{
    public static void main(String args[])
    {
        BeingSpied observed = new BeingSpied();
        Spy1 observing1 = new Spy1();
        Spy2 observing2 = new Spy2();
        observed.addObserver(observing1);
```

```

        observed.addObserver(observing2);
        System.out.print("\n# of Observers:");
        System.out.println(observed.countObservers());
        observed.counter(5);
        observed.deleteObservers();
        System.out.print("# of Observers:");
        System.out.println(observed.countObservers());
    }
}

OUTPUT:

# of Observers: 2
From Spy2 count is 5
From Spy1 count is 5
From Spy2 count is 4
From Spy1 count is 4
From Spy2 count is 3
From Spy1 count is 3
From Spy2 count is 2
From Spy1 count is 2
From Spy2 count is 1
From Spy1 count is 1
# of Observers: 0

```

Explanation: This program is similar to the previous one but here there are two observers implemented in the form of classes Spy1 and Spy2. In the main, an object each of class Spy1 and Spy2 is created and added as observer for the BeingSpied class object, observed using addObserver method. The number of observers is displayed using countObservers method. Note the update method for both the observers is called in turn. In this program, countObservers and deleteObservers method are also used. Before calling counter method, the countObservers method returns return2, and after calling counter method, all the observers are deleted, calling countObserver now returns 0.

19.7 THE SYSTEM CLASS

The System class defined in package java.lang contains several useful class fields and methods. It cannot be instantiated. Among the facilities provided by the System class are standard input, standard output and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

The various methods defined by System class are shown in the Table 19.6.

| Method Signature | Description |
|--|--|
| static void arraycopy(Object src, Object tar, int ts, int s) | Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array of specified length. |
| static long currentTimeMillis() | Returns the current time in terms of milliseconds since midnight, January 1, 1970. |
| static void exit(int status) | Terminates the currently running Java Virtual Machine. The argument serves as a status code; by convention, a non-zero status code indicates abnormal termination. |

| | |
|--|---|
| static void gc() | Runs the garbage collector. |
| static Properties getProperties() | Returns the properties associated with the Java runtime system |
| static String getProperty(String which) | Returns the property associated with which. A null object is returned if the desired property is not found. |
| static String getProperty (String which, String default) | Returns the property associated with which. If the desired property is not found, default is returned. |
| static SecurityManagement SecurityManager() | Returns the current security manager or a null object if no security manager is installed. |
| static void load(String library, FileName) | Loads the dynamic library whose file is specified by libraryFileName, which must specify its complete path. |
| static String setProperty (String which, String v) | Assigns the value v to the property name which. |
| static void setSecurityManager (SecurityManager secMan) | Sets the security manager to that specified by secMan. |

Table 19.6 Some useful methods of System class

Some of the methods are discussed programmatically.

```
/*PROG 19.17 COPYING ARRAY */

class JPS17
{
    public static void main(String[] args)
    {
        int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int[] copy = new int[arr.length/2];
        System.arraycopy(arr, 0, copy, 0, arr.length/2);
        System.out.println("\nElements are\n");
        for (int i = 0; i < copy.length; i++)
            System.out.print(copy[i] + " ");
        System.out.println("\n");
    }
}

OUTPUT:

Elements are
1 2 3 4 5
```

Explanation: The static method `arraycopy` of `System` class copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array. The first argument is source array name, second is the starting position from which to start copying, third is the name of the destination array, fourth is the starting position in the destination array at which elements are stored and the last argument is the number of elements to be copied. In this program, first 5 elements are copied from source `arr` to destination array `copy`.

```
/*PROG 19.18 FINDING EXECUTION TIME OF A CODE SEGMENT */
```

```
class JPS18
{
    public static void main(String[] args)
    {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 10000000; i++) {
        }
        long end = System.currentTimeMillis();
        long exec_time = end-start;
        System.out.println("\nExecution time := "
                           +exec_time);
    }
}
OUTPUT:
Execution time: = 938
```

Explanation: The method `currentTimeMillis()` returns the current time in milliseconds. Before the execution of the `for` loop, starting time is stored in `start`. The ending time is stored in the `end`. The difference of these two gives execution time of `for` loop.

19.7.1 Environment Variables/Property

Environment variables are variables related with the present operating system or present Java working environment. They provide different system setting and path to various directories in the current OS. The various properties/environment variables are given in the Table 19.7. The set of system properties always includes values for the following keys/properties.

| Key/Properties | Description of Associated Value |
|--|---|
| <code>java.version</code> | Java runtime environment version. |
| <code>java.vendor</code> | Java runtime environment vendor. |
| <code>java.vendor.url</code> | Java vendor URL. |
| <code>java.home</code> | Java installation directory. |
| <code>java.vm.specification.version</code> | Java virtual machine specification version. |
| <code>java.vm.specification.vendor</code> | Java virtual machine specification vendor. |
| <code>java.vm.specification.name</code> | Java virtual machine specification name. |
| <code>java.vm.version</code> | Java virtual machine implementation version. |
| <code>java.vm.vendor</code> | Java virtual machine implementation vendor. |
| <code>java.vm.name</code> | Java virtual machine implementation name. |
| <code>java.specification.version</code> | Java runtime environment specification version. |
| <code>java.specification.vendor</code> | Java runtime environment specification vendor. |
| <code>java.specification.name</code> | Java runtime environment specification name. |

| | |
|---------------------------------|-----------------------------------|
| <code>java.class.version</code> | Java class format version number. |
| <code>java.class.path</code> | Java class path. |
| <code>os.name</code> | Operating system name. |
| <code>os.arch</code> | Operating system architecture. |
| <code>os.version</code> | Operating system version. |
| <code>file.separator</code> | File separator (“/” on UNIX). |
| <code>path.separator</code> | Path separator (“.” On UNIX). |
| <code>line.separator</code> | Line separator (“\n” on UNIX). |
| <code>user.name</code> | User’s account name. |
| <code>user.home</code> | User’s home directory. |
| <code>user.dir</code> | User’s current working directory. |

Table 19.7 System properties

All of the properties can be accessed as:

```
String p = System.getProperty("os.name");
System.out.println("OS name = "+p);
```

A complete program for some of the properties is given below:

```
/*PROG 19.19 DEMO OF SYSTEM PROPERTIES */

class JPS19
{
    static void show(String s)
    {
        System.out.println(s);
    }
    static String getp(String s)
    {
        return System.getProperty(s);
    }
    public static void main(String args[])
    {
        show("\nJava Version:= " + getp("java.version"));
        show("\nJava Home := " + getp("java.home"));
        show("\nJava Virtual Machine Name := "
            + getp("java.vm.name"));
        show("\nOperating System := " + getp("os.name"));
        show("\nUser Name := " + getp("user.name"));
        show("\nUser Home := " + getp("user.home"));
        show("\nUser Dir := " + getp("user.dir"));
    }
}
```

OUTPUT:

```
Java Version:= 1.6.0
Java Home := C:\Program Files\Java\jre1.6.0
Java Virtual Machine Name := Java HotSpot(TM) Client VM
Operating System :=Windows XP
User Name := Hari Mohan Pandey
User Home := C:\Documents and Settings\Hari Mohan Pandey
User Dir := C:\JPS\ch19
```

19.8 THE OBJECT CLASS

This class is defined within `java.lang` package. Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class. The various methods of this class are shown in the Table 19.8.

| Method Signature | Description |
|--|---|
| <code>Object clone()</code> | Creates a new object that is the same as the invoking object. |
| <code>boolean equals(Object object)</code> | Returns true if the invoking object is equivalent to object. |
| <code>void finalize()</code> | Default <code>finalize()</code> method. This is usually overridden by subclasses. |
| <code>final Class getClass()</code> | Obtains a <code>Class</code> object that describes the invoking object. |
| <code>final void notify()</code> | Resumes execution of a thread waiting on the invoking object. |
| <code>final void notifyAll()</code> | Resumes execution of all threads waiting on the invoking object. |
| <code>String toString()</code> | Returns a string that describes the object. |
| <code>final void wait()</code> | Waits on another thread of execution. |
| <code>final void wait(long ms)</code> | Waits up to the specified number of milliseconds on another thread of execution. |

Table 19.8 Few methods of `Object` class

Note: The method number **1** may throw `CloneNotSupportedException`, number **3** may throw `Throwable` exception and method number **8, 9** may throw `InterruptedException`.

Among a number of methods given in the table of `Object` class, the principal method of importance is the `clone` method. The `getClass` method is also of importance but it will be discussed later. Rest of the methods was seen in some of the programs done earlier.

19.8.1 The Clone Method

The `clone()` methods generates a duplicate copy of the object on which it is called. The method creates and returns a copy of this object. The method `clone` for class `Object` performs a specific cloning operation. If the class of this object does not implement the `Cloneable` interface a `CloneNotSupportedException` is thrown; therefore only classes that implement the `Cloneable` interface can be cloned. A `clone` is an exact copy of the original, that is, all the members are duplicated in the new cloned object except references. The complete signature of the `clone` method is as shown below:

```
protected Object clone() throws CloneNotSupportedException
```

As method clone is protected, it cannot be used directly. To use clone method, either of the two approaches must be followed:

1. Define a class that implements Cloneable interface and use clone method inside any method of that class.
2. Override clone method so that it becomes public.

Both of the approaches are demonstrated in the program as given below:

```
/*PROG 19.20 DEMO OF CLONE METHOD VER 1 */

class mobile implements Cloneable
{
    String model;
    double price;
    mobile(String s, double p)
    {
        model = s;
        price = p;
    }
    void show()
    {
        System.out.println("\nModel = " + model +
                           "\tPrice =" + price);
    }
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            System.out.println("Cloning not allowed");
            return this;
        }
    }
}
class JPS20
{
    public static void main(String args[])
    {
        mobile m1 = new mobile("Nokia 6300", 10000);
        mobile m2;
        m2 = (mobile)m1.clone();
        m1.show();
        m2.show();
        m1.model = "Sony Ericsson K750i";
        System.out.print("\nAfter Changing the model");
        System.out.println(" of first mobile");
        m2.show();
    }
}
```

OUTPUT:

```

Model = Nokia 6300 Price =10000.0
Model = Nokia 6300 Price =10000.0

After changing the model of first mobile

Model = Nokia 6300 Price =10000.0

```

Explanation: In this program, the clone method is overridden so it becomes public. The overridden method returns super.clone (i.e., defaults clone method of Object class). In case cloning is not supported exception may be thrown. Therefore, try catch blocks have been used. If execution is thrown, current object will be returned. In the main, clone method of mobile class is called to make a clone of object **m1**. As the returned type is Object, typecasting by mobile is done. Note both the objects **m1** and **m2** after cloning are identical but different in case of members. Change in members of one object does not affect values of members of other cloned object.

```

/*PROG 19.21 DEMO OF CLONE METHOD VER 2*/
class mobile implements Cloneable
{
    String model;
    double price;
    mobile(String s, double p)
    {
        model = s;
        price = p;
    }
    void show()
    {
        System.out.println("\nModel := " + model +
                           "\tPrice := " + price);
    }
    mobile getClone()
    {
        try
        {
            return (mobile)super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            System.out.println("Cloning not allowed");
            return this;
        }
    }
}
class JPS21
{
    public static void main(String args[])
    {
        mobile m1 = new mobile("Nokia 6300", 10000);
    }
}

```

```

        mobile m2;
        m2 = m1.getClone();
        m1.show();
        m2.show();
        m1.model = "Sony Erricson K750i";
        System.out.println("\nAfter Changing the model of
                           first mobile");
        m2.show();
    }
}

OUTPUT:

Model := Nokia 6300 Price := 10000.0
Model := Nokia 6300 Price := 10000.0
After Changing the model of first mobile
Model := Nokia 6300 Price := 10000.0

```

Explanation: This time for performing cloning, `getClone` is written which returns the cloned object using `super.clone` method. But as the returning object is of `mobile` class this has to be typecasted by the `mobile` class. In the main, the cloning is done by calling `getClone` method by object **m1**. The returned cloned object is stored in **m2**.

```

/*PROG 19.22 DEMO OF CLONE METHOD VER 3 */

class demo1
{
    String str;
    demo1(String s)
    {
        str = s;
    }
}
class demo2 implements Cloneable
{
    demo1 d1;
    int num;
    demo2(int n, String s)
    {
        num = n;
        d1 = new demo1(s);
    }
    demo2 getClone()
    {
        try
        {
            return (demo2)super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("Cloning not allowed");
        }
    }
}

```

```

        return this;
    }
}
class JPS22
{
    public static void main(String args[])
    {
        demo2 A = new demo2(10, "Test");
        demo2 B;
        B = A.getClone();
        A.d1.str = "\nChange is the nature";
        System.out.println(B.d1.str);
    }
}
OUTPUT:
Change is the nature

```

Explanation: This program shows how cloning can be dangerous some times. There is a class `demo1` with just one member `str` of `String` type. A reference of this `demo1` class named `d1` is a member of `demo2` class. The member `str` of `demo1` class is initialized with value ‘Test’ when constructor of `demo2` class is called. Using the `getClone` method of `demo1` class a cloned object is obtained and is stored in `B`. In this cloning operation, only the member `num` is copied to object `B` but both objects share a single copy of reference `d1`. Now, changes performed to `str` through this reference `d1` by any of the object `A` or `B` reflects back to other object. This is what done in this program. The value of `str` is changed to ‘Change is the nature’ using object `A` and same value is displayed using object `B`. The value of `str` for object `B` is the changed value `str` modified by `A`.

19.9 THE Class CLASS

The `Class` class encapsulates the runtime state of an object or interface. Instance of the `Class` class represents classes and interfaces in a running Java application. The primitive Java types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`) and the keyword `void` are also represented as `Class` objects. `Class` has no public constructor; instead `Class` objects are constructed automatically by the Java virtual machine as classes are loaded. The most commonly used methods of `Class` class are shown in the Table 19.9.

| Method Signature | Description |
|--|--|
| <code>static Class forName(String name)</code> | Returns a <code>Class</code> object given its complete name. |
| <code>Class[] getClasses()</code> | Returns a <code>Class</code> object for each of the public classes and interfaces that are members of the invoking object. |
| <code>Constructor[] getConstructors()</code> | Returns a <code>Constructor</code> object for all the public constructors of this class. |
| <code>Constructor[] getDeclaredConstructors()</code> | Returns a <code>Constructor</code> object for all the constructors that are declared by this class. |

| | |
|-------------------------------|--|
| Field[] getDeclaredFields() | Returns a Field object for all the fields that are declared by this class. |
| Method[] getDeclaredMethods() | Returns a Method object for all the methods that are declared by this class or interface. |
| Field[] getFields() | Returns a Field object for all the public fields of this class. |
| Method[] getMethods() | Returns a Method object for all the public methods of this class. |
| String getName() | Returns the complete name of the class or interface of the invoking object. |
| Class getSuperclass() | Returns the superclass of the invoking object. The return value is null if the invoking object is of type Object. |
| boolean isInstance() | Returns true if the invoking object is an interface. Otherwise, it returns false. |
| Object newInstance() | Creates a new instance (i.e., a new object) that is of the same type as the invoking object. This is equivalent to using new with the class default constructor. The new object is returned. |
| String getSimpleName() | Returns the simple name of the underlying class as given in the source code. |

Table 19.9 Few methods of Class class

The usage of some of the methods is demonstrated for better illustration.

```
/*PROG 19.23 GETTING CLASS INFORMATION */

class demo1
{
}
class demo2
{
}
class JPS23
{
    public static void main(String[] args)
    {
        String str = " ";
        Class cls;
        cls = str.getClass();
        System.out.println("\nThe class of str is "
                           +cls.getName());
        demo1 d1 = new demo1();
        cls = d1.getClass();
        System.out.println("The class of d1 is "
                           +cls.getName());
        demo2 d2 = new demo2();
        cls = d2.getClass();
        System.out.println("The class of d2 is "
                           +cls.getName());
        Integer I = 23;
```

```

        cls = I.getClass();
        System.out.println("The class of I is "
                           +cls.getName());
        Float F = 35.45f;
        cls = F.getClass();
        System.out.println("The class of F is "
                           +cls.getName());
    }
}

OUTPUT:

The class of str is java.lang.String
The class of d1 is demo1
The class of d2 is demo2
The class of I is java.lang.Integer
The class of F is java.lang.Float

```

Explanation: The associated class name is obtained at run time by calling `getClass` method from any of the object as shown in the above program. The returned instance must be sorted in a reference of Class type. Then using the `getName` method, the class name is displayed. The class name can be obtained by writing as also:

```

System.out.println("Class is " + String.class.getName());
This displays: Class is java.lang.String

```

Note full class name qualified by package name is displayed using `getName`. In order to have only the class name in the output, use `getSimpleName` which displays only String as compared to `java.lang.String`.

```
/*PROG 19.24 GETTING FIELD INFORMATION */
```

```

import java.lang.reflect.Field;
class demo
{
    int x;
    float y;
    public String s;
    double d;
    char ch;
    public byte b;
}
class JPS24
{
    public static void main(String[] a)
    {
        demo d = new demo();
        Class cls = d.getClass();
        Field[] farr = cls.getDeclaredFields();

```

```

        System.out.println("ALL DATA MEMBERS OF DEMO
                           CLASS \n");
        for (int i = 0; i < farr.length; i++)
            System.out.println(farr[i]);
        farr = cls.getFields();
        System.out.println("\nPUBLIC DATA MEMBERS ");
        System.out.println(" OF DEMO CLASS\n");
        for (int i = 0; i < farr.length; i++)
            System.out.println(farr[i]);
    }
}

OUTPUT:

ALL DATA MEMBERS OF DEMO CLASS

int demo.x
float demo.y
public java.lang.String demo.s
double demo.d
char demo.ch
public byte demo.b

PUBLIC DATA MEMBERS
OF DEMO CLASS

public java.lang.String demo.s
public byte demo.b

```

Explanation: In this program, a class demo is defined which contains a number of members of various data types. Two members are public. The Field class is defined within java.lang.reflect package so this package has to be imported into the program. The getDeclaredFields returns all the data members of the class on which it is called regardless of their access specifier. The number of fields are stored in the array farr and then displayed. The method getFields returns only public fields. This is stored in the Fields array farr and displayed using for loop.

```
/*PROG 19.25 GETTING METHOD INFORMATION */
```

```

import java.lang.reflect.Method;
class demo
{
    public void show() { };
    public int sum(int x, int y)
    {
        return x + y;
    }
    double sqr(double d)
    {
        return d * d;
    }
}

```

```
}

class JPS25
{
    public static void main(String[] args)
    {
        demo d = new demo();
        Class cls = d.getClass();
        Method[] marr = cls.getDeclaredMethods();
        System.out.println("\n ALL DATA METHODS OF DEMO
                           CLASS \n");
        for (int i = 0; i < marr.length; i++)
            System.out.println(marr[i]);

        marr = cls.getMethods();
        System.out.print("\nPUBLIC DATA METHODS");
        System.out.println(" OF DEMO CLASS \n");

        for (int i = 0; i < marr.length; i++)
            System.out.println(marr[i]);
    }
}
```

OUTPUT:

```
ALL DATA METHODS OF DEMO CLASS

public int demo.sum(int,int)
double demo.sqr(double)
public void demo.show()

PUBLIC DATA METHODS OF DEMO CLASS

public int demo.sum(int,int)
public void demo.show()
public native int java.lang.Object.hashCode()
public final native java.lang.Class
    java.lang.Object.getClass()
public final void java.lang.Object.wait(long,int) throws
    java.lang.InterruptedEx
ception
public final void java.lang.Object.wait() throws
    java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws
java.lang.Interrupe
dException
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```

Explanation: To find out various methods of the class, `getDeclaredMethods` and `getMethods` method of the `Class` class can be used. The `getDeclaredMethods` displays all the methods of the class regardless of their visibility mode. The `getMethods` displays all `public` methods of its class along with all the public methods of `Object` class which is the superclass for every class in Java.

```
/*PROG 19.26 OBTAINING SUPER CLASS INFORMATION */

class cdemo { }
class JPS26
{
    public static void main(String args[])
    {
        Class cls = new cdemo().getClass();
        cls = cls.getSuperclass();
        System.out.println("\nSuper class = " +
                           cls.getSimpleName());
    }
}
OUTPUT:
Super class = Object
```

Explanation: This program is simple to understand.

19.10 THE RUNTIME CLASS

The class `Runtime` is an abstraction of runtime environment. Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `getRuntime` method. An application cannot create its own instance of this class, that is, instances of `Runtime` class cannot be created yet a reference of this can be obtained using static method `getRuntime`. The various frequently used methods of this class are given in the Table 19.10.

| Method Signature | Description |
|--|--|
| <code>Process exec(String progName)</code> | Executes the program specified by <code>progName</code> as a separate process. An object of type <code>Process</code> is returned that describes the new process. |
| <code>void exit(int status)</code> | Halts execution and returns the value of <code>status</code> to the parent process. By convention, 0 indicates normal termination. All other values indicate some form of error. |
| <code>long freeMemory()</code> | Returns the approximate number of bytes of free memory available to the Java runtime system. |
| <code>void gc()</code> | Indicates garbage collection. |
| <code>static Runtime getRuntime()</code> | Returns the current <code>Runtime</code> object. |
| <code>void load(String libraryFileName)</code> | Loads the dynamic library whose file is specified by the <code>libraryFileName</code> , which must specify its complete path. |

(Continued)

| | |
|--------------------------------------|---|
| void loadLibrary(String libraryName) | Loads the dynamic library whose name is associated with library name. |
| long totalMemory() | Returns the total number of bytes of memory. |
| long maxMemory() | Returns the maximum amount of memory JVM will attempt to use. |

Table 19.10 Few methods of Runtime class

The practical usage of some of the methods is given below:

```
/*PROG 19.27 FINDING MEMORY INFORMATION */

class JPS27
{
    public static void main(String args[])
    {
        Runtime r = Runtime.getRuntime();

        long fm = r.freeMemory();
        long tm = r.totalMemory();
        long mm = r.maxMemory();

        System.out.println("\nFree Memory="+fm+" Bytes");
        System.out.println("Occupied Memory="+ (tm-fm) +
                           " Bytes");
        System.out.println("Total Memory="+tm+" Bytes");
        System.out.println("Maximum Memory= "+mm+" Bytes");

        int arr[] = new int[65536];
        fm = r.freeMemory();
        tm = r.totalMemory();

        System.out.println("\nAfter allocating 65536 bytes of
                           memory");
        System.out.println("Free Memory = " + fm + " "
                           + Bytes);
        System.out.println("Occupied Memory="+(tm-fm) +
                           "Bytes");
        r.gc();

        fm = r.freeMemory();
        tm = r.totalMemory();

        System.out.println("\nAfter calling method gc");
        System.out.println("Free Memory = " + fm + " "
                           + Bytes);
        System.out.println("Occupied Memory = " + (tm-fm) +
                           " Bytes");
    }
}
```

OUTPUT:

```

Free Memory = 4996984 Bytes
Occupied Memory = 180360 Bytes
Total Memory = 5177344 Bytes
Maximum Memory = 66650112 Bytes
After allocating 65536 bytes of memory
Free Memory = 4734824 Bytes
Occupied Memory = 442520 Bytes

After calling method gc
Free Memory = 4806352 Bytes
Occupied Memory = 370992 Bytes

```

Explanation: The method `freeMemory()` returns the amount of free memory in the Java virtual machine. Calling the `gc` method may result in increasing the value returned by `freeMemory`. In this program, the free memory, total memory and the occupied memory are first shown. An integer array of **65537** bytes is then allocated. Again, `freeMemory` and `occupiedMemory` are shown. This time, `freeMemory` decreases and `occupiedMemory` increases. Next, `gc` method is called which does some garbage collection and returns the unused memory to the system's free memory. This results in increase of free memory. The program also shows the maximum memory returned by `maxMemory` method. The method returns the maximum amount of memory that the Java virtual machine will attempt to use. If there is no inherent limit, the value `Long.MAX_VALUE` will be returned.

The other methods are not used for ordinary programming so they are not discussed here. However, the important method `exec` is discussed in the `Process` class section.

19.11 THE PROCESS CLASS

A process is a program in execution. The `Process` class is an abstraction of the runtime program. The `Runtime.exec` methods create a native process and returns an instance of a subclass of `Process` that can be used to control the process and obtain information about it. The `Process` class provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process and destroying (killing) the process.

The methods that create processes may not work well for special process on certain native platforms, such as native windowing processes, daemon processes, Win16/DOS processes on Microsoft Windows, or shell scripts. The created subprocess does not have its own terminal or console. All its standard I/O (i.e., `stdin`, `stdout` and `stderr`) operations will be redirected to the parent process through three streams (`getOutputStream()`, `getInputStream()`, `getErrorStream()`). The parent process uses these streams to feed input to and get output from the subprocess.

The methods of `Process` class are shown in the Table 19.11.

| Method Signature | Description |
|---|---|
| <code>void destroy()</code> | Terminates the process. |
| <code>Int exitValue()</code> | Returns an exit code obtained from a subprocess |
| <code>InputStream getErrorStream()</code> | Returns an input stream that reads input from the process errs output stream. |

(Continued)

| | |
|--|--|
| <code>InputStream getInputStream()</code> | Returns an input stream that reads from the process out output stream. |
| <code>OutputStream
getOutputStream()</code> | Returns an output stream that writes output to the process in input stream. |
| <code>Int waitFor() throws
InterruptedException</code> | Returns the exit code returned by the process. This method does not return until the process on which it is called terminates. |

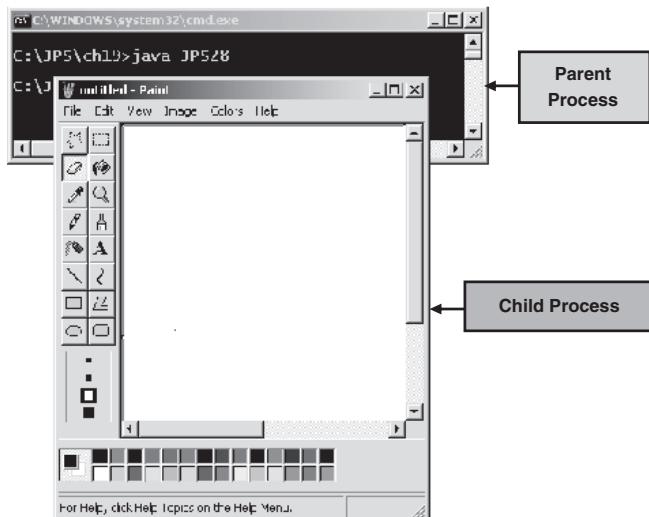
Table 19.11 Few methods of Process class

Now that the `Process` class is understood, few programs can be presented which make use of methods of `Runtime` class.

```
/*PROG 19.28 DEMO OF PROCESS CLASS, EXECUTING OTHER PROCESS VER 1 */
```

```
class JPS28
{
    public static void main(String args[])
    {
        Runtime r = Runtime.getRuntime();
        Process p = null;
        try
        {
            p = r.exec("mspaint");
        }
        catch (Exception e)
        {
            System.out.println("Error executing paint");
        }
    }
}
```

OUTPUT:



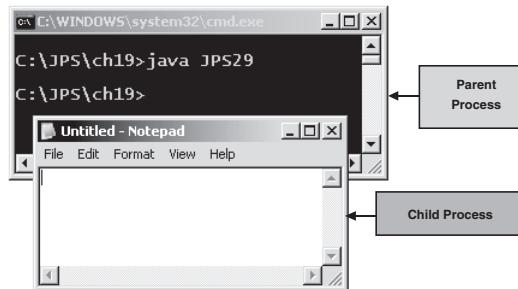
Explanation: The method `exec` is used for executing the given command as its argument as a separate process. The method returns a new `Process` object that describes the new process. In this program, `exec` method is used which takes parameter '`mspaint`' (i.e., '`mspaint`' is run from within the program). The main program in execution is the parent of the new process created and the new process '`mspaint`' is the child process. Note parent process does not wait for the child process to be terminated. Observe '**Press any key to continue**' in the parent window. In case one wants parent process to wait for child process `waitFor` method of `Process` class can wait as shown in the next program (prog19.30).

If the child process is '`notepad`' then program will be:

```
/*PROG 19.29 DEMO OF PROCESS CLASS, EXECUTING OTHER PROCESS VER 2 */
```

```
class JPS29
{
    public static void main(String args[])
    {
        Runtime r = Runtime.getRuntime();
        Process p = null;
        try
        {
            p = r.exec("notepad");
        }
        catch (Exception e)
        {
            System.out.println("Error executing paint");
        }
    }
}
```

OUTPUT:



Explanation: This program is simple to understand.

```
/*PROG 19.30 DEMO OF PROCESS CLASS, EXECUTING OTHER PROCESS
VER 3 */
```

```
class JPS30
{
    public static void main(String args[])
    {
```

```

        Runtime r = Runtime.getRuntime();
        Process p = null;
        try
        {
            p = r.exec("mspaint");
            p.waitFor();
        }
        catch (Exception e)
        {
            System.out.println("Error executing paint");
        }
    }
}

```

OUTPUT:



Explanation: The method `waitFor` called by object process `p` (actually `mspaint` process) does not return until the process on which it is called terminates. Note this time, parent does not terminate. Observe there is no ‘press any key to continue’ written onto the parent window.

19.12 WRAPPER CLASS

A primitive wrapper class in the Java programming language is one of eight classes provided in the `java.lang` package which provide object methods for the eight primitive types: `byte`, `short`, `long`, `int`, `float`, `char`, `boolean` and `double`. These classes represent the primitive values as objects. Objects of wrapper classes are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed. These classes are needed when an object representation needs to be created for one of these simple types. The classes are known as wrapper classes or wrappers as they wrap the simple types within a class. The primitive types and the corresponding wrapper classes are shown below.

| Primitive Type | Wrapper Class | Primitive Type | Wrapper Class |
|--------------------|----------------------|----------------------|------------------------|
| <code>byte</code> | <code>Byte</code> | <code>float</code> | <code>Float</code> |
| <code>short</code> | <code>Short</code> | <code>double</code> | <code>Double</code> |
| <code>int</code> | <code>Integer</code> | <code>char</code> | <code>Character</code> |
| <code>long</code> | <code>Long</code> | <code>boolean</code> | <code>Boolean</code> |

In the following section, all the wrapper classes are discussed in detail.

19.12.1 The Number Class

The abstract Number class is the superclass of classes Byte, Double, Float, Integer, Long and Short. Subclasses of Number class must provide methods to convert the represented numeric value to byte, double, float, int, long and short. For that the class has six methods which the subclasses must define. They are given below.

| Method Signature | Description |
|----------------------|--|
| byte byteValue() | Returns the value of the specified number as a byte. |
| short shortValue() | Returns the value of the specified number as a short. |
| int intValue() | Returns the value of the specified number as a int. |
| float floatValue() | Returns the value of the specified number as a float. |
| long longValue() | Returns the value of the specified number as a long. |
| double doubleValue() | Returns the value of the specified number as a double. |

All the methods may involve rounding or truncation.

19.12.2 The Byte, Short, Integer and Long Class

All four wrapper classes wrap a value of corresponding primitive type in an object. To refer each of them by a common name, ITYPE word is used. Now whenever ITYPE is mentioned, it means it is applicable to all four classes and itype their corresponding primitive type. An object of type ITYPE contains a single field whose type is itype. In addition, this class provides several methods for converting an itype to a String and a String to itype, as well as other constants and methods useful when dealing with an itype. All wrapper class ITYPE define the two constructors: one that takes the primitive value and another that takes the String representation of the value. For example, constructor for Integer class is shown as:

1. **public Integer(int value)**
e.g., Integer i1 = new Integer(50);
2. **public Integer(String s) throws NumberFormatException**
e.g., Integer i2 = new Integer("50");

Same type of constructors are there for Byte, Long and Short classes.

All the classes have the following static constants:

| | |
|----------------------|--|
| static int MAX_VALUE | A constant holding the maximum value of type |
| static int MIN_VALUE | A constant holding the minimum value of type. |
| static int SIZE | The number of bits used to represent a type's value in two's complement binary form. |
| static Class TYPE | The Class instance representing the primitive type. |

All the classes provide a number of methods for parsing, for converting from primitive to objects and for base conversions. Following is a list of methods of all the classes:

| Method Signature | Description |
|--|--|
| <code>int compareTo(Byte anotherByte)</code> | Compares two Byte object numerically. |
| <code>static byte parse(String s) throws NumberFormatException</code> | Parses the string argument as a signed decimal byte. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign ‘-’ to indicate a negative value. |
| <code>static byte parseByte(String s, int radix) throws NumberFormatException</code> | Parses the string argument as a signed byte in the radix specified by the second argument. |
| <code>String toString()</code> | Returns a String object representing this Byte’s value. |
| <code>static String toString(byte b)</code> | Returns a new String object representing the specified byte. The radix is assumed to be 10. |
| <code>static BytevalueOf(byte b)</code> | Returns a Byte instance representing the specified byte value. |
| <code>static BytevalueOf(String) throws NumberFormatException</code> | Returns a Byte object holding the value given by the specified String. |
| <code>static BytevalueOf(String s, int radix) throws NumberFormatException</code> | Returns a Byte object holding the value extracted from the specified String when parsed with the radix given by the second argument. |

Table 19.12 Frequently used methods of Byte class

The `compareTo` method is overloaded for `Byte`, `Long` and `Integer` classes. Similar to `parseByte` method, `parseShort`, `parseLong` and `parseInteger` methods are there for `Short`, `Long` and `Integer` class, respectively.

The three overloaded `valueOf` methods are also provided for the other three classes with the difference that the return type is the respective class name (i.e., for `Integer` class the return type is `Integer` and so on).

The `toString` methods are also the same for `Short`/`Long`/`Integer` class where in second form the argument passed is of respective primitive type.

The `Integer` and `Long` class defines one additional form of `toString` method that is shown below.

| Method Signature | Description |
|--|--|
| <code>static String toBinaryString(int i)</code> | Returns a string representation of the integer argument as an unsigned integer in base2. |
| <code>static String toHexString(int i)</code> | Returns a string representation of the integer argument as an unsigned integer in base 16. |
| <code>static String toOctalString(int i)</code> | Returns a string representation of the integer argument as an unsigned integer in base 8. |

The same method also exists for `Long` class where they take long value.

Most of the methods have been used in the previous chapters so they will not be discussed here again. Few methods used in this program are given below:

```

/*PROG 19.31 BASE CONVERSION */

import java.util.*;
import java.io.*;
class JPS31
{
    static void show(String s)
    {
        System.out.print(s);
    }
    public static void main(String args[]) throws IOException
    {
        Scanner sc = new Scanner(System.in);
        show("\n Enter an integer number:= ");
        int num = sc.nextInt();
        show("\n Integer number := " + num);
        show("\nBinary Equivalent:=" + Integer.toBinaryString(num));
        show("\nOctal Equivalent:=" + Integer.toOctalString(num));
        show("\n Hex Equivalent:=" + Integer.toHexString(num));
    }
}

```

OUTPUT:

```

Enter an integer number: = 15
Integer number := 15
Binary Equivalent := 1111
Octal Equivalent := 17
Hex Equivalent := f

```

Explanation: In this program, an integer number `num` is read and converted into binary, octal and hexadecimal using the methods `toBinaryString`, `toOctalString` and `toHexString`, respectively. The equivalent code for the conversion can be written as:

```

show("Binary Equivalent:=" + Integer.toString(num, 2));
show("Octal Equivalent:=" + Integer.toString(num, 8));
show("Hex Equivalent:=" + Integer.toString(num, 16));

```

19.12.3 The Float and Double Class

The `Float/Double` class wraps a value of primitive type `float/double` in an object. An object of type `Float/Double` contains a single field whose type is `float/double`.

Similar to the four integral wrapper classes seen in the previous section, the above classes provide two constructors: one that takes the primitive value and another that takes the String representation of the value. But `Float` class provides one extra constructor as shown below:

```

Float(float value)
Float(String s) throws NumberFormatException

```

The extra constructor for `Float` class is:

```
Float(double value)
```

This form of constructor creates a newly allocated `Float` object that represents the argument converted to type `float`.

Both the classes define some static constants that are shown below:

| Method Signature | Description |
|--------------------------------------|---|
| <code>MAX_VALUE</code> | A constant holding the largest positive finite value of type <code>float</code> / <code>double</code> . |
| <code>MIN_VALUE</code> | A constant holding the smallest positive finite value of type <code>float</code> / <code>double</code> . |
| <code>POSITIVE_INFINITY</code> | A constant holding the negative infinity of type <code>float</code> / <code>double</code> . |
| <code>NEGATIVE_INFINITY</code> | A constant holding the negative infinity of type <code>float</code> / <code>double</code> . |
| <code>NaN</code> | A constant holding a Not-a-Number (<code>NaN</code>) value of type <code>float</code> / <code>double</code> . |
| <code>Static final int SIZE</code> | The number of bits used to represent a <code>float</code> / <code>double</code> value. |
| <code>Static final Class TYPE</code> | The class instance representing the primitive type <code>float</code> / <code>double</code> . |

Note: First five constants are of type `static final float`/`double`.

19.12.4 Methods of `Float`/`Double` Class

The six methods defined by parent class `Number` are not shown. The commonly used methods of `Float` class are as follows in Table 19.13.

| Method Signature | Description |
|--|---|
| <code>Static int compare(float f1, float f2)</code> | Compares the two specified float values. |
| <code>Int compareTo(Float anotherFloat)</code> | Compares two float objects numerically. |
| <code>Boolean isInfinite()</code> | Returns true if this <code>Float</code> value is infinitely large in magnitude, false otherwise. |
| <code>Static boolean is Infinite(float v)</code> | Returns true if the specified number is infinitely large in magnitude, false otherwise. |
| <code>boolean isNaN()</code> | Returns true if this <code>Float</code> value is a Not-a-Number (<code>NaN</code>), false otherwise. |
| <code>Static boolean isInfinite(float v)</code> | Returns true if the specified number is infinitely large in magnitude, false otherwise. |
| <code>String toString()</code> | Returns a string representation of this <code>Float</code> object. |
| <code>Static String toString(float f)</code> | Returns a string representation of the float argument. |
| <code>Static Float valueOf(float f)</code> | Returns a <code>Float</code> instance representing the specified float value. |
| <code>Static Float valueOf(String s) throws NumberFormatException</code> | Returns a <code>Float</code> object holding the float value represented by the argument string <code>s</code> . |

Table 19.13 Method of `Float`/`Double` class

All methods listed in the table are applicable to `Double` class with float argument in any of the method being changed to double and `Float` class to `Double` class.

```
/*PROG 19.32 DEMO FO CONSTANTS OF FLOAT/DDOUBLE CLASS */
```

```
class JPS32
{
    static void show(String s){
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        show("\n*****");
        show("Constants for Float");
        show("\n*****");
        show("Size if bits :" + Float.SIZE);
        show("Maximum value:" + Float.MAX_VALUE);
        show("Mimimum value:" + Float.MIN_VALUE);
        show("Type:" + Float.TYPE);

        show("\n*****");
        show("Constants for Double");
        show("\n*****");
        show("Size in bits:" + Double.SIZE);
        show("Maximum value:" + Double.MAX_VALUE);
        show("Mininum value:" + Double.MIN_VALUE);
        show("Type:" + Double.TYPE);
        show("\n*****");
        show("\nPositive Infinity:" + Float.POSITIVE_INFINITY);
        show("\nNegative Infinity:" + Float.NEGATIVE_INFINITY);
        show("\n*****");
    }
}

OUTPUT:
*****
Constants for Float
*****
Size if bits: 32
Maximum value:3.4028235E38
Mimimum value:1.4E-45
Type: float
*****
Constants for Double
*****
Size in bits:64
Maximum value:1.7976931348623157E308
Minimum value:4.9E-324
```

```
Type:double
*****
Positive Infinity: Infinity
Negative Infinity:-Infinity
*****
```

Explanation: In this program, the values of various static constants are shown for both Float and Double class. The program is not hard to understand by seeing its output.

```
/*PROG 19.33 USAAGE OF FEW METHODS OF FLOAT/DDOUBLE CLASS VER 1 */
```

```
class JPS33
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        Float F = new Float(23.45f);
        Double D = new Double(23.12);

        float f = F.floatValue();
        double d = D.doubleValue();

        show("\nfloat value = "+f+"\tdouble value =" +d);
        String fval = "float value:" + F.toString(f);
        String dval = "double value:" + D.toString(d);
        show(fval + "\t" + dval);

        String s = "23.45f";
        Float F1 = Float.valueOf(s);

        System.out.println("F1.compareTo(F) :");
        show(F1.compareTo(F) == 0 ? "true" : "false");
    }
}
OUTPUT:
float value = 23.45      double value = 23.12
float value: 23.45      double value: 23.12
F1.compareTo(F): true
```

Explanation: The floatValuedoubleValue method of Float/Double class extracts a float/ double value from Float/Double object. The toString method takes a float/double value and returns the String object equivalent to that value. The valueOf method takes a primitive value as argument and returns the equivalent Float object. The compareTo method compares two objects numerically and returns 0 if both are same, negative if first one is smaller and positive if first one is larger than second.

```
/*PROG 19.34 USAGE OF FEW METHODS OF FLOAT/DOUBLE CLASS VER 2 */
```

```
import java.util.*;
import java.io.*;
class JPS34
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        boolean b1 = new Double(34/0.).isInfinite();
        boolean b2 = Double.isInfinite(34/0.);
        show("\n34/0 is infinite : " + (b1&&b2));
        b1 = new Double(0/0.).isNaN();
        b2 = Double isNaN(0/0.);
        show("\n0/0 is Not a Number: " +(b1&&b2));
    }
}
OUTPUT:
34/0 is infinite : true
0/0 is Not a Number: true
```

Explanation: The `isInfinite` and `isNaN` method has two forms as given in the table above. Both the forms return true if number in context is infinite or **Not-a-Number**. Both the forms of `isInfinite` method are used for checking the **34/0** for infiniteness. The returned value is stored in `b1` and `b2`. Now both `b1` and `b2` will be storing **true** value so `b1 && b2` will return **true**. The expression **0/0.** is not a number so when checked using `isNaN` method, it returns **true**.

19.12.5 The Character Class

The `Character` class wraps a value of the primitive type `char` in an object. An object of type `Character` contains a single field whose type is `char`. The class has got just one constructor:

```
Character(char value)
```

The various constructor a newly allocated `Character` object representing the specified `char` value.

The various constants defined by `Character` are not significant for general programming so they are not discussed here.

A number of useful methods defined by `Character` class are given in the Table 19.14.

| Method Signature | Description |
|---|--|
| <code>char charValue()</code> | Returns the value of this character object. |
| <code>int compareTo(Character chObj)</code> | Compares two character objects numerically. |
| <code>static int digit(char ch, int radix)</code> | Returns the numeric value of the character <code>ch</code> in the specified radix. |

(Continued)

| | |
|---|---|
| static char forDigit(int digit, int radix) | Determines the character representation for a specific digit in the specified radix. |
| static boolean isDefined(char ch) | Determines if a character is defined in Unicode |
| static boolean isDigit (char ch) | Determines if the specified character is a digit. |
| static boolean isJavaIdentifierPart(char ch) | Determines if the specified character may be part of a Java identifier as other than the first character. |
| static boolean isJavaIdentifierStart(char ch) | Determines if the specified character is permissible as the first character in a Java identifier. |
| static boolean isLetter(char ch) | Determines if the specified character is a letter. |
| static boolean isLetterOrDigit (char ch) | Determines if the specified character is a letter or digit. |
| static boolean isLowerCase (char ch) | Determines if the specified character is a lowercase character. |
| static boolean isSpaceChar (char ch) | Determines if the specified character is a Unicode space character. |
| boolean isUpperCase (char ch) | Determines if the specified character is an uppercase character. |
| boolean isWhitespace (char ch) | Determines if the specified character is white space according to Java. |
| static char toLowerCase (char ch) | Converts the character argument to lowercase. |
| String toString() | Returns a String object representing this Character's value. |
| static String toString(char c) | Returns a String object representing the specified char. |
| static char toUpperCase (char ch) | Converts the character argument to uppercase. |
| static Character valueOf1 (char ch) | Returns the Character object equivalent to ch. |

Table 19.14 Methods of Character class

Some of the methods are used programmatically as follows:

```
/*PROG 19.35 DEMO FO FEW METHODS OF CHARACTER CLASS VER 1 */

class JPS35
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        String s = "Life 143$#";
        for(int i =0;i<s.length();i++)
        {
            if(Character.isDigit((s.charAt(i))))
                show(s.charAt(i)+" is a digit.");
        }
    }
}
```

```

        if(Character.isLetter((s.charAt(i))))
            show((s.charAt(i))+" is a letter.");

        if(Character.isWhitespace(s.charAt(i)))
            show(s.charAt(i)+" is whitesapce.");

        if(Character.isUpperCase((s.charAt(i))))
            show(s.charAt(i)+" is uppercase.");
        if(Character.isLowerCase(s.charAt(i)))
            show(s.charAt(i)+" is lowercase.");
    }
}
}

OUTPUT:
L is a letter.
L is uppercase.
i is a letter.
i is lowercase.
f is a letter.
f is lowercase.
e is a letter.
e is lowercase.
    is whitesapce.
1 is a digit.
4 is a digit.
3 is a digit.

```

Explanation: In this program, a String is taken that consists of few small and upper case letters as well as some digits. Using for loop each character of the string is checked for letter, digit, uppercase, lowercase and so on using methods of Character class.

```

/*PROG 19.36 CASE CONVERSION USING METHODS OF CHARACTER CLASS */

import java.util.*;
import java.io.*;
class JPS36
{
    public static void main(String[] args) throws IOException
    {
        InputStreamReader isr;
        isr = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(isr);
        System.out.println("Enter a string");
        String str = bf.readLine();
        System.out.println("Reversed case string is");
        for (int i = 0; i < str.length(); i++)
    {

```

```

        if (Character.isLowerCase(str.charAt(i)))
            System.out.print(Character.toUpperCase(str.
                charAt(i)));
        else if (Character.isUpperCase(str.charAt(i)))
            System.out.print(Character.toLowerCase(str.
                charAt(i)));
        else
            System.out.print(str.charAt(i));
    }
}

OUTPUT:

*****
Enter a string
*****
MPSTME NMIMS UNIVERSITY MUMBAI
*****
Reversed case string is
*****
mpstme nmims university mumbai
*****

```

Explanation: In this program, a string is taken from the user and its characters analysed. Characters that are in uppercase are converted to lowercase and vice versa. Characters other than alphabets are printed as is.

```
/*PROG 19.37 DEMO OF FEW METHODS OF CHARACTER CLASS ver 2*/
```

```

import java.util.*;
import java.io.*;
class JPS37
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args) throws IOException
    {
        show("\nCharacter.forName(10,16):=
            "+Character.forName(10,16));
        Character ch1 = new Character('P');
        show("ch1 := "+ch1.charValue());
        show("Character.digit(A,16):=
            "+Character.digit('A',16));
        Character ch2 = Character.valueOf(ch1.charValue());
        System.out.print("ch1.compareTo(ch2):=");
        show((ch1.compareTo(ch2)==0)? "true": "false");
    }
}

```

OUTPUT:

```
Character.forDigit(10,16) := a
ch1 := P
Character.digit(A,16) :=10
ch1.compareTo(ch2) :=true
```

Explanation: The `forDigit` method returns the character equivalent of the first argument in radix that is specified by second argument. In order to print the character equivalent of digit 10 in hexadecimal number system 16, the first argument is supplied as 10 and second method takes two parameters: first one is the character and second one the radix. The method returns the integer equivalent of the character supplied. The `valueOf()` method takes a character value and returns the equivalent `Character` object. The `compareTo` methods two `Character` objects and returns 0 if both are equal.

19.12.6 The Boolean Class

The `Boolean` class wraps a value of the primitive type `boolean` in an object. An object of type `Boolean` contains a single field whose type is `boolean`. The class does not have many methods and is less frequently used. Constructor of this class is shown below:

1. **`Boolean(boolean value)`**

This form of constructor allocates a `Boolean` object representing the value argument. This form should be used when new `Boolean` instance is required: one should rather rely on `valueOf` method.

2. **`Boolean(String s)`**

This form of constructor allocates a `Boolean` object representing the value true if the string argument is not null and is equal, ignoring case, to the string ‘true’. Otherwise, allocate a `Boolean` object representing the value false.

Example:

- (a) `new Boolean("True")` produces a `Boolean` object that represents true.
- (b) `new Boolean("yes")` produces a `Boolean` object that represents false.

The class contains three static constants:

| | |
|---|---|
| <code>static final Boolean FALSE</code> | The <code>Boolean</code> object corresponding to the primitive value false. |
| <code>static final boolean TRUE</code> | The <code>Boolean</code> object corresponding to the primitive value true. |
| <code>static final Class TYPE</code> | The <code>Class</code> objects representing the primitive type <code>boolean</code> . |

The methods of `Boolean` class are given in Table 19.15.

| Method Signature | Description |
|--|--|
| <code>boolean booleanValue()</code> | Returns the value of this <code>Boolean</code> object as a <code>boolean</code> primitive. |
| <code>int compareTo(Boolean b)</code> | Compares this <code>Boolean</code> instance with another. |
| <code>static boolean parseBoolean(String s)</code> | Parses the string argument as a <code>boolean</code> . The <code>boolean</code> returned represents the value true if the string argument is not null and is equal, ignoring case, to the string ‘true’. |
| <code>String toString()</code> | Returns a <code>String</code> object representing this <code>Boolean</code> ’s value. |

(Continued)

| | |
|--|--|
| <code>static String
toString(boolean b)</code> | Returns a String object representing the specified boolean value. |
| <code>static Boolean
valueOf(boolean b)</code> | Returns a Boolean instance representing the specified Boolean value. |
| <code>static Boolean
valueOf(String s)</code> | Returns a Boolean with a value represented by the specified String. |

Table 19.15 Methods of Boolean class

```
/*PROG 19.38 DISPLAYING BOOLEAN CONSTANTS */

import java.util.*;
import java.io.*;
class JPS38
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        show("\nBoolean.TRUE :=" + Boolean.TRUE);
        show("\nBoolean.FALSE:=" + Boolean.FALSE);
        show("\nBoolean.TYPE :=" + Boolean.TYPE);
    }
}
OUTPUT:
Boolean.TRUE :=true
Boolean.FALSE:=false
Boolean.TYPE :=boolean
```

Explanation: This program is self-explanatory.

```
/*PROG 19.39 USING SOME METHODS OF BOOLEAN CLASS */

import java.io.*;
import java.util.*;
class JPS39
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        Boolean b1 = Boolean.parseBoolean("Hello");
        Boolean b2 = Boolean.parseBoolean("True");
        show("\nb1:= " + b1 + "\tb2 := " + b2);
    }
}
```

```

        boolean bool1 = b1.booleanValue();
        Boolean b3 = Boolean.valueOf(bool1);
        System.out.print("\nb1.compareTo(b3) :=");
        show(b1.compareTo(b3) == 0 ? "true" : "false");
    }
}

OUTPUT:
b1:= false b2:= true
b1.compareTo(b3) :=true

```

Explanation: The b1 is a Boolean object representing false value as string is other than ‘true’ regardless of case of characters. Boolean object represents true. The boolean variable bool1 contains false value as it is extracted using booleanValue() method from b1. Using this bool1 as argument to valueOf method, a new Boolean object b3 is constructed. Now comparing b1 and b3 returns 0 as both are equal.

19.13 PONDERABLE POINTS

1. The StringTokenizer class allows an application to break a string into tokens. It is defined within the package java.util. StringTokenizer implements the Enumeration interface. Therefore, given an input string, the individual tokens contained in it can be enumerated using StringTokenizer.
2. The Date class encapsulates the current date and time. It is defined within the java.util package.
3. The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, etc. The class is defined within java.util package.
4. An instance of Random class is used to generate a stream of pseudo-random numbers. These numbers are called so, as they are uniformly distributed over a range of numbers.
5. The Observable class represents an observable object. It can be subclassed to represent an object that the application wants to have observed. When an object of such a subclass undergoes a change, observing classes are notified. The observing classes must implement Observer interface. A class can implement the Observer interface when it wants to be informed of changes in observable objects.
6. The System class defined in package java.lang contains several useful class fields and methods. It cannot be instantiated as it is a final class. Among the facilities provided by the System class are standard input, standard output and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility methods for quickly copying a portion of an array.
7. Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
8. The clone() method of Object class generates a duplicate copy of the object on which it is called. The method creates and returns a copy of this object.
9. The Class class encapsulates the runtime state of an object or interface. Instance of the class Class represents classes and interfaces in a running Java application. The primitive Java type (boolean, byte, char, short, int, long, float and double), and the keyword void are also represented as Class objects.
10. The class Runtime is an abstraction of runtime environment. Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.

11. The Process class is an abstraction of the runtime program. The Runtime.exec methods create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it.
12. Wrapper classes represent the primitive values as objects. A primitive wrapper class in the Java programming language is one of eight classes provided in the java.lang package which provide object methods for the eight primitive types: byte, short, long, int, float, char, boolean and double.

REVIEW QUESTIONS

1. Write a program for matrix manipulation using array copy.
2. Write short notes on System class and Class class.
3. Write a program to display all the system properties.
4. Write a program for linked list processing.
5. Write a program for stack operations.
6. List any five methods available in Math package. Explain its uses.
7. Write a program to find the difference between two dates.
8. Write a program to insert the elements in Hash table and display it.
9. Write short notes on StringTokenizer and Properties class.
10. Write short notes on:
 - (a) Object class
 - (b) Random class
11. Why date class is deprecated?
12. What will be the output of the following program?

```
class test
{
    public static void main(String args[])
    {
        System.out.println(Math.floor(2.7));
        System.out.println(Math.ceil(2.7));
        System.out.println(Math.round(2.7));
        System.out.println(Math.round(2.3));
        System.out.println(Math.rint(2.7));
        System.out.println(Math.ceil(-2.2));
        System.out.println(Math.floor(-2.2));
        System.out.println(Math.round(-2.7));
        System.out.println(Math.max(2, 3));
        System.out.println(Math.max(-2, -3));
        System.out.println(Math.min(2, 3));
        System.out.println(Math.min(-2, -3));
    }
}
```

13. Which class is the root class for all the Java class? Justify.
14. Write a program to run the following applications:
 - (a) System calculator – (C:\windows\calc.exe)
 - (b) Paint brush – (C:\Program Files\Accessories\MSPAINT.EXE)
 - (c) Internet Explorer – (C:\Program Files\Internet Explorer\IEXPLORE.EXE)
15. Explain the terms Observable class and Observable interface with suitable example.
16. Explain the role of Environmental Variables.
17. Why do we use clone method in Java? Explain with example.
18. Why do we use Process class?
19. Write short note on Wrapper class with example.

Multiple Choice Questions

1. The _____ class allows an application to break into tokens.
 - (a) StringTokenizer (c) StringSplit
 - (b) StringBreak (d) None of the above
 2. System class is defined in
 - (a) java.Applet (c) java.lang
 - (b) java.awt (d) java.util
 3. The _____ method creates a native process and return an instance of a subclass of Process class that can be used to control the process and obtain information about it.
 - (a) Runtime method
 - (b) Runtime.exec method
 - (c) Runtime.exe method
 - (d) System.exec method
 4. A primitive wrapper class in the Java programming language is one of the eight classes provided in the
 - (a) java.awt (c) java.util
 - (b) java.applet (d) java.lang
 5. The current runtime can be obtained from the
 - (a) runtime method
 - (b) getRuntime method
 - (c) getRuntime.exec method
 - (d) None of the above
6. Trace the following code:
- ```

import java.util.*;
import java.io.*;
class TEST
{
 static void show(String s)
 {
 System.out.println(s);
 }
 public static void main(String[] args)
 {
 show("Boolean.TRUE = " + Boolean.TRUE);
 show("Boolean.FALSE = " + Boolean.FALSE);
 show("Boolean.TYPE = " + Boolean.TYPE);
 }
}

```
- (a) Boolean.TRUE = TRUE  
Boolean.FALSE = FALSE  
Boolean.FALSE = TYPE
  - (b) Boolean.TRUE = true  
Boolean.FALSE = false  
Boolean.TYPE = type
  - (c) Boolean.TRUE = true  
Boolean.FALSE = false  
Boolean.TYPE = Boolean
  - (d) Boolean.TRUE = true  
Boolean.FALSE = false  
Boolean.TYPE = Boolean
7. Trace the following code:
- ```

import java.util.*;
import java.io.*;
class TEST
{
    static void show(String s)
    {

```

```

        System.out.println(s);
    }
    public static void main(String[] args)
    {
        boolean b1 = new Double (35 / 0.).isInfinite();
        boolean b2 = Double. isInfinite(35 / 0.);
        show("35/0 is infinite: " + (b1 && b2));
        b1 = new Double (0 / 0.).isNaN();
        b2 = Double.isNaN (0 / 0.);
        Show("0/0 is Not a Number : " + (b1 && b2));
    }
}

```

- (a) 35/0 is infinite: true
0/0 is Not a Number: true
- (b) 35/0 is infinite: true
0/0 is Not a Number: false
- (c) Undefined (Error Message)
- (d) None of the above
8. Which of the following methods returns a string representation of the integer argument as an unsigned integer in base 2?
- (a) Static String toBinaryString(int i)
(b) Public static String toBinaryString(int i)
(c) Public void String toBinaryString(int i)
(d) None of the above
9. The method used to return the value of the specified number as a byte is
- (a) void byteValue(int i)
(b) public void byteValue(int i)
(c) byte byteValue()
(d) None of the above
10. The method used to initiate garbage collection is
- (a) String garbage()
(b) void initgarbage()
(c) void gc()
(d) void initgc()

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. c 3. c 4. d 5. b 6. c 7. a 8. a 9. c 10. c

Networking in Java

20

20.1 HOW DO COMPUTERS TALK TO EACH OTHER VIA INTERNET?

The Internet is composed of millions of computers, located across the globe, communicating and transmitting information over a variety of computing systems, platforms and networking equipment. Before communicating with another party, one must first know how to address the message so they can be delivered correctly. On IP networks, the addressing scheme in use is based on hosts and port numbers. Each of these computers (if they are connected via an Internet) will have a unique IP address.

A given host computer on an IP networking has a hostname and a numeric address. Either of these, in their fully qualified forms, is a unique identifier for a host on the network. The JavaSoft home page, for example, resides on a host named *www.javasoft.com*, which currently has the IP address 204.160.241.98. Either of these addresses can be used to locate the machine on an IP network. IP addresses are 32-bit numbers, containing four octets (8-bit numbers) separated by a full stop. Each computer with a direct Internet connection will have a unique IP address (e.g., 207.68.156.61). Some computers have temporary addresses, such as when the application connects to the ISP through a modem. Others have permanent address, and some even have their own unique domain name (e.g., <http://www.rediff.com>). The textual name for the machine is called its Domain Name Service (DNS), which can be considered as a kind of alias for the numeric IP address.

An IP address allows one to identify a particular device or system connected to the Internet. If an application wants to connect to a specific IP address, and send a message, it can do so. Without an IP address, the message to be sent would have no way of reaching its destination—somewhat like leaving the address off a letter or parcel.

Often, computers connected to the Internet provide services. This page is provided by a Web server, for example. Because computers are capable of providing more than one type of service, a way is needed to uniquely identify each service. Like an IP address, a number is used. This number is called a port. Common services (e.g., HTTP, FTP, Telnet and SMTP) have well-known port numbers. For example, most Web servers use port 80. Of course, one can use any port he likes—there is no rule that says one must use 80.

There are several communications mechanisms that can be used to provide network services. UDP (Unreliable Datagram Protocol), or TCP (Transfer-Control Protocol) could be used. But emphasis will be given on TCP. TCP guarantees that messages will arrive at their destination. UDP is unreliable, and the application is not notified if the message is lost in transit. Also, many protocols (e.g., HTTP, SMTP, POP and FTP) use TCP, so it is important to be familiar with it for networking in Java.

20.2 JAVA.NET PACKAGE

Suppose networking in Java is provided by the `java.net` package. The package provides the classes for implementing networking applications. The `java.net` package can be roughly divided into two sections:

1. **A low level API**, which deals with the following abstractions:
 - **Addresses**, which are networking identifiers, like IP addresses.
 - **Socket**, which is basic bidirectional data communication mechanism.
 - **Interface**, which describes network interface.
2. **A high level API**, which deals with the following abstractions:
 - **URIs**, which represent Universal Resource Identifiers.
 - **URLs**, which represent Universal Resource Locators.
 - **Connections**, which represent connections to the resource pointed to by URLs.

This chapter covers addresses, sockets and URLs.

20.3 INTERNET ADDRESSING WITH JAVA

Handling Internet addresses (domain names and IP addresses) is made easy with Java. Internet addresses are represented in Java by the `InetAddress` class. `InetAddress` provides simple methods to convert between domain names and numbered addresses. This class represents an Internet Protocol (IP) address. An IP address is either a 32-bit or 128-bit unsigned number used by IP, a lower-level protocol on which protocols like UDP and TCP are built. An instance of an `InetAddress` consists of an IP address and possibly its corresponding host name. The application interacts with this class by using the name of an IP host, which is more convenient and understandable than its IP address.

The class provides no constructor, instead it provides a number of static methods for getting host name or address string. The two commonly used methods of this class return instance of the `InetAddress` class. Other methods will be discussed later.

1. **`public static InetAddress getLocalHost() throws UnknownHostException`**
This method returns the IP address of the local host.
2. **`public static InetAddress getByName(String host) throws UnknownHostException`**

This method determines the IP address of a host, given the host's name. The host name can either be a machine name, such as 'java.sun.com', or a textual representation of its IP address. If a literal IP address is supplied, only the validity of the address format is checked.

The other methods of `InetAddress` class are shown in the Table 20.1.

| Method Signature | Description |
|--|--|
| <code>byte[] getAddress()</code> | Returns the raw IP address of this <code>InetAddress</code> object. The result in network byte order: the highest order byte of the address is in <code>getAddress () [0]</code> . |
| <code>String getHostAddress()</code> | Returns the IP address string in textual presentation. |
| <code>String getHostName()</code> | Gets the host name for this IP address. |
| <code>String getCanonicalHostName()</code> | Gets the fully qualified domain name for this IP address. |
| <code>String toString()</code> | Converts this IP address to a String. |

Table 20.1 Few methods of `InetAddress` class

```
/*PROG 20.1 DISPLAY IP ADDRESS AND LOCAL HOST */

import java.net.*;
public class JPS1
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        try
        {
            InetAddress local = InetAddress.
                getLocalHost();
            show("\nLocal IP Address : " + local);
            show("\nLocal hostname : " + local.
                getHostName());
        }
        catch (UnknownHostException e)
        {
            show("Can't detect localhost : " + e);
        }
    }
}

OUTPUT:

Local IP Address : harics/10.0.4.170
Local hostname : harics
```

Explanation: The application starts by importing the java.net package, which contains a set of pre-written networking routines (including InetAddress).

```
import java.net.*;
```

Next, a new variable of type InetAddress is declared, which is assigned the value of the local host machine (for machines not connected to a network, this should represent 10.0.4.170). Due to the fact that InetAddresses can generate exceptions, this code must be placed between a try..... catch 'UnknownHostException' block.

The following line obtains the InetAddress of the computer on which this program is running.

```
InetAddress local = InetAddress.getLocalHost();
```

The InetAddress can be printed out, as well as the domain name of the local address.

```
show("\nLocal IP Address : " + local);
show("\nLocal hostname : " + local.getHostName());
```

The output is as shown above.

```
/*PROG 20.2 DEMONSTRATING FEW METHODS OF INETADDRESS CLASS */
```

```
import java.net.*;
class JPS2
{
```

```

        static void show(String s){
            System.out.print(s);
        }
        public static void main(String args[])
        {
            int i;
            try
            {
                InetAddress local=InetAddress.
                getByName("127.0.0.1");
                byte[] arr = new byte[4];
                arr = local.getAddress();
                show("\nIP address using getAddress method \n");
                for (i = 0; i < arr.length - 1; i++)
                show(arr[i] + ".");
                show(arr[i] + "\n");
                show("\nIP address using getHostAddress method\n");
                String ip = local.getHostAddress();
                show(ip + "\n");
                show("\nHost name using getCanonicalHostName
method \n");
                ip = local.getCanonicalHostName();
                show(ip + "\n");
            }
            catch (UnknownHostException e)
            {
                System.out.println("Can't detect localhost:" + e);
            }
        }
    }
}

```

OUTPUT:

```

IP address using getAddress method
127.0.0.1
IP address using getHostAddress method
127.0.0.1
Host name using getCanonicalHostName method
localhost

```

Explanation: This program demonstrates various methods of InetAddress class. In the beginning, using `getByName` an instance of InetAddress is obtained in `local`. The static method `getByName` takes the IP address in string form. Here we are testing the program on to the local machine that is not connected to the Internet, so loopback address 127.0.0.1 has been written. Instead of this, ‘localhost’ can also be written as argument to the method `getByName`.

The method `getAddress` returns the raw IP address in a byte array. The highest order byte of the address is in `arr[0]`. Each individual byte of this array `arr` is displayed using the `for` loop. To construct the complete IP address, dot in string form “.” is appended.

The method `getHostAddress` returns the numeric IP address in String form. The method `getCanonicalHostName` returns the complete domain host name in string form.

20.4 SOCKET FUNDAMENTALS

The most common definition of socket is, ‘A socket is one endpoint of a two-way communication link between two programs running on the network’. Put it differently, it is through sockets that applications access the network and transmit data. The types of sockets are as varied as the purposes and platforms of applications. There are three types of sockets: Unix domain sockets, Internet domain sockets and NS domain sockets.

Of these, only Internet domain sockets are supported across all platforms. So to maintain the cross-platform characteristics intact, Java supports only Internet domain sockets. The next question that arises is, what are the characteristics of an Internet domain socket and what protocols are supported by it? The following section gives the answer.

20.4.1 Internet Domain Sockets

By definition ‘An Internet socket (commonly known as a socket or network socket) is a communication end-point unique to a machine communicating on an Internet protocol-based network, such as the Internet’. All applications communicating through the Internet use a network socket. The feature that distinguishes a network socket from other sockets is the protocols that it supports. The supported protocols are TCP, UDP and Raw IP.

The difference between them is based on whether the protocol is connection oriented or not. The supported protocols are discussed in detail in this section.

TCP is one of the core protocols of the Internet protocol suite. The protocol guarantees reliable and in-order (correct order of packets) delivery of data from sender to receiver. To put it simple, it is reliable. The second aspect of TCP is that it is connection-oriented. That means TCP requires that a connection be made between the sender and receiver before data is sent. If a packet expected at the receiving end of a TCP socket does not arrive in a set period of time, it is assumed lost, and the packet is required from the sender again. The receiver does not move on to the next packet until the first is received. TCP sockets are used in the large majority of IP applications. The socket associated with TCP is known as the stream socket.

UDP, like TCP is one of the core protocols of the IP suit. However, unlike TCP, it neither guarantees in-order delivery of data nor does it require a connection to be established for sending the data. The sender transmits a UDP packet, and it either makes it to the receiver or it does not. To put it simply, UDP is an unreliable and connectionless protocol. UDP sockets are typically used in bandwidth limited applications, where the overhead associated with resending packets is not tolerable. A good example of this real-time network is audio applications. If packets of audio information are delivered over the network to be played in real time, there is no point in resending a late packet. By the time it gets delivered it will be useless, since the audio track must play continuously and sequentially, without backtracking. Sockets associated with UDP are known as datagram sockets.

Raw IP is a non-formatted protocol, unlike TCP and UDP. It works at network and transport layers. A socket associated with raw IP is known as raw socket. UDP and TCP sockets just receive the payload or the data, whereas raw sockets receive the header info of the packet along with the data. The downside of raw sockets is that they are tightly coupled with the implementation provided by the underlying host operating system.

Next it can be discussed how Java places the different types of sockets in its libraries.

20.5 SOCKETS IN JAVA

At the core of Java’s networking support are the `Socket`, `DatagramSocket` and `ServerSocket` classes in `java.net`. These classes define channels for communication between processes over an IP network. Java abstracts out most of the low-level aspects of socket programming using these classes. A new socket is created by specifying a host, either by name or with an `InetAddress` object, and a port number on the host.

The `ServerSocket` class provides server sockets or sockets at server side. Such sockets wait for requests over the network. Once such requests arrive, a server socket performs operations based on the request and may return a result. The `ServerSocket` class wraps most of the options required to create server-side sockets.

The `Socket` class provides client side sockets or simply sockets. They are at the client side connecting to the server, sending the request to the server and accepting the returned result. Just as `ServerSocket` exposes only the necessary one required to create a server-side socket, similarly, `Socket` asks the user to provide only those parameters that are most necessary.

The `DatagramSocket` class represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

20.5.1 Step-by-step Socket Application Development

Any network-enabled application has two important parts: the code that executes at client side and the code that executes server side. So using the functionality of sockets can be grouped into two major steps: the server or the server-side code and the client or the client-side code.

The multi-threaded nature of the former can always be guaranteed whereas the latter may or may not be multi-threaded.

20.5.2 The Server

The server's main function is to wait for incoming request, and to service them when they come in. So the code to implement the server can be further broken down into the following steps:

1. Establish a server that monitors a particular port. This is done by creating an instance of the `ServerSocket` class. There are four different ways to create an instance of `ServerSocket`. They are

- **`ServerSocket()`**

This form of constructor simply sets the implementation that means everything is taken as default values:

- **`ServerSocket(int port)`**

This form of constructor creates a server-side socket and binds the socket to the given port number.

- **`ServerSocket(int port, int backlog)`**

This form of constructor not only binds the created socket to the port but also creates a queue of length specified by the number passed as the backlog parameter.

- **`ServerSocket(int port,int backlog,InetAddress bindAddr)`**

This form of constructor creates a server-side socket that is bound to the specified port number with the queue of length specified by the backlog and bound to the address specified by the bindAddr argument.

So to create a socket bound to port number **8328** with a backlog queue of size **5** and bound with address of local host the statement would be:

```
ServerSocket server;
server=new ServerSocket(8328,5,InetAddress.getLocalHost());
```

One point to keep in mind is that the above mentioned constructors return TCP sockets and not UDP sockets.

- The next step is to tell the newly created server socket to listen indefinitely and accept incoming requests. This is done by using the `accept()` method of `ServerSocket` class. When a request comes, `accept()` returns a `Socket` object representing the connection. The `accept()` method call is a blocking call that will wait for a client to initiate communications, and then returns with a normal `Socket` that is then used for communication with the client.

It can be written in the following code:

```
Socket incoming = server.accept();
```

- Communicating with the socket means reading from and writing to the `Socket` object. To communicate with a `Socket` object, two tasks have to be performed. First the input and output stream corresponding to the `Socket` object has to be obtained. That can be done by using the `getInputStream()` and `getOutputStream()` methods of `Socket` class.

This can be coded as:

```
InputStream sin = incoming.getInputStream();
InputStreamReader isr = new InputStreamReader(sin);
BufferedReader in = new BufferedReader(isr);
OutputStream sout = incoming.getOutputStream();
PrintWriter pout = new PrintWriter(sout, true);
```

The second task is to read from and write to the `Socket` object. Since the communication has to continue until the client breaks the connection, the reading from and writing to are done within a loop like this:

```
boolean done = false;
while(!done)
{
    String line = in.readLine();
    if(line == null) done = true;
    else
    {
        out.println("Echo: "+line);
        if(line.trim().equals("BYE"))
            done = true;
    }
}
```

The actual syntax for reading and writing is not different from the I/O done for simple files.

- Once the client breaks the connection or stops sending the request, the `Socket` object representing the client has to be closed. This can be done by calling `close()` method on the `Socket` object. The statement would be:

```
incoming.close();
```

The actual syntax for reading and writing is not different from the I/O done for simple files.

20.5.3 The Client

The main purpose of the client is to connect to the server and communicate with it using the connection. So coding a client requires the following steps:

- Connect to the server.* Connecting to the server can be accomplished in two steps:
 - *Creating a Socket object.* The socket at client side just needs to know the host name (the name of the machine where the server is running) and the port where the server is listening. To create a `Socket` object, there are seven constructors provided by the `Socket` class, of which the most commonly used are:

a. `Socket()`

This form of constructor creates a new Socket instance without connecting to host.

b. `Socket(InetAddress address, int port)`

This form of constructor creates a new Socket object and connects to the port specified at the given address.

c. `Socket(java.lang.String host, int port)`

This form of constructor works the same way as `Socket()`, except that instead of an address, the host name is used.

So to create a Socket object that connects to ‘localhost’ at 8328, the statement would be:

```
Socket s = new Socket ("localhost", 8888);
```

- Connecting to the server comes into picture if no argument constructor is used. It takes the object of the `SocketAddress` object as an argument. So to connect to localhost at port 8328, the code would be:

```
Socket s = new Socket();  
s.connect(new SocketAddress("localhost", 8328));
```

The different code for the same purpose may be:

```
Socket s = new Socket("localhost", 8328);
```

2. *Communicating with the server:* Communicating with the server using a socket at client side is no different from how the server communicates with the client. First, the input and output streams connected with the `Socket` object are to be retrieved thus:

```
InputStream sin = incoming.getInputStream();  
InputStreamReader isr = new InputStreamReader(sin);  
BufferedReader in = new BufferedReader(isr);  
OutputStream sout = incming.getOutputStream();  
PrintWriter pout = new PrintWriter(sout, true);
```

Then one can read and write using the corresponding streams. For example, if the client just waits for the data sent by the server, the code would be:

```
boolean more = true;  
while(more)  
{  
    String line = in.readLine();  
    if(line == null)  
        more = false;  
    else  
        System.out.println(line);  
}
```

That covers all the steps involved in creating a network-enabled application. When running both server and client onto the same machine, the server and client both must be run in separate window and server must start first.

Now few programs are presented on the basis of the concept built so far. All programs will be in form of pairs of client and server.

| |
|--|
| /*PROG 20.3 DAY TIME CLIENT, FILE client.java */ |
|--|

```
import java.net.*;  
import java.io.*;
```

```
import java.util.Date;
class Client
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        int port = 1500;
        String server = "localhost";
        Socket socket = null;
        BufferedReader input;
        String msg = null;
        try
        {
            socket = new Socket(server, port);
            show("Connected with server ");
        }
        catch (Exception e)
        {
            show(e + " ");
            System.exit(1);
        }
        try
        {
            InputStream sin = socket.getInputStream();
            InputStreamReader isr=new InputStreamReader(sin);
            input = new BufferedReader(isr);
            msg = input.readLine();
            show("Current Date and Time received by server");
            show(msg);
        }
        catch (Exception e)
        {
            show(e + " ");
        }
        try
        {
            socket.close();
        }
        catch (IOException e)
        {
            show(e + " ");
        }
    }
}
```

```
/*PROG 20.4 DAY TIME SERVER, FILE server.java */  
  
import java.net.*;  
import java.io.*;  
import java.util.Date;  
class Server  
{  
    static void show(String s)  
    {  
        System.out.println(s);  
    }  
    public static void main(String args[])  
    {  
        int port;  
        ServerSocket server_socket;  
        PrintWriter output;  
        String msg = null;  
        Socket socket = null;  
        try  
        {  
            server_socket = new ServerSocket(1500);  
            show("Server waiting for client ");  
            while(true)  
            {  
                socket = server_socket.accept();  
                OutputStream sout = socket.getOutputStream();  
                output = new PrintWriter(sout, true);  
                show("New connection accepted ");  
                show("Sending current date and time ");  
                output.println(new Date()+" ");  
                break;  
            } //while ends  
        } //try ends  
        catch(Exception e)  
        {  
            show(e+" ");  
        }  
        try  
        {  
            socket.close();  
        }  
        catch(IOException e)  
        {  
            show(e+" ");  
        }  
    }  
}
```

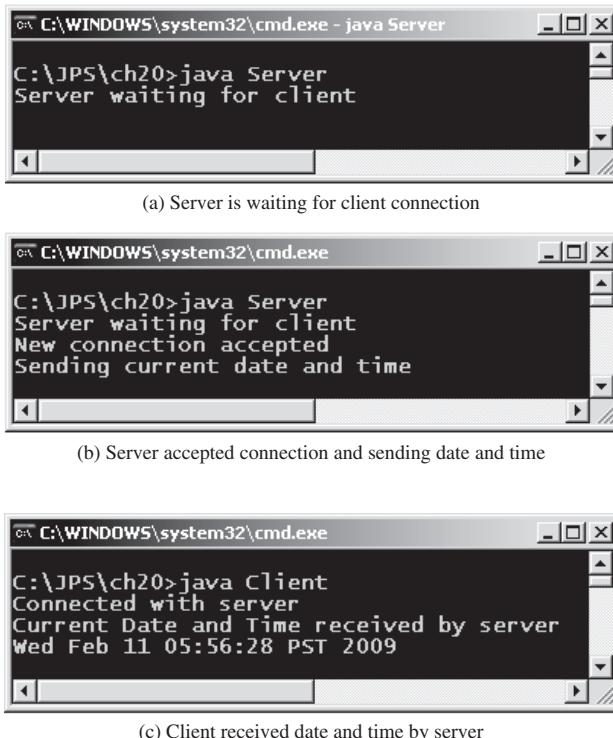


Figure 20.1 Output screen of Programs 20.3 and 20.4

Explanation: The server sends the current date and time using `new Date()` to the client. The client reads and displays the same onto the console. In the server code, an instance `server_socket` of class `ServerSocket` is created. The server listens onto the port number 1500. In the while loop, the `accept` method of `ServerSocket` class returns true when a client connects to the server at port number 1500. The method returns a new socket of type `Socket` (i.e., `socket` here represents client in the server program). The output stream for this socket is retrieved using `getOutputStream` method and stored in `sout`, an instance of `OutputStream` type. This `sout` is passed to the constructor of `PrintWriter` class that creates an instance `output` of this class. After that following line are displayed on to the server console window:

```
New Connection accepted
Sending current date and time
```

The next line

```
Output.println(new Date() + " ");
```

sends the output onto the client's socket which is current date and time. In other words, current date and time are written onto the client socket which is there onto the server side.

In the client code, a client side socket is created with host name as 'localhost' and port number 1500. In the try block, the message (current date and time) sent by server is read using input stream of the socket created earlier. For this purpose, classes `BufferedReader`, `InputStreamReader` and `InputStream` are used. The input stream for the socket is obtained using the method `getInputStream`. The message sent by server is read and displayed as:

```
msg = input.readLine();
show("Current Date and Time received by server");
show(msg);
```

```
/*PROG 20.5 ECHO CLIENT FILE ECHOCLIENT.JAVA */

import java.net.*;
import java.io.*;
import java.util.Date;
class EchoClient
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        int port = 1500;
        String server;
        Socket socket = null;
        BufferedReader input_sock = null, input_con = null;
        PrintWriter output = null;
        String msg = null;
        if (args.length != 1)
        {
            show("Usage:EchoClient <IP address>");
            System.exit(0);
        }
        try
        {
            server = args[0];
            socket = new Socket(server, port);
            show("Connected with server ");
            output=new PrintWriter(socket.
                getOutputStream(),true);
            input_sock = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
            input_con = new BufferedReader(new
                InputStreamReader(System.in));
        }
        catch (Exception e)
        {
            show(e + " ");
            System.exit(1);
        }
        try
        {
            while (true)
            {
                msg = input_con.readLine();
                output.println(msg);
                msg = input_sock.readLine();
                if (msg == null)
                    System.exit(0);
            }
        }
    }
}
```

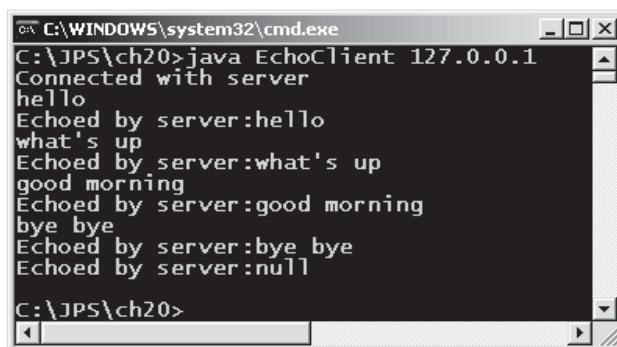
```
        show(msg);
    }
}
catch (Exception e)
{
    show(e + " ");
}
try
{
    socket.close();
}
catch (IOException e)
{
    show(e + " ");
}
}
}
```

```
/*PROG 20.6 ECHO SERVER, FILE ECHOSERVER.JAVA */
```

```
import java.net.*;
import java.io.*;
class EchoServer
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        int port;
        ServerSocket server_socket;
        PrintWriter output;
        BufferedReader input;
        String msg = null;
        Socket socket = null;
        try
        {
            server_socket = new ServerSocket(1500);
            show("Server waiting for client ");
            socket = server_socket.accept();
            show("New connection accepted ");
            while (true)
            {
                output=newPrintWriter(socket.getOutputStream(),true);
                InputStream sin = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(sin);
                input = new BufferedReader(isr);
                msg = input.readLine();
                if (msg.equals("Bye Bye"))

```

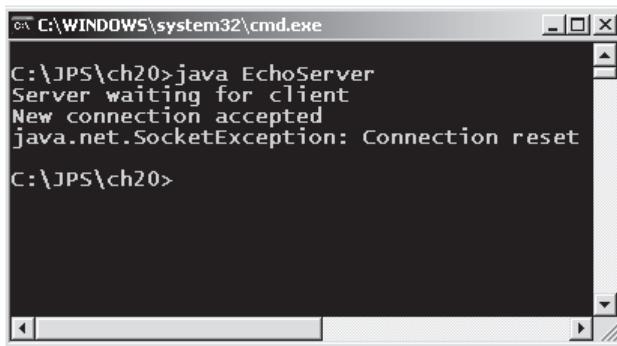
```
        {
            show("Connection terminated by client");
            break;
        }
    output.println("Echoed by server:" + msg);
} //while ends
}//try ends
catch (Exception e)
{
    show(e + " ");
}
try
{
    socket.close();
}
catch (IOException e)
{
    show(e + " ");
}
}
}
}
```



A screenshot of a Windows command prompt window titled 'cmd.exe'. The window shows the output of a Java application named 'EchoClient'. The client sends a series of messages to a server at '127.0.0.1': 'hello', 'what's up', 'good morning', 'bye bye', and 'null'. The server echos each message back to the client. The client also shows its own input line 'C:\JPS\ch20>'.

```
C:\WINDOWS\system32\cmd.exe
C:\JPS\ch20>java EchoClient 127.0.0.1
Connected with server
hello
Echoed by server:hello
what's up
Echoed by server:what's up
good morning
Echoed by server:good morning
bye bye
Echoed by server:bye bye
Echoed by server:null
C:\JPS\ch20>
```

(a) Client window



A screenshot of a Windows command prompt window titled 'cmd.exe'. The window shows the output of a Java application named 'EchoServer'. The server is waiting for a client connection. It receives a connection from a client and logs the message 'New connection accepted'. When the client disconnects, the server logs 'java.net.SocketException: Connection reset'. The server's input line 'C:\JPS\ch20>' is also visible.

```
C:\WINDOWS\system32\cmd.exe
C:\JPS\ch20>java EchoServer
Server waiting for client
New connection accepted
java.net.SocketException: Connection reset
C:\JPS\ch20>
```

(b) Server window

Figure 20.2 Output screen of Programs 20.5 and 20.6

Explanation: In this client server application, client reads some data from keyboard and sends it to server; server sends the same data back to the client; client reads back the data sent by the server and displays it onto the screen. In other words, data is echoed back.

In the client program, the host name is provided as command line argument. After the socket is created with specified hostname and port number 1500, two input streams are created, one by the name **input_con** for reading from console (i.e., keyboard) and other for reading from the socket by name **input_sock**. An output stream is created by the name **output** for writing to the socket. In the while loop, the string is read from the console and stored in **msg**. The string **msg** is sent to the server.

In the server program, the listening socket **server_socket** listens at port number 1500. New connection from the client is accepted using the accept method and a new connecting socket ‘socket’ is created. Input stream for this connecting socket is created by the name **sin** and output stream is created by the name **output**. The server reads whatever was sent by the client using the following line:

```
msg = input.readLine();
```

If **msg** is equal to ‘Bye Bye’ connection is terminated else the same string **msg** is sent back to the client as:

```
output.println("Echoed by server:" + msg);
```

This message string when sent back to the client, is read by him as:

```
msg = input_sock.readLine();
```

If the **msg** is null, connection is terminated:

```
if(msg == null)
    System.exit(0);
```

Otherwise, the message is shown as follows:

```
show(msg);
```

```
/*PROG 20.7 QUOTATION CLIENT, FILE QUOTATIONCLIENT.JAVA */
```

```
import java.net.*;
import java.io.*;
import java.util.Date;
class QuoteClient
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[]args)
    {
        int port = 1500;
        String server;
        Socket socket = null;
        BufferedReader input_sock = null;
        String msg = null;
        if(args.length!=1)
        {
            show("Usage: QuoteClient<IP Address>");
            System.exit(0);
        }
    }
}
```

```

        }
        try
        {
            server = args[0];
            socket = new Socket(server, port);
            show("Connected with server");
            input_sock = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
        }
        catch (Exception e)
        {
            show(e+" ");
            System.exit(1);
        }
        try
        {
            while (true)
            {
                msg = input_sock.readLine();
                show("/**Quotation got from
                      server**/");
                show("\\" + msg + "\\");
                break;
            }
        }
        catch (Exception e)
        {
            show(e + " ");
        }
        try
        {
            socket.close();
        }
        catch (IOException e)
        {
            show(e+" ");
        }
    }
}

```

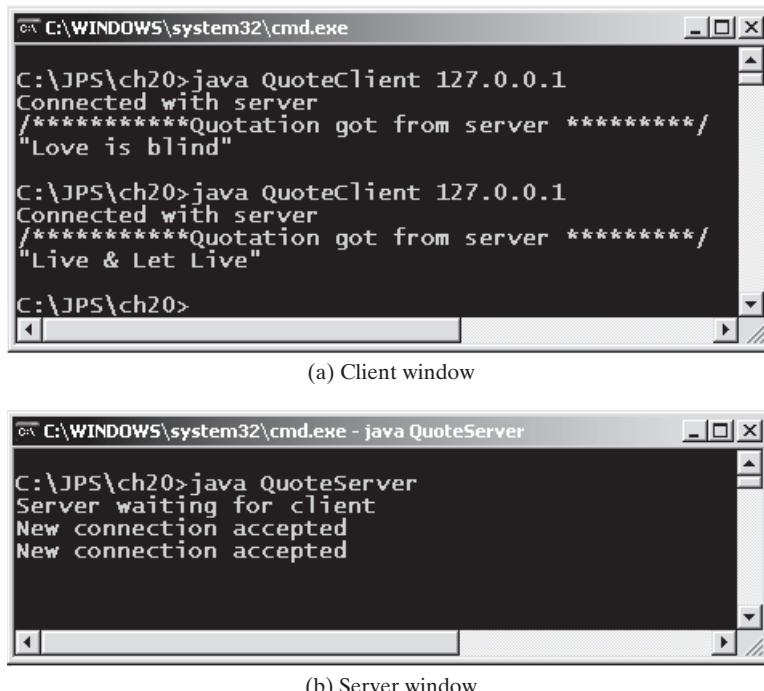
/*PROG 20.8 QUOTATION SERVER, FILE QUOTESERVER.JAVA */

```

import java.net.*;
import java.io.*;
import java.util.*;
class QuoteServer
{
    static void show(String s)

```

```
{  
    System.out.println(s);  
}  
public static void main(String args[])  
{  
    String str[] = {"Live & Let Live",  
                    "Honesty is the best policy",  
                    "Strike the iron when it is hot",  
                    "Love is blind",  
                    "Work is worship"};  
    int port;  
    ServerSocket server_socket;  
    PrintWriter output;  
    BufferedReader input;  
    String msg = null;  
    Socket socket = null;  
    try  
{  
        server_socket = new ServerSocket(1500);  
        show("Server waiting for client");  
        while (true)  
        {  
            socket = server_socket.accept();  
            show("New connection accepted");  
            output = new  
                PrintWriter(socket.getOutputStream(), true);  
            Random rnd = new Random(new  
                Date().getTime());  
            int x = Math.abs(rnd.nextInt());  
            x = x % str.length;  
            output.println(str[x]);  
        } //while ends  
    } //try ends  
    catch (Exception e)  
    {  
        show(e + " ");  
    }  
    try  
    {  
        socket.close();  
    }  
    catch (IOException e)  
    {  
        show(e + " ");  
    }  
}
```



(a) Client window

```
C:\JPS\ch20>java QuoteClient 127.0.0.1
Connected with server
*****Quotation got from server *****/
"Love is blind"

C:\JPS\ch20>java QuoteClient 127.0.0.1
Connected with server
*****Quotation got from server *****/
"Live & Let Live"

C:\JPS\ch20>
```

(b) Server window

```
C:\JPS\ch20>java QuoteServer
Server waiting for client
New connection accepted
New connection accepted
```

Figure 20.3 Output screen of Programs 20.7 and 20.8

Explanation: The server program is known as “quotation server” as it returns a new random quotation whenever a new client is connected to it. For this purpose, in the server program, any array of quotation string of length 5 has been taken. The size may be increased by adding some more quotations. In the while an object **rnd** of Random class is created where the seed value is the current time passed in the constructor of the Random class. Next a random value between 0 and 4 is returned and stored in **x**. This value **x** is used as an index and **str[x]** gives the quotation to be returned to the client.

Onto the client side, the string sent by the server is read from socket stream using **input_sock** and displayed. Note the client terminates after displaying the quotation but server continues to accept new connection from the clients, provided new clients request for connection.

```
/*PROG 20.9 CHAT CLIENT FILE CHATCLIENT.JAVA */

import java.net.*;
import java.io.*;
import java.util.Date;
class ChatClient
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        int port = 1500;
        String server;
```

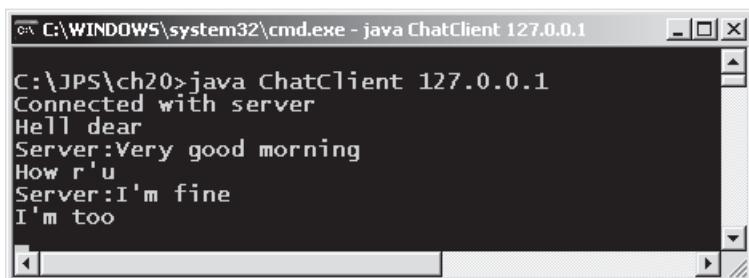
```
Socket socket = null;
BufferedReader input_sock = null, input_con = null;
PrintWriter output = null;
String msg = null;
if (args.length != 1)
{
    show("Usage : ChatClient<IP Address>");
    System.exit(0);
}
try
{
    server = args[0];
    socket = new Socket(server, port);
    show("Connected with server");
    output = new
        PrintWriter(socket.getOutputStream(), true);
    input_sock = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
    input_con = new BufferedReader(new
        InputStreamReader(System.in)));
}
catch (Exception e)
{
    show(e + " ");
    System.exit(1);
}
try
{
    while (true)
    {
        msg = input_con.readLine();
        output.println(msg);
        msg = input_sock.readLine();
        if (msg == null)
            System.exit(0);
        show("Server:" + msg);
    }
}
catch (Exception e)
{
    show(e + " ");
}
try
{
    socket.close();
}
catch (IOException e)
{
    show(e + " ");
}
}
```

```
/*PROG 20.10 CHAT SERVER, FILE CHATSERVER.JAVA */  
  
import java.net.*;  
import java.io.*;  
class ChatServer  
{  
    static void show(String s)  
    {  
        System.out.println(s);  
    }  
    public static void main(String args[])  
    {  
        int port;  
        ServerSocket server_socket;  
        PrintWriter output;  
        BufferedReader input, input_con;  
        String msg = null;  
        Socket socket = null;  
        try  
        {  
            server_socket = new ServerSocket(1500);  
            show("Server waiting for client ");  
            socket = server_socket.accept();  
            show("New connection accepted ");  
            //For writing to socket  
            output = new  
                PrintWriter(socket.getOutputStream(),  
                           true);  
            //For reading from socket  
            InputStream sin = socket.getInputStream();  
            InputStreamReader isr = new  
            InputStreamReader(sin);  
            input = new BufferedReader(isr);  
            //For reading from console  
            InputStream in = System.in;  
            InputStreamReader isrl = new  
                InputStreamReader(in);  
            input_con = new BufferedReader(isrl);  
            while (true)  
            {  
                //read from socket  
                msg = input.readLine();  
                if (msg.equals("Bye Bye"))  
                {  
                    show("Connection terminated by client");  
                    break;  
                }  
                //show onto console  
                show("Client: " + msg);  
                //read from console
```

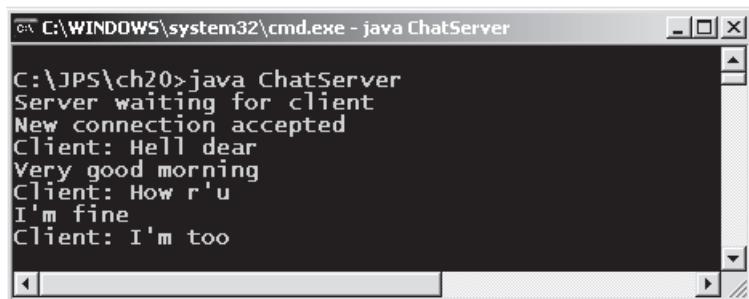
```

        msg = input_con.readLine();
        //write to socket
        output.println(msg);
    }//while ends
}//try ends
catch (Exception e)
{
    show(e + " ");
}
try
{
    socket.close();
}
catch (IOException e)
{
    show(e + " ");
}
}
}

```



(a) Client window



(b) Server window

Figure 20.4 Output screen of Programs 20.9 and 20.10

Explanation: This program is similar to Echo client-server program. The difference here is that the server does not echo back to the client, instead server displays whatever is sent by the client onto its console window. After this, it reads string from its console and sends it to the client. The client reads the string sent by the server and displays it onto its console. This continues. The main drawback of this client-server application is that the client and the server have to exchange messages in turn. That is, first, the client sends

one line of text to server, then it is server's turn to send one line of text to the client. Either of them cannot send more than one line of text continuously. For that threads have to be used in a program, which will be discussed later. The output windows of the client and the server are shown above.

```
/*PROG 20.11 SERVER FOR ADDITION OF TWO INTEGER NUMBERS
FILE SUMSERVER.JAVA */

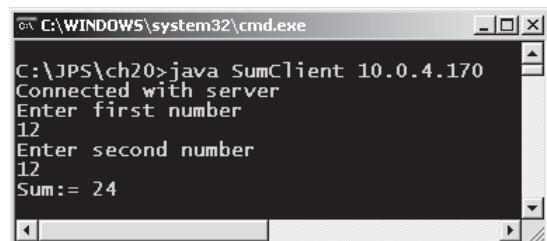
import java.net.*;
import java.io.*;
class SumServer
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        int port;
        ServerSocket server_socket;
        PrintWriter output;
        BufferedReader input, input_con;
        String msg = null;
        Socket socket = null;
        try
        {
            server_socket = new ServerSocket(1500);
            show("Server waiting for client");
            while (true)
            {
                socket = server_socket.accept();
                show("New connection accepted");
                output = new
                    PrintWriter(socket.getOutputStream(),true);
                InputStream sin = socket.getInputStream();
                InputStreamReader isr = new
                    InputStreamReader(sin);
                input = new BufferedReader(isr);
                try
                {
                    msg = input.readLine();
                    int n1 = Integer.parseInt(msg);
                    msg = input.readLine();
                    int n2 = Integer.parseInt(msg);
                    int s = n1 + n2;
                    msg = "Sum:= " + s;
                    output.println(msg);
                }
                catch (NumberFormatException e)
                {
                    show(e + " ");
                }
            }
        }
```

```
        } //while ends
    } //try ends
    catch (Exception e)
    {
        show(e + " ");
    }
    try
    {
        socket.close();
    }
    catch (IOException e)
    {
        show(e + " ");
    }
}
}
```

```
/*PROG 20.12 SERVER FOR ADDITION OF TWO INTEGER NUMBERS,
FILE SUMCLIENT.JAVA */
```

```
import java.io.*;
import java.net.*;
class SumClient
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        int port = 1500;
        String server;
        Socket socket = null;
        BufferedReader input_sock = null, input_con = null;
        PrintWriter output = null;
        String msg = null;
        if(args.length !=1)
        {
            show("Usgae: ChatClient<IP address>");
            System.exit(0);
        }
        try
        {
            server = args[0];
            socket = new Socket(server, port);
            show("Connected with server");
            output = new
                PrintWriter(socket.getOutputStream(), true);
        }
        catch (IOException e)
        {
            show("Error: " + e);
            System.exit(1);
        }
    }
}
```

```
        input_con = new BufferedReader(new
InputStreamReader(System.in));
        input_sock = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    }
    catch(Exception e)
    {
        show(e+" ");
        System.exit(1);
    }
try
{
    while(true)
    {
        show("Enter first number");
        msg = input_con.readLine();
        output.println(msg);
        show("Enter second number");
        msg = input_con.readLine();
        output.println(msg);
        msg = input_sock.readLine();
        show(msg);
        break;
    }
}
catch(Exception e)
{
    show(e+" ");
}
try
{
    socket.close();
}
catch(IOException e)
{
    show(e+" ");
}
}
}
```



(a) Client window for SumClient.java



(b) Server window for SumServer.java

Figure 20.5 Output screen of Programs 20.10 and 20.11

Explanation: In this program, the client program reads two integer numbers from the console and sends them to server one by one string form. At the server, these numbers are converted into integers and stored in n1 and n2. There addition is performed and they are returned back in the form of string as “Sum = “+s. Rest is the same as seen in previous programs. The output windows are as shown above.

```
/*PROG 20.13 A FILE TRANSFER APPLICATION, FILE SERVER CODE
FILE, FILESERVER.JAVA */
```

```
import java.net.*;
import java.io.*;
public class FileServer
{
    int LP;
    public void show(String s)
    {
        System.out.println(s);
    }
    public FileServer(int p)
    {
        LP = p;
        show("Waiting for connection");
    }
    public void accept_con()
    {
        try
        {
            ServerSocket server = new ServerSocket(LP);
            Socket con_sock = null;
            while (true)
            {
                con_sock = server.accept();
                show("Connection Accepted");
                handle_con(con_sock);
            }
        }
        catch (BindException e)
        {
            show("Unable to bind to port " + LP);
        }
        catch (IOException e)
        {
            show(" ServerSocket error on port: " + LP);
        }
    }
}
```

```

        }
    }
    public void handle_con(Socket con_sock)
    {
        try
        {
            OutputStream output = con_sock.getOutputStream();
            InputStream input = con_sock.getInputStream();
            BufferedReader input_sock;
            input_sock = new BufferedReader(new
                InputStreamReader(input));
            FileReader fr = new FileReader(new
                File(input_sock.readLine()));
            BufferedReader bfr = new BufferedReader(fr);
            PrintWriter pw = new
                PrintWriter(con_sock.getOutputStream());
            String line = null;
            while ((line = bfr.readLine()) != null)
            {
                pw.println(line);
            }
            fr.close();
            pw.close();
            input_sock.close();
        }
        catch (Exception e)
        {
            show("Error handling a client:" + e);
        }
    }
    public static void main(String[] args)
    {
        FileServer server = new FileServer(3000);
        server.accept_con();
    }
}

```

/*PROG 20.14 A FILE TRANSFER APPLICATION, FILE CLIENT CODE,
FILE FILECLIENT.JAVA */

```

import java.io.*;
import java.net.*;
public class FileClient
{
    BufferedReader sock_rdr;
    PrintWriter sock_wr;
    String HIP;
    int HP;
    public void show(String s)
    {

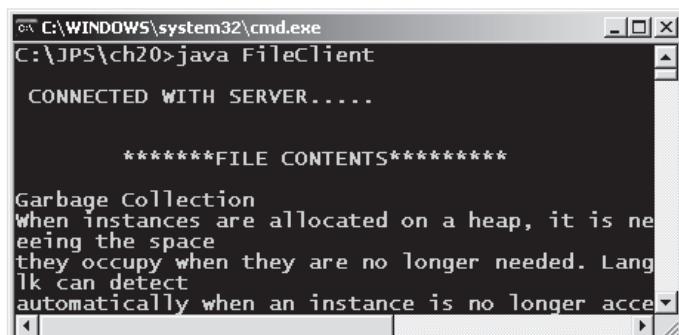
```

```
        System.out.println(s);
    }
    public FileClient(String Ip, int Port)
    {
        HIP = Ip;
        HP = Port;
    }
    public String getFile(String fname)
    {
        StringBuffer fileLines = new StringBuffer();
        try
        {
            sock_wr.println(fname);
            sock_wr.flush();
            String line = null;
            while ((line = sock_rdr.readLine()) != null)
                fileLines.append(line + "\n");
        }
        catch (IOException e)
        {
            show("\nError reading from file: " + fname);
        }
        return fileLines.toString();
    }
    public static void main(String[] args)
    {
        FileClient Fclient=new FileClient("127.0.0.1",3000);
        Fclient.setUpCon();
        String fileContents = Fclient.getFile("c:\\file.txt");
        Fclient.shutDownCon();
        System.out.println("\n\t*****FILECONTENTS*****\n");
        System.out.println(fileContents);
    }
    public void setUpCon()
    {
        try
        {
            Socket client = new Socket(HIP, HP);
            sock_rdr = new BufferedReader(new
                InputStreamReader(client.getInputStream()));
            sock_wr = new
                PrintWriter(client.getOutputStream());
            show("\n CONNECTED WITH SERVER.....");
        }
        catch(UnknownHostException e)
        {
            show("Unkown host at " +HIP+ ":" +HP);
        }
        catch(IOException e)
        {
            show("Error setting up connection: "+e);
```

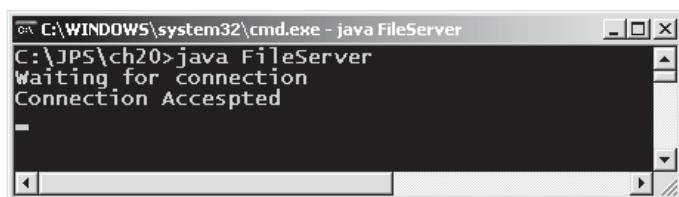
```

        }
    }
    public void shutDownCon() {
        try
        {
            sock_wr.close();
            sock_rdr.close();
        }
        catch (IOException e)
        {
            show("Error shutting down connection: " +
e);
        }
    }
}

```



(a) Client window FileClient.java



(b) Server window FileServer.java

Figure 20.6 Output screen of Programs 20.13 and 20.14

Explanation: In this application, there is a file client program that requests for a file residing onto the server. The server locates the file onto the server, opens it and transfers the contents of file to the client line by line. The client stores the contents of file into a string buffer using the append method. In the end, the string buffer instance is converted to string.

Both the client and server programs are different from the other programs seen so far. Here some methods are used for different sections of the program. In the client program, there is a constructor for storing the host and port number into class members. The method getFile gets a file from the server. The method getFile returns the file contents as String object. The method setUpCon is there for setting up the socket connection from host and port number, creating input and output stream for socket. The method shutDownCon closes the connection.

In the server-side code, the file name is retrieved by the server. Connection from the server is accepted within `accept_con` function. A `FileReader` instance `fr` is created for reading the contents of the file. This `fileReader` instance is passed to constructor of `BufferedReader` class and same is assigned to `bfr`. Now from this `bfr`, the contents of file are read. Reading from the file using `bfr` until `readLine` returns null and sending the read line to the client are continued using stream to socket `pw` which was created earlier using `PrintWriter` class.

This program can be modified to construct the file onto the client side, store the contents into it and read from the server. The output windows are displayed above. Here a file '**file.txt**' are read residing in the C: directory onto the same machine.

```
/*PROG 20.15 SIMPLE CLIENT FOR DISPLAYING SOCKET RELATED
INFORMATION */
```

```
import java.net.*;
import java.io.*;
import java.util.Date;
class Info_Socket_Client
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        int port = 1500;
        Socket socket = null;
        String server;
        if (args.length != 1)
        {
            show("Usage : Client <IP address>");
            System.exit(0);
        }
        try
        {
            server = args[0];
            socket = new Socket(server, port);
            show("Connected with server");
            show("Socket's host IP: "
                +socket.getInetAddress());
            show("Socket's Port no: "
                +socket.getPort());
            show(" and Local Port no. := "
                +socket.getLocalPort());
        }
        catch (Exception e)
        {
            show(e + " ");
            System.exit(1);
        }
    }
}
```

```
        }
        try
        {
            socket.close();
        }
        catch (IOException e)
        {
            show(e + " ");
        }
    }
}
```

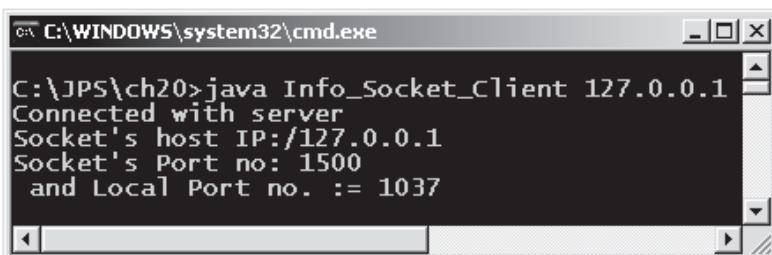
```
/*PROG 20.16 SIMPLE SERVER FOR DISPLAYING SOCKET RELATED  
INFORMATION */
```

```
import java.net.*;
import java.io.*;
class Info_Socket_Server
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String args[])
    {
        ServerSocket server_socket;
        Socket socket = null;
        try
        {
            server_socket = new ServerSocket(1500);
            while (true)
            {
                show("Server waiting for client ");
                socket = server_socket.accept();
                show("Client Port no.:= "
                     +socket.getPort());
                show("Local Port no. := "
                     +socket.getLocalPort());
            }
        }
        catch (Exception e)
        {
            show(e + " ");
        }
        try
        {
            socket.close();
        }
```

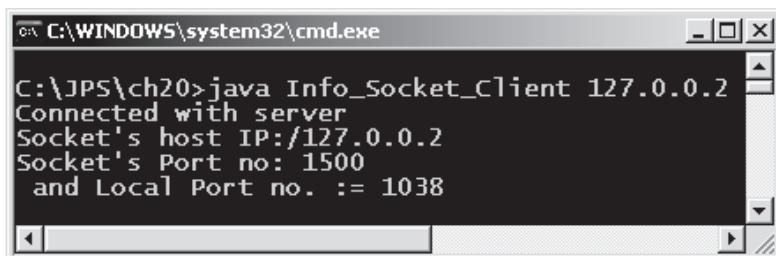
```
        catch (IOException e)
    {
        show(e + " ");
    }
}
```



(a) Server window



(b) Client window having IP 127.0.0.1



(c) Client window having IP 127.0.0.2

Figure 20.7 Output screen of the Programs 20.15 and 20.16

Explanation: The application is very simple. Onto the client side, using its connecting socket, the host IP address, its port number and local port number are displayed onto the server to which this socket is communicating. In the server side, whenever a new client is connected, a new connecting socket is given for that client at a port number, that is generated dynamically, and all communication to that client takes place using that socket with newly created port number and IP address. Now onto server side the socket.getPort gives that port number. The method getLocalPort gives the port number on which the server is listening (i.e., 1500 in this case).

On client side, local port returns the port number onto which client is communicating with server using `getLocalPort`. The main port number is obtained using `getPort` method.

Now it can be said that `getPort` onto server side and `getLocalPort` onto client side give the same port number and `getLocalport` onto server side and `getPort` onto client side give the same portnumber.

The output is shown below for two clients and one server. Note the port for both the clients onto the server side is different and the local port is the same.

20.5.4 Thread Chat Server and Client

In the previous section, a simple chat application was developed. The limitation of the application was that both the client and server have to send message in turn. This limitation is now removed by providing a separate thread of execution for each client onto the server side. In this case, the code that client and server use for chatting with each other is separated. For this purpose, both the client and server programs use `Thread` class. The code is given below:

```
/*PROG 20.17 CHAT SERVER CODE, USING MULTITHREADING */

import java.io.*;
import java.net.*;
import java.util.*;
class Thread_Server extends Thread
{
    ServerSocket ss;
    static Socket s;
    public Thread_Server()
    {
        System.out.println("Server is ready....Type your
                           message :");
        try
        {
            ss = new ServerSocket(25003);
            s = ss.accept();
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
        start();
    }
    public void run()
    {
        try
        {
            DataInputStream dis;
            dis = new DataInputStream(s.getInputStream());
        }
    }
}
```

```

        while (true)
        {
            System.out.println("Client: " + dis.readUTF());
            sleep(250);
        }
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
public static void main(String[] args)
{
    new Thread_Server();
    try
    {
        DataOutputStream dos;
        dos = new DataOutputStream(s.getOutputStream());
        InputStreamReader isr;
        isr = new InputStreamReader(System.in);
        BufferedReader br;
        br = new BufferedReader(isr);
        while (true)
        {
            dos.writeUTF(br.readLine());
        }
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
}

```

/*PROG 20.18 CHAT CLIENT CODE, USING MULTITHREADING */

```

import java.util.*;
import java.io.*;
import java.net.*;
class Thread_Client extends Thread
{
    static Socket s;
    DataInputStream dis;
    public Thread_Client()
    {
        try
        {
            System.out.println("Client is ready... (Type u'r
message):\n");

```

```
s = new Socket("Localhost",25003);
dis = new DataInputStream(s.getInputStream());
}
catch(Exception ie)
{
    System.out.println(ie);
}
start();
}
public void run()
{
    while(true)
{
try
{
    System.out.println("Server: "+dis.readUTF()+"\n");
    sleep(250);
}
    catch(Exception e)
{
    System.out.println(e);
}
}
}
public static void main(String[]args)
{
    new Thread_Client();
try
{
    InputStreamReader isr;
    isr = new InputStreamReader(System.in);
    BufferedReader br;
    br = new BufferedReader(isr);
    DataOutputStream dos;
    dos = new DataOutputStream(s.
                                getOutputStream());
    while(true)
    {
        String msg = br.readLine();
        if(msg.equals("Bye Bye"))
            System.exit(0);
        dos.writeUTF(msg);
    }
}
catch(Exception e)
{
    System.out.println(e);
}
}
```

```
C:\> C:\WINDOWS\system32\cmd.exe - java Thread_Server
C:\JPS\ch20>java Thread_Server
Server is ready....Type your message :
Client: Hello
Client: what r'u doing
nothing special
where r'u nw a day's
Client: I'm in Mumbai
Client: doing job
```

(a) Server Window Thread_Server.java

```
C:\> C:\WINDOWS\system32\cmd.exe - java Thread_Client
C:\JPS\ch20>java Thread_Client
Client is ready...(Type u'r message):
Hello
what r'u doing
Server: nothing special

Server: where r'u nw a day's

I'm in Mumbai
doing job
```

(b) Client Window Thread_Client.java

Figure 20.8 Output screen of Programs 20.17 and 20.18

Explanation: The server class `Thread_Server` extends `Thread` class. In the default constructor of this method, a server socket is created by the name `ss` which listens at port number 25003. The server waits for the incoming connection using `accept` method. The default constructor is called in the `main` method. When the server executes, `main` method is called and in the `main`, default constructor is called. From the default constructor, `start` method causes `run` method to be called. In the `run` method, an instance `dis` of `DataInputStream` refers to socket input stream. The string sent by the client is read from the socket stream using `readUTF` method. Reads from the stream is a representation of a unicode character string encoded in modified **UTF-8** format; this string of characters is then returned as a string. The **UTF** in the method stands for Unicode Text Format.

In the main method of server code, the string data is read from the console and sent to the client. The important point to note here is that reading from client and displaying onto the console is handled by the thread in `run` method, and reading from console and sending to the client is done within `main` method in the `while` loop. That makes the code work.

Onto the client side, the `run` method reads data sent by the server and displays onto the console. In the `main`, the string data is read from the console and sent to the server. When the string '**Bye Bye**' is entered onto the console, the client terminates.

20.5.5 Datagram

A datagram is an independent, self-contained message sent over the network whose arrival time and content are not guaranteed.

Client and servers that communicate via a reliable channel, such as a URL or a socket, have a dedicated point-to-point channel between themselves or at least the illusion of one. To communicate, they establish a connection, transmit the data, and then close the connection. All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.

In contrast, applications that communicate via datagram, send and receive completely independent packets of information. These clients and servers do not need a dedicated point-to-point channel. The delivery of data to its destination is neither guaranteed nor is the order of its arrival.

The `java.net` package contains three classes to help one write Java programs that use datagrams to send and receive packets over the network: **DatagramSocket**, **DatagramPacket**, and **MulticastSocket**. An application can send and receive `DatagramPackets` through a `DatagramSocket`. In addition, `DatagramPackets` can be broadcast to multiple recipients all listing to a `MulticastSocket`.

The DatagramPacket Class: This class represents a datagram packet. Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to other solely based on information contained within that packet. Multiple packets sent from one machine to another might be routed differently, and might arrive in any order. Packet delivery is not guaranteed.

The class defines the following constructors for the creation of datagram packets.

1. **public DatagramPacket (byte[]buf, int length)**

This form of constructor creates a `DatagramPacket` for receiving packets of length 'length'. The 'length' argument must be less than or equal to `buf.length`.

2. **public DatagramPacket (byte[]buf, int ioffset, int length, InetAddress address, int port)**

This form of constructor creates a datagram packet for sending packets of length 'length' with offset 'ioffset' to the specified port number on the specified host. The length argument must be less than or equal to `buf.length`. The `address` and `port` are destinations.

3. **public DatagramPacket (byte[]buff, int length, InetAddress address, int port)**

This form of constructor creates a datagram packet for sending packets of length 'length' to the specified port number on the specified host. The length argument must be less than or equal to `buf.length`.

4. **public DatagramPacket (byte[]buf, int offset, int length)**

This form of constructor creates a `DatagramPacket` for receiving packets of length 'length', specifying an offset into the buffer. The length argument must be less than or equal to `buf.length`.

Some of the useful methods of this class are given in the Table 20.2.

| Method Signature | Description |
|--|---|
| <code>public InetAddress getAddress()</code> | Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received. |
| <code>public byte[]getData()</code> | Returns the data buffer. The data received or the data to be sent starts from the offset in the buffer, and runs for long length. |
| <code>public int getLength()</code> | Returns the length of the actual data to be sent or the length of the actual data received. |
| <code>public int getPort()</code> | Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received. |

Table 20.2 Methods of `DatagramPacket` class

Other methods for setting the length, port, data, etc, are not discussed.

The DatagramSocket Class: This class represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and they arrive in any order.

The three frequently used constructors of this class are as follows:

- 1. public DatagramSocket() throws SocketException**

This form of constructor creates a datagram socket and binds it into any available port on the local host machine.

- 2. public DatagramSocket(int port) throws SocketException**

This form of constructor creates a datagram socket and binds it to the specified port on the local host machine.

- 3. public DatagramSocket(int port, InetAddress laddr) throws SocketException**

This form of constructor creates a datagram socket, bound to the specified local address. The local port must be between 0 and 65535 inclusive.

The two methods send and receive of this class are used for sending and receiving data. Now, few practical examples of datagram sockets and packets are given below:

```
/*PROG 20.19 DATAGRAM PACKET SENDER, EQUIVALENT TO CLIENT */

import java.io.*;
import java.net.*;
class DSender
{
    public static void main(String[] args) throws
IOException
    {
        InetAddress addr = InetAddress.getByName(args[0]);
        byte[] buf = "Hello from Sender".getBytes();
        DatagramPacket p;
        p = new DatagramPacket(buf, buf.length, addr,
1115);
        DatagramSocket socket = new DatagramSocket();
        socket.send(p);
    }
}
```

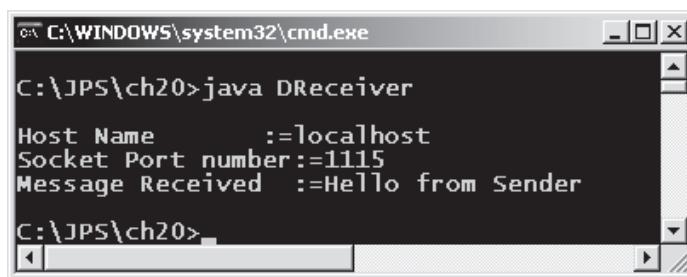
```
/*PROG 20.20 DATAGRAM PACKET RECEIVER, EQUIVALENT TO SERVER */

import java.io.*;
import java.net.*;
public class DReceiver
{
    static void show(String s)
    {
        System.out.println(s);
    }
    public static void main(String[] args) throws
IOException
    {
```

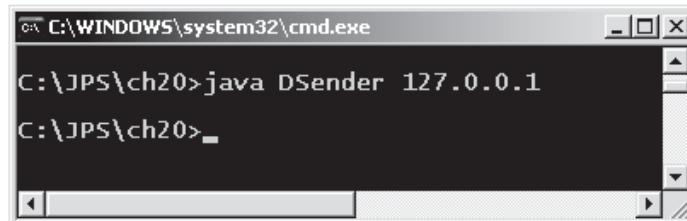
```

DatagramSocket socket = new DatagramSocket(1115);
byte[] buf = new byte[256];
DatagramPacket p = new DatagramPacket(buf,
        buf.length);
socket.receive(p);
String s = new String(p.getData(),0,p.getLength());
show("\nHost Name:=" + p.getAddress().getHostName());
show("Socket Port number:=" + socket.getLocalPort());
show("Message Received :=" + s);
}
}

```



(a) Receiver window DReceiver.java



(b) Sender window DSender.java

Figure 20.9 Output screen of Programs 20.19 and 20.20

Explanation: In the file **DSender.java**, the IP address is passed as command line argument. The string '**Hello from Sender**' is converted into byte array **buff** using **getBytes** method. A **DatagramPacket** instance **p** is created using **DatagramPacket** constructor form that takes four parameters: byte array **buff**, its length, **InetAddress** **addr** where packet is to be sent and, destination port number 1115. The packet is sent using **send** method with **DatagramSocket** instance **socket**.

Onto the receiving side, that is, in file **DReceiver.java**, a new **DatagramSocket** instance **socket** is created with port number 1115. A new **DatagramPacket** instance **p** is created using its constructor form that takes two parameters byte array **buf** and its length. In this **DatagramPacket** instance **p**, the data sent by the sender is received. The received data is in byte array form, so it is converted into **String** object **s** and displayed. For the conversion, the byte array from **DatagramPacket** is extracted using **getData** method and its length using **getLength** method. Rest is simple to understand. The sender may be considered as client and receiver as server.

```
/*PROG 20.21 DATAGRAM ECHO SERVER */

import java.net.*;
import java.io.*;
public class DEReceiver
{
    public static void main(String[] args) throws IOException
    {
        DatagramSocket socket = new DatagramSocket(1115);
        while (true)
        {
            byte[] buf = new byte[256];
            DatagramPacket p=new DatagramPacket(buf, buf.length);
            socket.receive(p);
            socket.send(p);
        }
    }
}
```

```
/*PROG 20.22 DATAGRAM ECHO CLIENT */

import java.io.*;
import java.net.*;
class DESender
{
    public static void main(String[] args) throws IOException
    {
        InetAddress addr = InetAddress.getByName(args[0]);
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(isr);
        String msg;
        DatagramSocket socket = new DatagramSocket();
        DatagramPacket p;
        while (true)
        {
            msg = bf.readLine();
            byte[] buf = msg.getBytes();
            p = new DatagramPacket(buf, buf.length, addr, 1115);
            socket.send(p);
            if (msg.equals("Bye Bye"))
                System.exit(0);
            socket.receive(p);
            msg = new String(p.getData(), 0, p.getLength());
            System.out.println("Echoed by Server:" + msg);
        }
    }
}
```

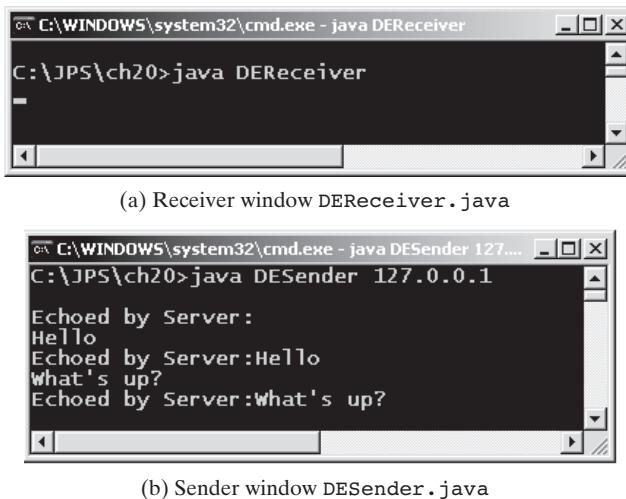


Figure 20.10 Output screen of Programs 20.21 and 20.22

Explanation: One echo client-server was made earlier using TCP/IP sockets. Here the same logic is followed but with `DatagramSocket` and `DatagramPacket` classes. It is not hard to understand how the application is working. This can be checked out in itself. The output windows are shown above. The other client server programs may be tried which has been created using TCP/IP sockets.

20.6 URL

URL is the acronym for uniform resource locator. It is a reference (an address) to a resource on the Internet. URLs are provided to one's favorite Web browser so that it can locate files on the Internet in the same way that addresses are provided on letters so that the post office can locate one's correspondents. Those who have been surfing the Web, must be familiar with the term URL, which is used to access HTML pages from the Web. However, remember that URLs also can point to other resources on the network, such as database queries and command output.

Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. Java programs can use a class called `URL` in the `java.net` package to represent a URL address.

20.6.1 An Example of URL

The following is an example of a URL which addresses a popular Web site: <http://www.rediff.com>

In the above URL, there are two components: protocol identifier and resource name.

The protocol name is on the left and separated by resource name using a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the Internet. Other protocols include File Transfer Protocol (FTP), Gopher, File and News.

The resource name is the complete address to the resource; the format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the components listed in the following table:

| | |
|-------------|---|
| Host Name | The name of the machine on which the resource lives. |
| Filename | The pathname to the file on the machine. |
| Port Number | The port number to which to connect (typically optional). The default port for http is 80. |
| Reference | A reference to a named anchor within a resource that usually identifies a specific location within a file (typical optional). |

The URL class in Java represents a uniform resource locator, a pointer to a ‘resource’ on the world wide web. The class defines a number of constructors or the creation of URLs. Some of them are shown below:

1. **public URL(String spec) throws MalformedURLException**

This form of constructor creates a URL object from the String representation. For example:

```
URL myurl = new URL("http://www.yahoo.com/");
```

2. **public URL(String protocol, String host, String file) throws MalformedURLException**

This form of constructor creates a URL from the specified protocol name, host name and file name. The default port for the specified protocol is used. For example

```
URL u1 = new URL("http", "www.havefun.com", /pages/Havefun.net.html");
```

The above is equivalent to:

```
URL u1 = new URL("http://www.havefun.com/pages/Havefun.net.html");
```

3. **public URL(String protocol, String host, int port, String file) throws MalformedURLException**

This form of constructor creates a URL object from the specified protocol, host, port number and file.

20.6.2 Creating Relative URLs

If the Java programs, a URL object can be created from a relative URL specified. For example, suppose two URLs are known at the Havefun site:

```
http://www.havefun.com/pages/Havefun.game.html  
http://www.havefun.com//pages/Havefun.net.html
```

URL object can be created for these pages relative to their common base URL `http://www.havefun.com/pages` like this:

```
URL havefun = new URL("http://www.havefun.com/pages/");  
URL havefunGames = new URL(havefun, "Havefun.game.html");  
URL havefunNetwork = new URL(havefun, "Havefun.net.html");
```

This code snippet uses the URL constructor that lets one creates URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:

```
URL(URL baseURL, String relativeURL)
```

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. If `baseURL` is null, this constructor treats `relativeURL` like an absolute URL specification. Conversely, if `relativeURL` is an absolute URL specification, the constructor ignores `baseURL`.

20.6.3 The MalformedURLException

Each of the four URL constructors seen above throws a MalformedURLException if the argument to the constructor refers to a null or unknown protocol. Typically, one wants to catch and handle this exception by embedding the URL constructor statements in a try/catch pair, like this:

```
try
{
    URL myURL = new URL (...)
}
catch(MalformedURLException e)
{
    .....
    //exception handler code here
}
```

20.7 PONDERABLE POINTS

1. Every computer has, over a network, got an Internet Protocol (IP) address that uniquely identifies a device or system connected to the Internet.
2. Because computers are capable of providing more than one type of service, a way is needed to uniquely identify each service. For this purpose, a number is used that is known as software port.
3. The number 1–1024 are known as reserved ports and beyond that but below 65536 can be used by application programs.
4. For communication over the network, two important communication protocols are used: TCP and UDP. TCP is connection oriented and reliable; UDP is connectionless and unreliable.
5. Support for networking in Java is provided by the java.net package.
6. InetAddress provides simple methods to convert between domain name and numbered addresses. This class represents an Internet Protocol (IP) address.
7. A socket is one endpoint of a two-way communication link between two programs running on the network.
8. A socket has two parts: an IP address and a port number.
9. The socket associated with TCP is known as the stream socket.
10. The socket associated with UDP is known as the datagram socket.
11. A server-side socket is also known as listening socket as it listens for incoming requests.
12. A client-side socket is also known as connecting socket at it tries to connect to the server.
13. At the core of Java's networking support are the `Socket`, `DatagramSocket` and `ServerSocket` classes in `java.net`. These classes define channels for communication between processes over an IP network.
14. The `ServerSocket` class provides server sockets or sockets at server side. Such sockets wait for requests over an IP network.
15. The `Socket` class provides client-side sockets or simply sockets. They are at the client side connecting to the server, sending the request to the server and accepting the returned result.
16. Reading and writing from/to can be done using `getInputStream` and `getOutputStream` method.
17. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time and content are not guaranteed.
18. The `java.net` package contains three classes to help one write Java programs that use datagrams to send and receive packets over the network: `DatagramSocket`, `DatagramPacket` and `MulticastSocket`.
19. An application can send and receive DatagramPackets through a `DatagramSocket`.
20. URL is the acronym for uniform resource locator. It is a reference (an address) to a resource on the Internet.

REVIEW QUESTIONS

1. What is the difference between TCP/IP and UDP/IP?
2. Write a program for simple chatting using TCP.
3. Write a program for simple chatting using UDP.
4. What is a port? What is a well-known port? Give example.
5. What is IP address? Explain different types of IP addresses? Why are they used?
6. What is the difference between IP address and Port address?
7. What is the use of java.net package? Explain with an example.
8. What is the difference between stream socket and datagram socket?
9. Explain the following:
 - (a) Socket Class
 - (b) ServerSocket Class
 - (c) DatagramSocket Class
10. Write short note on the DNS. Also give an example.
11. What is a socket? Which socket is used for Client and Server in TCP?
12. What is a URL? Write a program to parse a URL.
13. Give the procedure to create a relative URL.
14. What is datagram? What classes are provided by java.net packages in support of datagram?

Multiple Choice Questions

1. The socket has two parts: an IP address and a
 - (a) socket address (c) interface
 - (b) port number (d) none of the above
2. A high level API deals with
 - (a) interface (c) URLs
 - (b) sockets (d) none of the above
3. A low level API deals with
 - (a) addresses (c) (a) and (b)
 - (b) sockets (d) connections
4. Internet addresses are represented in Java by
 - (a) InetAddress class (c) socket class
 - (b) IPaddress class (d) none of the above
5. byte[] getAddress() returns the raw IP address of the InetAddress object. The result is in network byte order, the highest order byte of the address is in
 - (a) getAddress()[0]
 - (b) getAddress()[1]
 - (c) getAddress()[2]
 - (d) getAddress()[3]
6. Datagram packets are used to implement a connectionless packet delivery service. The constructor defined is
 - (a) public DatagramPacket()
 - (b) public DatagramPacket(int length)
 - (c) public DatagramPacket(byte buf[])
7. The public int getLength() method returns
 - (a) Length of IP
 - (b) Length of actual data to be sent
 - (c) Length of the actual data received
 - (d) (b) and (c)
8. Which constructor of DatagramSocket class is used to construct a datagram socket and binds it to any available port on the local host machine?
 - (a) public DatagramSocket throws SocketException()
 - (b) public DatagramSocket(byte buf[]) throws SocketException
 - (c) public DatagramSocket() throws SocketException
 - (d) None of the above
9. The _____ class provides server sockets or socket at server side. The _____ class provides client-side sockets or simply sockets.
 - (a) ServerSocket, ClientSocket
 - (b) Socket, ClientSocket
 - (c) ServerSocket, Socket
 - (d) None of the above
10. In Java, networking is supported by
 - (a) java.netw package (c) java.util package
 - (b) java.net package (d) None of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. b 2. c 3. c 4. a 5. a 6. d 7. d 8. c 9. c 10. b

21.1 JAVA NATIVE INTERFACE (JNI)

The Java native interface (JNI) is a powerful feature of the Java platform. Applications that use JNI can incorporate the native code written in programming languages such as C and C++, as well as in the Java programming language.

The Java platform is a programming environment consisting of the Java virtual machine (VM) and the Java application programming interface (API). Java applications are written in the Java programming language, and compiled into a machine-independent binary class format. A class can be executed on any Java virtual machine implementation. The Java API consists of a set of predefined classes. Any implementation of the Java platform is guaranteed to support the Java programming language, VM and API.

The term host environment represents the host operating system, a set of native libraries and the CPU instruction set. Native applications are written in native programming languages such as C and C++, compiled into host-specific binary code and linked with native libraries. Native applications and native libraries are typically dependent on a particular host environment. A C application built for one operating system, for example, typically does not work on other operating systems.

Java platforms are commonly deployed on top of a host environment. For example, the Java runtime environment (JRE) is a Sun product that supports the Java platform on existing operating systems such as Solaris and Windows. The Java platform offers a set of features that applications can rely on, independent of the underlying host environment.

21.1.1 Role of the JNI

The JNI is a powerful feature that allows one to take advantage of the Java platform, but still utilize code written in other languages. As a part of the Java virtual machine implementation, the JNI is a two-way interface that allows Java applications to invoke native code and vice versa. Figure 21.1 illustrates the role of the JNI.

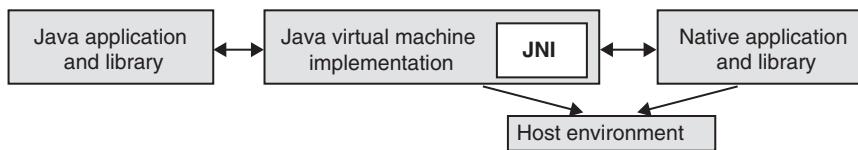


Figure 21.1 Role of JNI for Java virtual machine

The JNI is designed to handle situations where Java applications need to be combined with native code. As a two-way interface, the JNI can support two types of native code: libraries and native applications.

- The JNI can be used to write native methods that allow Java applications to call functions implemented in native libraries. Java applications call native methods in the same way that they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in another language and reside in native libraries.

- The JNI supports an invocation interface that allows the application to embed a Java virtual machine implantation into native applications. Native applications can link with a native library that implements the Java virtual machine, and then use the invocation interface to execute software components written in the Java programming language. For example, a Web browser written in C can execute downloaded applets in an embedded Java virtual machine implementation.

Sometimes, however, it might be necessary for a Java application to communicate with native code that resides in the same process. This is when the JNI becomes useful.

21.1.2 An Example of JNI

In this section, a simple example of using JNI is written. A Java application will be written that calls a C function to print **Hello World!**.

Figure 21.2 illustrates the process for writing a simple Java application that calls a C function to print ‘Hello World!’. The process consists of the following steps:

1. Create a class (`HelloWorld.java`) that declares the native method.
2. Use `javac` to compile `HelloWorld` source file, resulting in the class file `HelloWorld.class`.
3. Use `javah -jni` to generate a C header file (`HelloWorld.h`) containing the function prototype for the native method implementation. Write the C implementation (`HelloWorld.c`) of the native method.
4. Compile the C implementation into a native library, creating `HelloWorld.dll`. Use the C compiler and linker available on the host environment.
5. Run the `HelloWorld` program using the Java runtime interpreter. Both the class files (`HelloWorld.class`) and the native library (`HelloWorld.dll`) are loaded at runtime.

All these steps are discussed in detail.

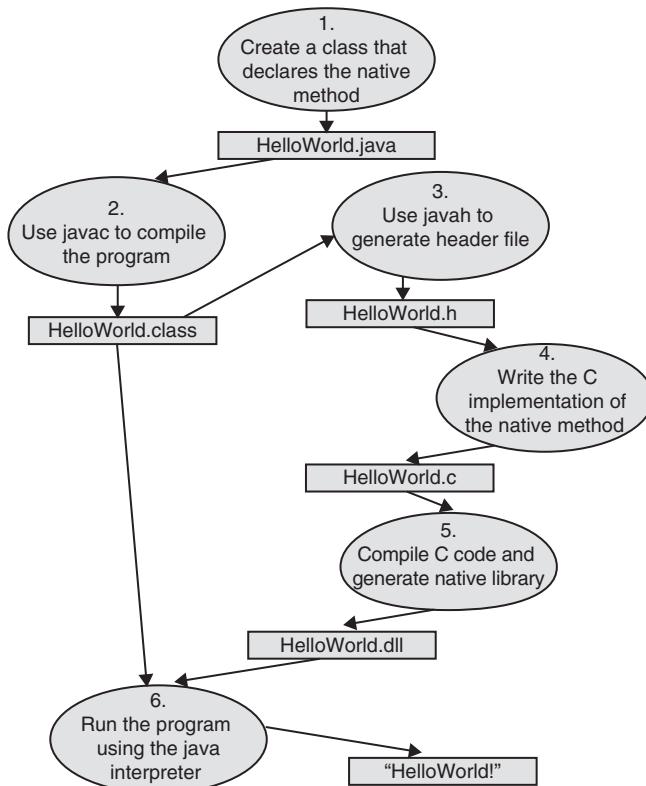


Figure 21.2 Steps in writing and running the ‘Hello World’ program

Declare the Native Method

The application begins by writing the following program in the Java programming language. The program defines a class named 'HelloWorld' that contains print method.

```
/*PROG 21.1 DEMO OF JNI */

class HelloWorld
{
    private native void print();
    public static void main(String[] args)
    {
        new HelloWorld.print();
    }
    static
    {
        System.loadLibrary("HelloWorld");
    }
}
```

The HelloWorld class definition begins with the declaration of the print native method. This is followed by a main method that instantiates the Hello-World class and invokes the print native method for this instance. The last part of the class definition is a static initializer that loads the native library containing the implementation of the print native method.

There are two differences between the declaration of a native method such as print and that of regular methods in the Java programming language. That a native method declaration must contain the native modifier indicates that this method is implemented in another language. Also, the native method declaration is terminated with a semicolon, the statement terminator symbol, because there is no implementation for native methods in the class itself. The print method will be implemented in a separate C file.

Before the native method print can be called, the native library that implements print must be loaded. In this case, the native library is loaded in the static initializer of the HelloWorld class. The Java virtual machine automatically runs the static initializer before invoking any methods in the HelloWorld class, thus ensuring that the native library is loaded before the print native method is called.

A main method is defined to be able to run the HelloWorld class. Hello-World.main calls the native method print in the same manner as it would call a regular method.

`System.loadLibrary` takes a library name, locates a native library that corresponds to that name and loads the native library into the application. For now, simply remember that in order for `System.loadLibrary("HelloWorld")` to succeed, a native library called `HelloWorld.dll` needs to be created on windows.

Compile the HelloWorld Class

After the HelloWorld class is defined, save the source code in a file called `HelloWorld.java`. Then compile the source file using the `javac` compiler.

```
javac HelloWorld.java
```

This command will generate a `HelloWorld.class` file in the current directory.

Create the Native Method Header File

Next, the `javah` tool will be used to generate a JNI-style header file that is useful when implementing the native method in C. `javah` can be run on the `Hello-world` class as follows:

```
javah -jni HelloWorld
```

The name of the header file is the class name with a “.h” appended to the end of it. The command as shown above generates a file name **HelloWorld.h**. The generated file is listed here:

```
/* DO NOT EDIT THIS FILE - it is machine generated */

#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
* Class: HelloWorld
* Method: print
* Signature: ()V
*/
JNIEXPORT void JNICALL Java_HelloWorld_print
    (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

The most important part of the header file is the function prototype for `Java_HelloWorld_print`, which is the C function that implements the `HelloWorld.print` method:

```
JNICALL void Java_HelloWorld_print (JNIEnv *, jobject);
```

Ignore the `JNIEXPORT` and `JNICALL` macros for now. One may have noticed that the C implementation of the native method accepts two arguments even though the corresponding declaration of the native method accepts no arguments. The first argument for every native method implementation is a `JNIEnv` interface pointer. The second argument is a reference to the `HelloWorld` object itself (sort of like the ‘this’ reference). How to use the `JNIEnv` interface pointer and the `jobject` arguments are discussed later in this chapter, but this simple example ignores both arguments.

Write the Native Method Implementation

The JNI-style header file generated by the javah helps one to write C or C++ implementation for the native method. The function that the application writes must follow the prototype specified in the ‘generates header’ file. The `Hello-World.print` method can be implemented in a C file `HelloWorld.c` as follows:

```
#include<jni.h>
#include<stdio.h>
#include"HelloWorld.h"
JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject
obj)
{
    printf("Hello World!\n");
    return;
}
```

The implementation of this native method is straightforward. It uses the `printf` function to display the string '**Hello World!**' and then returns. As mentioned before, both arguments, the `JNIEnv` pointer and the reference to the object, are ignored.

The C program includes three header files:

- `jni.h`: This header file provides information that the native code needs to call JNI functions. When writing methods, this file must always be included in a C or C++ source files.
- `stdio.h`: The code snippet above also includes `stdio.h` because it uses the `printf` function.
- `HelloWorld.h`: This header file is generated using `javah`. It includes the C/C++ prototype for the `Java_HelloWorld_print` function.

Compile the C Source and Create a Native Library

Remember that when creating the `HelloWorld` class in the `HelloWorld.java` file, the application includes a line of code that loaded a native library into the program:

```
System.loadLibrary("HelloWorld");
```

Now that all the necessary C code is written, `Hello-World.c` needs to be compiled and this native library built.

On Windows, the following command builds a dynamic link library (DLL) `HelloWorld.dll` using the Microsoft Visual C++ compiler.

```
CL-Ic: \java\include\ -Ic:\java\include\win32 -MD -LD  
HelloWorld.c -FeHelloWorld.dll
```

`Java` is the root directory where `java` is installed. The `/Fefilename` option names an .EXE file or DLL or creates it in a different directory. No space is allowed between `/Fe` and `filename`.

The `-MD` option ensures that `HelloWorld.dll` is linked with the Windows multithreaded C library. The `-LD` option instructs the C compiler to generate a DLL instead of a regular Win32 executable. On Windows, the included paths need to be put in that reflect the set-up on a machine. The `CL` command is found in the `bin` directory of `VC98` folder.

Run the Program

At this point, the two components are ready to run the program. The class file (`HelloWorld.class`) calls a native method, and the native library (`HelloWorld.dll`) implements the native method.

Because the `HelloWorld` class contains its own `main` method, the program can be run on Windows as follows. Note before running `HelloWorld.dll`, file has to be copied in the directory where `.class` file is present.

```
java HelloWorld
```

The following output should result:

```
Hello World!
```

21.1.3 Limitations of Using JNI

There are two main limitations of using JNI:

1. Java applications that depend on the JNI can no longer readily run on multiple host environments. Even though the part of an application written in the Java programming language is portable to multiple host environments, it will be necessary to recompile the part of the application written in native programming languages.
2. While the Java programming language is type-safe and secure, native languages such as C or C++ are not. As a result, extra care must be taken when writing applications using the JNI. A misbehaving native method can corrupt the entire application. For this reason, Java applications are subject to security checks before invoking JNI features.

21.2 SERIALIZATION

Serialization is a process by which the object writes a record of itself to a persistent storage medium such as a disk file and reloads itself by reading the record back. Serialization is the process of converting a set of object instances that contains references to each other into a linear stream of bytes, which can then be sent through a socket, sorted to a file or simply manipulated as a stream of data. Serialization is a mechanism used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation. The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be recreated by reading, or deserializing, the object's state from the secondary storage. Serialization handles all the details of object pointers and circular references to objects that are used when an object is serialized. A key point is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization process.

In particular, there are three main uses of serialization:

1. *As a persistence mechanism:* If the stream being used is `FileOutputStream`, the data will automatically be written to a file.
2. *As a copy mechanism:* If the stream being used is `ByteArrayOutputStream`, the data will be written to a byte array in memory. This byte array can then be used to create duplicates of the original objects.
3. *As a communication mechanism:* If the stream being used comes from a socket, the data will automatically be sent over the wire to the receiving socket, at which point another program will decide what to do.

The important thing to note is that the use of serialization is independent of the serialization algorithm itself. If there is a serializable class, it can be saved to a file or a copy of it can be made simply by changing the way the output of the serialization mechanism is used.

As one might expect, serialization is implemented using a pair of streams. Even though the code that underlies serialization is quite complex, the way it is invoked is designed to make serialization as transparent as possible to Java developers. To serialize an object, create an instance of `ObjectOutputStream` and call the `writeObject()` method; to read in a serialized object, create an instance of `ObjectInputStream` and call the `readObject()` object.

21.2.1 Class and Interfaces for Serialization

The basic serialization mechanism is implemented by interfaces `Serializable`, `ObjectOutput` and `ObjectInput`, and class `ObjectInputStream` and `ObjectOutputStream`.

1. *The Serializable interface:* The serializable interface is a must for serialization. Serializability of a class is enabled by the class implementing the `java.io.Serializable` interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable. The variables with transient and static modifiers cannot be saved.
2. *The ObjectOutput interface:* `ObjectOutput` extends the `DataOutput` interface to include writing of objects. `DataOutput` includes methods for output of primitive types; `ObjectOutput` extends that interface to include objects, arrays and Strings. Most frequently used method of this interface is the `writeObject` which is used for writing state of the object to persistent media. The various methods of this interface are shown in the Table 21.1.

| Method Signature | Description |
|--|--|
| <code>void close()</code> | Closes the invoking stream. Further write attempts will generate an IOException. This method must be called to release any resources associated with the stream. |
| <code>void flush()</code> | Flushes the stream. This will write any buffered output bytes. |
| <code>void write(byte buffer[])</code> | Writes an array of bytes. This method will block until the bytes are actually written. |
| <code>void write(byte buffer[], int offset, int numBytes)</code> | Writes a sub-array of bytes. |
| <code>void write(int b)</code> | Writes a byte. This method will block until the byte is actually written. |
| <code>void writeObject(Object obj)</code> | Writes an object to the underlying storage or stream. The class that implements this interface defines how the object is written. |

Table 21.1 Some methods of ObjectOutput interface

3. *The ObjectInput interface:* ObjectInput extends the DataInput interface to include the reading objects. DataInput includes methods for the input of primitive types. ObjectInput extends that interface to include objects, arrays and Strings. Most frequently used method of this interface is the `readObject` which is used for reading state of the object from persistent media. The various methods of this interface are shown in the Table 21.2.

| Method Signature | Description |
|---|---|
| <code>int available()</code> | Returns the number of bytes that can be read without blocking. |
| <code>void close()</code> | Closes the input stream. |
| <code>int read()</code> | Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered. |
| <code>int read(byte[] b)</code> | Attempts to read up to <code>buffer.length</code> bytes into <code>buffer</code> , returning the number of bytes that were successfully read. -1 is returned when the end of the file is encountered. |
| <code>int read(byte[] b, int off, int len)</code> | Attempts to read up to <code>numByte</code> bytes into <code>buffer</code> starting at <code>buffer[offset]</code> , returning the number of bytes that were successfully read. -1 is returned when the end of the file is encountered. |
| <code>Object readObject()</code> | Reads an object from the invoking stream. |
| <code>long skip(long numBytes)</code> | Ignores (i.e., skips) <code>numBytes</code> bytes in the invoking stream, returning the number of bytes actually ignored. |

Table 21.2 Some methods of ObjectInput interface

4. *The ObjectOutputStream class:* The declaration of the class is as shown:

```
public class ObjectOutputStream extends OutputStream implements
ObjectOutput
```

The class extends `OutputStream` class and implements `ObjectOutput` interface. An `ObjectOutputStream` writes primitive data types and graphs of Java objects to an `OutputStream`. The object can be read (reconstituted) using an `ObjectInput` stream. Persistent storage of objects can be accomplished by using a file for the stream. If the stream is a network socket stream, the objects can be

reconstituted on another process. The method `writeObject` is used to write an object to the stream. Any object, including Strings and arrays, is written with `writeObject`. Multiple objects or primitives can be written to the stream. The objects must be read back from the corresponding `ObjectInputStream` with the same types and in the same order as they were written.

The default serialization mechanism for an object writes the class of the object, the class signature and the values of all non-transient and non-static fields. References to other objects (except in transient or static fields) cause those objects to be written also. Multiple references to a single object are encoded using a reference sharing mechanism so that graphs of objects can be restored to the same shape as when the original was written.

The constructor of this class is as given below:

```
public ObjectOutputStream(OutputStream out) throws IOException
```

For example, consider the following code:

```
FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();
```

The data and objects are written to the file `t.tmp` using methods of `ObjectOutputStream` class.

The most commonly used methods of this class are shown in the Table 21.3.

| Method Signature | Description |
|--|---|
| <code>void close()</code> | Closes the invoking stream. Further write attempts will generate an <code>IOException</code> . This method must be called to release any resource associated with the stream. |
| <code>void flush()</code> | Flushes the stream. This will write any buffered output bytes. |
| <code>void write(byte buffer[])</code> | Writes an array of bytes. This method will block until the bytes are actually written. |
| <code>void write(byte buffer[], int offset, int numBytes)</code> | Writes a sub-array of bytes. |
| <code>void write(int b)</code> | Writes a byte. This method will block until the byte is actually written. |
| <code>void writeBoolean(boolean b)</code> | Writes a Boolean to the invoking stream. |
| <code>void writeByte(int b)</code> | Writes a byte to the invoking stream. The byte written is the low-order byte of <code>b</code> . |
| <code>void writeBytes(String str)</code> | Writes the bytes representing <code>str</code> to the invoking stream. |
| <code>void writeChar(int c)</code> | Writes a char to the invoking stream. |
| <code>void writeChars(String str)</code> | Writes the characters in <code>str</code> to the invoking stream. |
| <code>void writeDouble(double d)</code> | Writes a double to the invoking stream. |
| <code>void writeFloat(float f)</code> | Writes a float to the invoking stream. |
| <code>void writeInt(int i)</code> | Writes an int to the invoking stream. |
| <code>void writeLong(long l)</code> | Writes a long to the invoking stream. |
| <code>final void writeObject(Object obj)</code> | Writes <code>obj</code> to the invoking stream. |
| <code>void writeShort(int i)</code> | Writes a short to the invoking stream. |

Table 21.3 Some methods of `ObjectOutputStream` interface

5. *The ObjectInputStream class:* The declaration of the class is as given below:

```
public class ObjectInputStream extends InputStream implements ObjectInput
```

The class extends `InputStream` class and implements `ObjectInput` interface. The class `ObjectInputStream` is responsible for reading objects from a stream. An `ObjectInputStream` deserializes primitive data and objects previously written using an `ObjectOutputStream`.

`ObjectOutputStream` and `ObjectInputStream` can provide an application with persistent storage for graphs of objects when used with a `FileOutputStream` and `FileInputStream`, respectively. `ObjectInputStream` is used to recover those objects previously serialized. `ObjectInputStream` ensures that the types of all object in the graph created from stream match the classes present in the Java virtual machine.

The method `readObject` is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during the serialization. When read, they need to be cast to the expected type. Primitive data types can be read from the stream using the appropriate method on **DataInput**.

The constructor of this class is as given below:

```
public ObjectInputStream(InputStream in) throws IOException
```

The constructor creates an `ObjectInputStream` that reads from the specified `InputStream`. For example, to read from a stream as written is given by the example in `ObjectOutputStream`:

```
FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
int i = ois.readInt();
String today = (String)ois.readObject();
Date date =(Date)ois.readObject();
ois.close();
```

The most commonly used methods of this class are shown in the Table 21.4.

| Method Signature | Description |
|---|---|
| <code>int available()</code> | Returns the number of bytes that are now available in the input buffer. |
| <code>void close()</code> | Closes the invoking stream. Further read attempts will generate an <code>IOException</code> . |
| <code>int read()</code> | Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered. |
| <code>int read(byte buffer[], int offset, int numByte)</code> | Attempts to read up to numBytes bytes into buffer starting at buffer [offset], returning the number of bytes successfully read. -1 is returned when the end of the file is encountered. |
| <code>boolean readBoolean()</code> | Reads and returns a Boolean from the invoking stream. |
| <code>byte readByte()</code> | Reads and returns a byte from the invoking stream. |
| <code>char readChar()</code> | Reads and returns a char from the invoking stream. |
| <code>double readDouble()</code> | Reads and returns a double from the invoking stream. |
| <code>float readFloat()</code> | Reads and returns a float from the invoking stream. |
| <code>void readFully(byte buffer[])</code> | Reads buffer.length bytes into buffer. Returns only when all bytes |

| | |
|---|---|
| void readFully(byte offset[], int offset, int numBytes) | Reads numBytes bytes into buffer starting at buffer[offset]. Returns only when numBytes have been read. |
| int readInt() | Reads and returns an int from the invoking stream. |
| long readLong() | Reads and returns an int from the invoking stream. |
| final Object readObject() | Reads and returns an object from the invoking stream. |
| short readShort() | Reads and returns a short from the invoking stream. |

Table 21.4 Some methods of ObjectOutputStream interface

```
/*PROG 21.2 DEMO OF SERIALIZATION VER 1 */

import java.util.Date;
import java.io.*;
class JPS2
{
    public static void main(String args[])
    {
        /*Serialization Process */
        try
        {
            Date d1 = new Date();
            System.out.println("\nSerializing a Data class object");
            FileOutputStream fos=new FileOutputStream("demo_serial");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(d1);
            oos.close();
        }
        catch (Exception e)
        {
            System.out.println("\nError in serialization" + e);
            System.exit(0);
        }
        /*Deserialization process */
        try
        {
            Date d2;
            FileInputStream fis=new FileInputStream("demo_serial");
            ObjectInputStream ois = new ObjectInputStream(fis);
            d2 = (Date)ois.readObject();
            ois.close();
            System.out.println("\nDe-serializing a Date class
                                object");
            System.out.println("\nDate read from file:" + d2);
        }
        catch (Exception e)
        {
            System.out.println("\nException during de-
                                serialization:"+ e);
        }
    }
}
```

```

        System.exit(0);
    }
}
}

OUTPUT:

Serializing a Data class object
De-serializing a Date class object
Date read from file:Tue Jan 20 10:45:23 PST 2009

```

Explanation: In the first part of this program, serialization is performed, and de-serialization in the second part. In the serialization part, an object `fos` of `FileOutputStream` class is first created and linked to file '`demo_create`'. This object is passed as argument to constructor of `ObjectOutputStream` class and reference of newly created object is stored in `oos`. From this object using `writeObject` method, the object `d` of `Date` class is stored. The stream is then closed.

In the de-serialization process, the same file is then opened for reading and reference of `FileInputStream` is stored in `fis`. This reference is passed as argument to the constructor of `ObjectInputStream` class and reference of newly created object is stored in `ois`. From this object using `readObject` method, the `Date` object is read which was placed during the serialization process. Note the `readObject` returns the object of type `Object` so it has to be typecasted into `Date` class.

The important thing to note in the program all Java built-in classes implement is the interface `Serializable`; this being the reason that they can be serialized and deserialized.

```

/*PROG 21.3 DEMO OF SERIALIZATION VER 2 */

import java.io.*;
class JPS3
{
    public static void main(String args[])
    {
        /*Serialization Process */
        try
        {
            Person p1 = new Person("Hari", 25);
            System.out.println("\nSerializing a Person class
                                object");
            FileOutputStream fos;
            fos = new FileOutputStream("demo_serial");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(p1);
            oos.flush();
            oos.close();
        }
        catch (Exception e)
        {
            System.out.println("\nError in serialization");
            System.exit(0);
        }
        /*Deserialization Process */
        try

```

```

    {
        Person p2;
        FileInputStream fis=new FileInputStream("demo_serial");
        ObjectInputStream ois = new ObjectInputStream(fis);
        p2 = (Person)ois.readObject();
        ois.close();
        System.out.println("\nDe-serializing a Person class
                           object");
        System.out.println("\nPerson object read from file\n"
                           +p2);
    }
    catch (Exception e)
    {
        System.out.println("\nError is de-serialization");
        System.exit(0);
    }
}
class Person implements Serializable
{
    String name;
    int age;
    Person(String s, int a)
    {
        name = s;
        age = a;
    }
    public String toString()
    {
        return "\nName =" + name + "\t" + "Age=" + age;
    }
}

```

OUTPUT:

```

Serializing a Person class object
De-serializing a Person class object
Person object read from file
Name = Hari Age = 25

```

Explanation: In this program, a Person class is created which implements Serializable interface. The Person class has two data members: **name** and **age**. They are initialized to data using constructor of the Person class. In the serialization process, the object p1 is stored to the file. The same object is de-serialized and read back in the second part. The Person class also overrides the **toString** method so that an object can be treated as a string. The method is called automatically when an object of Person class is displayed using **System.out.println**.

21.3 RMI

The Java remote method invocation (RMI) application programming interface (API) enables client and server communications over the Internet. Remote method invocation allows applications to call object methods located remotely, sharing resources and processing load across systems. Unlike other systems for

remote execution which require that only simple data types and defined structures be passed to and from methods, RMI allows any Java object type to be used, even if the client or server has never encountered it before. RMI allows both client and server to dynamically load new object types as required.

Remote method invocation (RMI) facilitates object function calls between Java virtual machines (JVMs). JVM can be located on separate computers, yet one JVM can invoke methods belonging to an object stored in another JVM. Methods can even pass objects that a foreign virtual machine has never encountered before, allowing dynamic loading of new classes as required.

21.3.1 Writing RMI Service

Writing RMI services can be little difficult at first, so to start off, a very small example is given. A RMI server will be written which will return maximum of two double numbers. The various steps in the development of a RMI service are as follows:

- 1. Writing an interface:** The first thing that needs to be done is to agree upon an interface. An interface is a description of the methods remote clients will be allowed to invoke. The method signature will be as follows:

```
double maxtwo(double a, double b);
```

Once it is decided, that on which method will compose a service, a Java interface will be created. An interface is a method which contains abstract methods; these methods must be implemented by another class. Here is the source code for a service that calculates maximum of two double numbers. Save it under file name **imax 2.java** in a directory name **server**.

```
import java.rmi.*;
public interface imax2 extends Remote
{
    double maxtwo(double a, double b) throws RemoteException;
}
```

The interface name is **imax2** and it must extend **java.rmi.Remote**, which indicates that this is a remote service. The **Remote** interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a ‘remote interface’, an interface that extends **java.rmi.Remote**, are available remotely.

In the interface **imax2**, method declaration is provided for a method and the interface is complete. The next step is to implement the interface, and provide methods for the square and power functions.

- 2. Implementing the interface:** In the implementation part, a class needs to be written which will be implementing the interface created in the first step. The class is responsible for providing the definitions of the methods declared in interface. In writing this class, the real code need to be concerned about is the default constructor. Assume the class name is **Max2Class**. Its constructor must be defined as:

```
public Max2Class() throws RemoteException {}
```

A default constructor has to be declared, even when there is no initialization code for a service. This is because a default constructor can throw a **java.rmi.RemoteException** from its parent constructor in **UnicastRemoteObject** (discussed shortly).

The implementation of the interface is given as:

```
public double maxtwo(double a, double b) throws RemoteException
{
    return a>b?a:b;
}
```

The complete source code of the file Max2Class.java is given below. Save it also in the directory server.

```
import java.rmi.*;
import java.rmi.server.*;
public class Max2Class extends UnicastRemoteObject
implements imax2
{
    public Max2Class() throws RemoteException
    {
    }
    public double maxtwo(double a,double b) throws
    RemoteException
    {
        return a>b?a:b;
    }
}
```

Note the interface class must extend the `UnicastRemoteObject` class. RMI provides some convenience classes that remote object implementations can extend, which facilitate remote object creation. The class `UnicastRemoteObject` is one of them. The class is used for exporting a remote object and obtaining a stub that communicates to the remote object.

After creating the above class, now create a server class which will act as a RMI server. The code for this class is given below. Save the file under the name **Max2Server.java** in the server directory.

```
import java.net.*;
import java.rmi.*;
public class Max2Server
{
    public static void main(String[] args)
    {
        try
        {
            Max2Class ref = new Max2Class();
            Naming.rebind("max2ser", ref);
        }
        catch (Exception e)
        {
            System.out.println("Exception:" + e);
        }
    }
}
```

The crux of the code is the two statements as rewritten below:

```
Max2Class ref = new Max2Class();
Naming.rebind("max2ser", ref);
```

Reference `ref` is of the class `Max2Class` created earlier. `Naming` is the class in `java.rmi` package. Its declaration is as follows:

```
public final class Naming extends Object
```

The `Naming` class provides method for storing and obtaining references to remote objects in a remote object registry. Each method of the `Naming` class takes as one of its arguments a name that is a `java.lang.String` in URL format (without the scheme component) of the form:

```
//host: port/name
```

In this case, `host` is the host (remote or local) where the registry is located, `port` is the port number on which the registry accepts calls and `name` is a simple string uninterpreted by the registry. Both `host` and `port` are optional. If `host` is omitted, the host defaults to the local host. If `port` is omitted, the port default to 1099, the ‘well-known’ port that RMI’s registry, `rmiregistry`, uses (discussed later).

Binding a name for a remote object is associating or registering a name for a remote object that can be used at a later time to look up that remote object. A remote object can be associated with a name using the `Naming` class’s `bind` or `rebind` method.

Once a remote object is registered (bound) with RMI registry on the local host, callers on a remote (or local) host can look up the remote object by name, obtain its reference and can then invoke remote methods on the object.

In the `rebind` method for the first argument is `name` through which reference object will be rebinding. Any existing binding for the name is replaced; second object is the actual reference object.

3. *Implementing the client:* What good is a service, if a client that uses it is not written? Writing clients is the easy part—all clients have to do is call the registry to obtain a reference to the remote object, and call its methods. All the underlying network communication is hidden from view, which makes RMI clients simple.

The client receives an instance of the interface defined earlier, not the actual implementation. Some behind-the-scenes work goes on, but this is completely accessible to the client.

```
String url = "rmi://127.0.0.1/max2ser";
imax2 mi =(imax2) Naming.lookup(url);
```

To identify a service, an RMI URL is specified. The URL contains the hostname on which the service is located, and the logical name of the service. This returns an `imax2` instance, which can then be used just like a local object reference. The methods can be called just as if an instance was created of the remote **Max2Server**.

```
//call remote method
System.out.println("Maximum: "+mi.maxtwo(20.4,23.4));
```

The full source code of the client is given below. Save the code under file name `Max2Client.java` under directory `client`.

```
import java.rmi.*;
public class Max2Client
{
    public static void main(String args[])
    {
        try
        {
            String url = "rmi://127.0.0.1/max2ser";
            imax2 mi = (imax2) Naming.lookup(url);
            System.out.println("Maximum is:"+mi.maxtwo(10.5,20.5));
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e);
        }
    }
}
```

4. *Running the application:* Running the application requires some necessary steps to be carried out.

They are discussed below:

- *Generation of stub and skeletons:* In the context of RMI, a stub is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method

calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine. An object passed as an argument to a remote method call must be serialized and sent to the remote machine.

A skeleton is a Java object that resides on the server machine. It works with the other RMI system to receive requests, performs de-serialization and invokes the appropriate code on the server.

To generate stubs and skeletons, a tool called the RMI compiler is used, which is invoked from the command line, as shown here, into the server directory:

```
rmic Max2Class
```

This command generates two new files: `Max2Class_Skel.class` (skeleton) and `Max2Class_Stub.class` (stub).

- *Install files on client and server machines:* Onto the server directory, the following files must be present: `imax2.class` (interface class file), `Max2Server.class` (the server class), `Max2Class.class` (interface implemented class), `Max2Class_Skel.class` (Skelton) and `Max2Class_Stub.class` (**stub**).

Onto the client directory, the following files must be present: `imax2.class` (interface class file), `Max2Client` (the client file) and `Max2Class_Stub.class` (stub).

- *Starting RMI registry:* The JDK provides a program called `rmiregistry`, which executes on the server machine. It maps names to object reference. Start the RMI registry from the command line as shown here:

```
start rmiregistry
```

When this command returns, one should see that a new window has been created. Let this window be opened till one is not finished working with RMI. It can be minimized and let it call at the task bar.

- *Running server and client:* Move onto the server directory and start the server in a separate window as:

```
java Max2Server
```

Now move onto the client directory and start the client in a separate window as:

```
java Max2Client
```

Output obtained will look like this:

```
Maximum is 20.5
```

21.4 PONDERABLE POINTS

1. The Java native interface (JNI) is a powerful feature of the Java platform. Applications that use the JNI can incorporate native code written in programming languages such as C and C++, as well as in the Java programming language.
2. JNI is useful for a Java application to communicate with native code that resides in the same process.
3. Serialization is a process by which the object writes a record of itself to a persistent storage medium, such as a disk file, and reloads itself by reading the record back. Serialization is the process of converting a set of object instances that contains references to each other into a process stream of bytes, which can then be sent through a socket, stored to a file or simply manipulated as a stream of data.
4. To serialize an object, create an instance of `ObjectOutputStream` and call the `writeObject()` method; to read in a serialized object, create an instance of `ObjectInputStream` and call the `readObject()` object.
5. The basic serialization mechanism is implemented by interfaces `Serializable`, `ObjectOutput` and `ObjectInput`, and classes `ObjectInputStream` and `ObjectOutputStream`.

6. The Java remote method invocation (RMI) application programming interface (API) enables client and server communications over the net. Remote method invocation, allowing applications to call object methods located remotely, sharing resources and processing load across systems.
7. The RMI API is implemented in Java using `java.rmi` package.
8. The `Remote` interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a ‘remote interface’, an interface that extends `java.rmi.Remote`, are available remotely.
9. The class `UnicastRemoteObject` is one of them. The class is used for exporting a remote object and obtaining a stub that communicates to the remote object.
10. The `Naming` class provides methods for storing and obtaining references to remote objects in a remote object registry.
11. Binding a name for a remote object is associating or registering a name for a remote object that can be used at a later time to look up that remote object. A remote object can be associated with a name using the `Naming` class's `bind` method.
12. In the context of RMI, a stub is a Java object that resides on the client machines. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub.
13. A skeleton is a Java object that resides on the server machine. It works with the other RMI system to receive requests, perform de-serialization and invoke the appropriate code on the server.

REVIEW QUESTIONS

1. What is JNI? Explain with example.
2. How does JNI support the working of Java Virtual Machine? Justify your answer with an example.
3. What are the limitations of JNI? How is Java Native Interface useful for virtualization like XEN, VMware, etc?
4. Explain the term 'serialization' with suitable example. Also explain the usage of serialization.
5. Explain the principal of Remote Method Invocation with block diagram.
6. What are stub and skeletons?
7. What is rmiregistry and its significance?
8. Compare and contrast distributed and non-distributed Java programs.
9. Explain the implementation of the server, client and interface classes?
10. Develop a Java class for marks list processing using RMI.
11. Write an employee class in which `pay()` is a method which calculates the gross pay from the basic pay. Write an application which runs this method remotely using RMI.
12. Which tool generates stub and the skeleton?
13. Which function returns a list of URLs that are currently known to the RMI registry?

Multiple Choice Questions

1. JNI lets Java code use code and code libraries written in

| | |
|-----------------|-----------------------|
| (a) Java | (c) Other language |
| (b) Java script | (d) None of the above |
2. Microsoft's proprietary implementation of a Java Virtual Machine (Visual J++) had a similar mechanism for calling native Windows code from Java, called the

| |
|----------------------------------|
| (a) Java Virtual Interface (JVI) |
| (b) Raw Native Interface (RNI) |
| (c) Java Visual Interface (JVI) |
| (d) None of the above |
3. Java Native Access provides Java programs easy access to native shared libraries without writing

| | |
|------------------------|-----------------------|
| (a) Standard libraries | (c) Boiler plate code |
| (b) Libraries | (d) None of the above |
4. A JNI interface pointer _____ is passed as an argument for each native function mapped to a Java method

| | |
|--------------------------|-------------------------|
| (a) <code>JNIENV*</code> | (c) <code>JNIEnv</code> |
| (b) <code>JNIEnv*</code> | (d) None of the above |

5. Serializability of a class is enabled by the class implementing the
 - (a) java.io.Serializable interface
 - (b) java.util.Serializable interface
 - (c) java.awt.Serializable interface
 - (d) None of the above
6. Which method used to write an object to the underlying storage or stream. The class that implements this interface how the object is written
 - (a) void write (int b)
 - (b) void write (int a, int b)
 - (c) void writeObject (Object obj)
 - (d) public void writeObject (Object obj)
7. Which of the following is used to extend the DataInput interface to include the reading of objects?
 - (a) ObjectExtend
 - (b) ObjectInputExtend
8. int available is used to return
 - (a) String available in input buffer
 - (b) Integer available in input buffer
 - (c) bytes available in input buffer
 - (d) None of the above
9. A skelton is a Java object that resides on
 - (a) Server machine
 - (b) Client machine
 - (c) Both server and client machine
 - (d) None of the above
10. A stub is a Java object that resides on
 - (a) Server machine
 - (b) Client machine
 - (c) Both server and client machine
 - (d) None of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. c 2. b 3. c 4. b 5. a 6. c 7. d 8. c 9. a 10. b

22

Working with Images

22.1 INTRODUCTION

An image is simply a rectangular graphical object. It is simply a 2D array of pixels. To a computer, an image is just a set of numbers. The numbers specify the colour of each pixel in the image. The numbers representing the image on the computer's screen are stored in a part of memory called a frame buffer. Many times each second, the computer's video card reads the data in the frame buffer and colours each pixels on the screen according to that data. Whenever the computer needs to make some change to the screen, it writes some new numbers to the frame buffer, and the change appears on the screen a fraction of a second later, the next time the screen is redrawn by the video card.

Since it is just a set of numbers, the data for an image does not have to be stored in a frame buffer. It can be stored elsewhere in the computer's memory. It can be stored in a file on the computer's hard disk. Just like any other data file, an image file can be downloaded over the Internet. Java includes standard classes and subroutines that can be used to copy image data from one part of the memory to another, to get data from an image file and use it to display the image on the screen.

The standard class `java.awt.Image` is used to represent images. Images are objects of the `Image` class. The abstract class `Image` is the super class of all classes that represent graphical images. The package `java.awt.image` contains large number of classes and interfaces for the manipulation or processing of images. The package contains around 8 interfaces and 40 classes for processing of images. A few of them will be discussed.

22.2 DRAWING AN IMAGE

Every image is coded as a set of numbers, but there are various ways in which the coding can be done. For images in file, there are two main coding schemes which are used in Java and on the Internet. One is used for GIF images, which are usually stored in files that have names ending in 'gif'. The other is used for JPEG image, which are stored in files that have names ending in '.jpg' or '.jpeg'.

The `Applet` class defines a method, `getImage`. It can be used for loading images stored in GIF and JPEG files and possibly in other types of image files, depending on the version of Java. It has got two forms:

1. **`public Image getImage(URL url)`**

The method returns an `Image` object that can then be painted on the screen. The `url` that is passed as an argument must specify an absolute URL.

2. **`public Image getImage(URL url, string name)`**

The method returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

Assume the following code in the applet class:

```
Image img = getCodeBase(), "ace.gif";
```

The method gets the base URL. This is the URL of the directory which contains this applet. For example, if the 'ace.gif' is in the directory C:/JPS, getCodeBase() returns 'file:\C:\JPS'.

Once an object of type `Image` is obtained, the image can be drawn in any graphics context. Suppose that `g` is a graphics context, that is, an object belonging to the class `Graphics`, and suppose that `img` is a variable of type `Image`. Then the usual command for drawing the image `img` in the graphics context `g` is:

```
g.drawImage(img, x, y, this);
```

The signature of the `drawImage` is shown below:

```
boolean drawImage(Image img,int x,int y,ImageObserver observer)
```

The parameters `x` and `y` are integers that give the position of the top-left corner of the displayed image. The fourth parameter requires some explanation, but it will be given in the next section.

The other form of the `drawImage` method is:

```
boolean drawImage(Image img, int x, int y, int width, int height,
ImageObserver observer)
```

The method draws as much of the specified image as has already been scaled to fit inside the specified rectangle. The image is drawn inside the specified rectangle of this graphics context's coordinate space, and is scaled if necessary.

Now getting all necessary concepts, a first program for displaying a simple image is attempted as follows:

```
/*PROG 22.1 LOADING AN IMAGE USING GETIMAGE VER 1*/
/*
<html>
<applet>
<applet code="image1" width=200 height = 200>
</applet>
</html>
*/
import java.awt.*;
import java.applet.*;
public class image1 extends Applet
{
    public void paint(Graphics g)
    {
        Image img = getImage(getCodeBase(), "ac.gif");
        g.drawImage(img, 10, 10, this);
    }
}
OUTPUT:
```

Explanation: This program is simple to understand. Instead of 'ac.gif', any of '.jpeg', '.gif', '.bmp' or any graphical image can be chosen which the Java may support. One important thing to note here is that if the specified file is not found, no error is shown; simply a blank applet window is displayed.

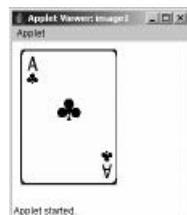


Figure 22.1 Output screen of Program 22.1

```
/*PROG 22.2 LOADING AN IMAGE USING GETIMAGE VER 2 */  
  
/*  
<html>  
<applet>  
<applet code="image2" width=200 height=200>  
</applet>  
</html>  
*/  
import java.awt.*;  
import java.applet.*;  
import java.net.*;  
public class image2 extends Applet  
{  
    public void paint(Graphics g)  
    {  
        try  
        {  
            URL url = new URL(getCodeBase() + "\\\gau4a.jpg");  
            Image img = getImage(url);  
            g.drawImage(img, 30, 30, 200, 200, this);  
        }  
        catch (Exception e)  
        {  
            g.drawString("Image not found or error", 20, 40);  
        }  
    }  
}  
OUTPUT:
```

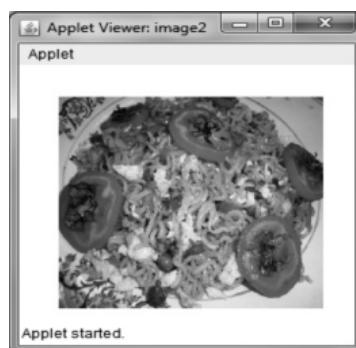


Figure 22.2 Output screen of Program 22.2

Explanation: In this program, a URL url instance is created using getCodeBase () and image file name. This time a variation of the drawImage method is used in which the image is confined within the specified width and height (i.e., 200). Rest is simple to understand. This method is especially useful in case of having limited space for displaying the image.

22.3 THE IMAGEOBSERVER INTERFACE

ImageObserver is an updated interface for receiving notification about image information as the image is constructed. The fourth parameter was not defined in the `drawImage` method above. This is explained in the following section.

The parameter is required in the `drawImage` method because of the way that Java works with images. When `getImage()` is used to create an `Image` object from an image file, the file is not downloaded immediately. The `Image` object simply remembers where the file is. The file will be downloaded first time when the image is drawn. However, when the image needs to be downloaded, the `drawImage()` method only initiates the downloading. It does not wait for the data to arrive. So, after `drawImage()` has finished executing, it is quite possible that the image has not actually been drawn. But then, when does it get drawn? That is where the fourth parameter to the `drawImage()` command comes in. The fourth parameter is something called an `ImageObserver`. After the image has been downloaded, the system will inform the `ImageObserver` that the image is available, and the `ImageObserver` will actually draw the image at that time. (For large images, it is even possible that the image will be drawn in several parts as it is downloaded.) Any `Component` object can act as an `ImageObserver`, including applets and canvases. In “`g.drawImage(int x, y, this);`” the special variable “`this`” refers to the object whose source code is being written. When an image is drawn to the screen, ‘`this`’ should almost always be used, as the fourth parameter to `drawImage()`.

As the image is loaded from the network computer, the other work can be done side by side so that a user does not notice the delay or is informed about the progress of the download.

The interface defines just one method.

```
boolean imageUpadte(Image img, int infoflags, int x, int width, int height)
```

This method is called when information about an image which was previously requested using an asynchronous interface becomes available. Asynchronous interfaces are method calls such as `drawImage(img, x, y, ImageObserver)` which takes an `ImageObserver` object as an argument.

This method should return true, if further updates are needed, or false, if the required information has been acquired. The image which was being tracked is passed using the `img` argument. Various constants are combined to form `infoflags` argument which indicates what information about the image is now available. The interpretation of the `x`, `y`, `width` and `height` arguments depends on the contents of the `infoflags` argument.

The `infoflags` argument should be the bitwise inclusive OR of the following flags: `WIDTH`, `HEIGHT`, `PROPERTIES`, `SOMEBITS`, `FRAMEBITS`, `ALLBITS`, `ERROR` and `ABORT`. A brief description of flags is given in Table 22.1.

| Flag | Description |
|----------------------|--|
| <code>ABORT</code> | Indicates that an image which was being tracked asynchronously was aborted before production was complete. If the <code>ERROR</code> flag also was not set in this image update, accessing any of the data in the image would restart the production again, probably from the beginning. |
| <code>ALLBITS</code> | Indicates that a static image which was previously drawn is now complete. The <code>x</code> , <code>y</code> , <code>width</code> and <code>height</code> arguments are ignored. |
| <code>ERROR</code> | Indicates that an image which was being tracked asynchronously has encountered an error. No further information will become available and drawing the image will fail. As a convenience, the <code>ABORT</code> flag will be indicated at the same time to indicate that the image production was aborted. |

(Continued)

| | |
|------------|---|
| FRAMEBITS | Indicates that another complete frame of multi-frame image which was previously drawn is now available to be drawn again. The x, y, width and height arguments are ignored. |
| PROPERTIES | Indicates that the properties of the image are now available and can now be obtained using getProperty method. |
| SOMEBITS | Indicates that more pixels needed for drawing the image are available. The bounding box of the new pixels can be taken from the x, y, width and height arguments. |
| WIDTH | Indicates that the width of the base image is now available. |
| HEIGHT | Indicates that the height of the base image is now available. |

Table 22.1 Methods of ImageObserver interface

The default **Component** class image observer used in previous examples called `repaint()` for use each time a new section of the image was available, so that the screen was updated more or less continuously as the data arrived. The **Applet** class has an implementation of the `imageUpdate()` method for the **ImageObserver** interface that is used to repaint images as they are loaded.

22.3.1 Why Override ImageUpdate?

There are several reasons behind overriding `imageUpdate()`:

1. The default implementation of the method repaints the entire image, each time any new data arrives. This causes flashing between the background colour and the image.
2. When using `getImage` method, if file does not exist, no warning or error of any type is displayed, that is, one will not come to know about missing files.
3. The defaults that repaint implementation cause the system to only repaint the image every tenth of a second or so. This causes a jerky, uneven feel as the image paints.

```
/*PROG 22.3 DEMO OF IMAGEOBSERVER VER 1*/
import java.awt.*;
import java.applet.*;
public class image3 extends Applet
{
    Image img;
    boolean err = false;
    String imgname = "gaulv.jpeg";
    public void init()
    {
        setBackground(Color.blue);
        img = getImage(getDocumentBase(), imgname);
    }
    public void paint(Graphics g)
    {
        if (err)
            g.drawString("Image not found:"+imgname,50,40);
        else
            g.drawImage(img, 0, 0, this);
    }
}
```

```

public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h)
{
    if((flags & ABORT) !=0)
    {
        err = true;
        repaint();
    }
    return(flags & (ALLBITS|ABORT) )==0;
}
}

```

OUTPUT:



Figure 22.3 Output screen of Program 22.3

Explanation: In this program, an image file name is intentionally chosen that does not exist. When image is being loaded, `imageUpdate` is called as the default observer is always working. In the method binary ANDing of flags parameter and ABORT constant is done. If it is not equal to 0, it indicates that the file does not exist. Then `err` is set to `true` in this situation and `repaint` is called.

```

/*PROG 22.4 DEMO OF IMAGEOBSERVER VER 2 */

import java.awt.*;
import java.awt.image.*;
import java.applet.*;
public class MyObserver extends Applet implements
ImageObserver
{
    Image img;
    public boolean imageUpdate(Image image, int flags, int
                               x, int y, int width, int height)
    {
        if ((flags & EIGHT) !=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));
    }
}

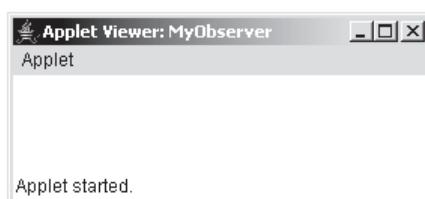
```

```

        if ((flags & ALLBITS) != 0)
        {
            System.out.println("Image finished!");
            return false;
        }
        if ((flags & ABORT) != 0)
        {
            System.out.println("Image load aborted...");
            return false;
        }
        return true;
    }
    public void init()
    {
        img = getImage(getDocumentBase(), "jc.gif");
        MyObserver mo = new MyObserver();
        img.getWidth(mo);
        img.getHeight(mo);
    }
}

```

OUTPUT:



(a) Applet for MyObserver.java

```

C:\WINDOWS\system32\cmd.exe - appletviewer MyObserver.html
C:\JPS\ch22>javac MyObserver.java
C:\JPS\ch22>appletviewer MyObserver.html
Warning: <applet> tag requires code attribute.
Image height =170
Image width =120
Image section :java.awt.Rectangle[x=0,y=0,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=1,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=2,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=3,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=4,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=5,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=6,width=120,height=1]
Image section :java.awt.Rectangle[x=0,y=7,width=120,height=1]

```

(b) Console showing the results of the image for MyObserver.java

Explanation: The class MyObserver implements ImageObserver interface and overrides imageUpdate method. In the overridden imageUpdate method, it uses all constants in the if conditions and displays the status onto the console. In the init method when the two lines execute:

```

img.getWidth(mo);
img.getHeight(mo);

```

`imageUpdate` method is called and output is shown. The applet window is not shown as it is simply a blank window. The output shows the width and height of the image file ‘jc.gif’. From third line onwards, output appears due to the following line in the `imageUpdate` method:

```
if ((flags & SOMEBITS) != 0)
    System.out.println("Image section :" + new Rectangle(x, y,
        width, height));
```

Our image observer reads one section of the image at a time starting from $x = 0$, $y = 0$, $width = 16$ and $height = 1$. Only the y parameters vary from 0 to 15 ($height - 1$).

When the image is loaded in the end, the “**Image finished!**” is displayed.

22.4 DOUBLE BUFFERING

In addition to image in image files, objects of type `Image` can be used to represent images stored in the computer's memory. What makes such images particularly useful is that it is possible to draw to an image in the computer's memory. This drawing is not visible to the user. Later, however, the image can be copied very quickly to the screen. If this technique is used for repainting the screen behind the scenes, then in memory, an old image is erased and a new one is drawn step by step. This takes some time. If all these drawings were done on screen, the user would see the image flicker. Instead, a complete new image replaces the old one on the screen almost instantaneously. The user does not see all the steps involved in redrawing. This technique can be used to do smooth, flicker-free animation and dragging.

An image in memory can be called an off-screen canvas/image. The technique of drawing to an off-screen canvas and then quickly copying the canvas to the screen is called double buffering. The use of an off-screen image to reduce flicker is called double buffering, because the screen is considered a buffer for pixels, and the off-screen image is the second buffer, where pixels can be prepared for display.

The name comes from the term ‘frame buffer’, which refers to the region in memory that holds the image on the screen. (In fact, true double buffering uses two frame buffers. The video card can display either frame buffer on the screen and can switch instantaneously from one frame buffer to the other. One frame buffer is used as an off-screen canvas to prepare a new image for the screen. Then the video card is told to switch from one frame buffer to the other. Now copying of memory is involved. Double buffering, as it is implemented in Java, does require copying, which takes some time, and is not entirely flicker free.)

An off-screen image can be created by calling the instance method `createImage()`, which is defined in the `Component` class. This method can be used in applets and canvases, for example. The `createImage()` method takes two parameters to specify the width and height of the image to be created. For example:

```
Image OSC = createImage(width, height);
```

Drawing to an off-screen canvas is done in the same way as any other drawing in Java, by using a graphics context. The `Image` class defines an instance method `getGraphics()` that returns a `Graphics` object that can be used for drawing on the off-screen canvas. (This works only for off-screen canvases. If one tries to do this with an image from a file, an error will occur.) That is, if `OSC` is a variable of type `Image` that refers to an off-screen canvas, it can be stated as:

```
Graphics offscreenGraphics = OSC.getGraphics();
```

Then, any drawing operations performed with the graphics context off-screen graphics are applied to the off-screen canvas. For example, `'offscreenGraphics.drawRect (10, 10, 50, 100);'` will draw a 50-by-100 pixel rectangle on the off-screen canvas. Once a picture has been drawn on the off-screen canvas, it can be copied into another graphics context, `g`, using the method `g.drawImage (OSC)`.

The following code creates an off-screen image over a canvas using `createImage`, then getting its graphics context and filling the entire image area in blue colour.

```
Canvas OSC = new Canvas();
Image OSI = OSC.createImage(200,200);
Graphics gc = OSI.getGraphics();
gc.setColor(Color.blue);
gc.fillRect(0,0,200,200);
```

A small example shows how to implement double buffering in Java:

```
/*PROG 22.5 DEMO OF DOUBLE BUFFERING VER 1 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class DoubleBuffer extends Applet
{
    Image buffer = null;
    int w, h;
    public void init()
    {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
        buffer = createImage(w, h);
    }
    public void paint(Graphics g)
    {
        Graphics screengc = null;
        screengc = g;
        g = buffer.getGraphics();
        int i;
        for (i = 0; i < h / 3; i++)
        {
            g.setColor(new Color(0, 0, (i * 3) % 255));
            g.drawLine(0, i, w, i);
        }
        for (i = h / 3; i < 2 * h / 3; i++)
        {
            g.setColor(new Color(0, (i * 3) % 255, 0));
            g.drawLine(0, i, w, i);
        }
        for (i = 2 * h / 3; i < h; i++)
        {
            g.setColor(new Color((i * 3) % 255, 0, 0));
            g.drawLine(0, i, w, i);
        }
        screengc.drawImage(buffer, 0, 0, null);
    }
    public void update(Graphics g)
```

```

    {
        paint(g);
    }
}

```

OUTPUT:

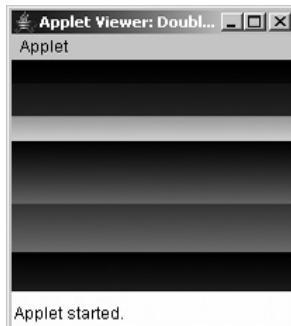


Figure 22.4 Output screen of Program 22.5

Explanation: In this program, a reference buffer of Image type is created. In the `init` method, the dimensions of applet window are stored into `w` and `h` parameters. The off-screen image is created using `createImage` method with dimension `w` and `h`. The returned `Image` instance is assigned to `buffer`.

In the `paint` method, the region of applet window is divided into three parts and each part is painted into three different colours. In the beginning of the method, the graphics context is stored to applet window into `screengc` using the statement:

```
screengc = g;
```

In the next line, graphics context is obtained for an off-screen image buffer using `getGraphics` method of `Image` class as:

```
g = buffer.getGraphics();
```

In the code that follows, all drawing is done onto the off-screen buffer and nothing is visible onto the screen. After all three for loops, the complete drawing is stored into the off-screen buffer. In the last line, the image is copied from off-screen buffer onto the screen as:

```
screengc.drawImage(buffer, 0, 0, null);
```

The last parameter can be null as we image is drawn into the memory and we are not getting it from a network or from a file onto the local machine, so no need of notification.

```
/*PROG 22.6 DEMO OF DOUBLE BUFFERING VER 2 */
```

```

/*
<html>
<applet>
<applet code="WDoubleBuffer" width=200 height=200>
</applet>
</html>
*/
import java.awt.*;

```

```

import java.awt.event.*;
import java.applet.*;
public class WDoubleBuffer extends Applet
{
    int w, h;
    int x, y;
    Image buffer = null;
    public void init()
    {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
        buffer = createImage(w, h);
        addMouseListener(new MMA());
    }
    class MMA extends MouseAdapter
    {
        public void mousePressed(MouseEvent me)
        {
            x = me.getX();
            y = me.getY();
            repaint();
        }
    }
    public void paint(Graphics g)
    {
        Graphics gr = buffer.getGraphics();
        gr.drawString("WELCOME", x, y);
        g.drawString("HELLO", x, y);
        g.drawImage(buffer, 0, 0, null);
    }
    public void update(Graphics g)
    {
        paint(g);
    }
}

```

Explanation: In general, any drawing or painting done on the applet window will disappear if the applet is covered up and then uncovered. Double buffering can be used to solve this problem. The idea is simple: keep a copy of the drawing in an off-screen canvas. When the component needs to be redrawn, copy the off-screen canvas onto the screen. Same has been employed in the program. The word 'HELLO' is written to the off-screen image buffer whenever the mouse is clicked in the applet window and same is displayed as image as the last line of the paint method. All 'HELLO' are written to the off-screen image and next to the window. Now, whenever applet window is covered and then uncovered, paint method is called and buffered image is shown onto the screen.

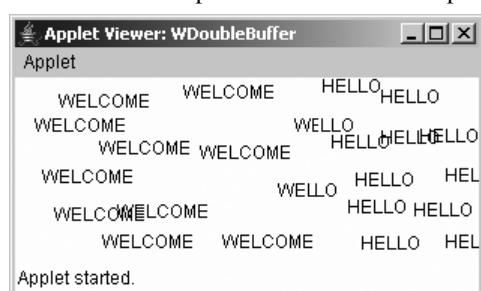


Figure 22.5 Output screen of Program 22.6

22.5 THE MEDIATRACKER CLASS

`java.awt.MediaTracker` is a utility class that simplifies the process if one were to wait for one or more images to be loaded before they are displayed. A `MediaTracker` monitors the preparation of an image or a group of images and lets one check on them periodically, or wait until they are completed. `MediaTracker` uses the `ImageObserver` interface internally to receive image update.

To use a media tracker, create an instance of `MediaTracker` and call its `addImage` method for each image to be tracked. In addition, each image can be assigned a unique identifier. This identifier controls the priority order in which the images are fetched. It can also be used to identify unique subsets of the images that can be waited on independently. Images with a lower ID are loaded in preference to those with a higher ID number.

The `addImage` method has two overloaded forms:

1. **`public void addImage(Image image, int id)`**

The method adds an image to the list of images being tracked by this media tracker. The `id` specifies an identification number for tracking the image. It should not necessarily be unique. The same number can be used for tracking with several images as a means of identifying them as part of a group.

2. **`public void addImage(Image image, int id, int w, int h)`**

This form of method is similar to previous one but here width and height of the image is also specified.

The other useful methods of the `MediaTracker` class are discussed below:

1. **`public boolean checkID(int id)`**

This method checks to see if all images tracked by this media tracker that are tagged with the specified identifier have finished loading or it can be waited for to be loaded completely. If there is an error while loading or scaling an image, that image is considered to have finished loading.

2. **`public boolean isErrorID(int id)`**

This method checks the error status of all the images tracked by this media tracker with the specified identifier. It returns true if any of the images with the specified identifier had an error during loading; false otherwise.

3. **`public boolean checkAll()`**

This method checks to see if all images being tracked by this media tracker have finished loading and returns true else returns false.

4. **`public void waitForID(int id) throws InterruptedException`**

This method starts loading all images tracked by this media tracker with the specified identifier. The method waits until all the images with the specified identifier have finished loading.

5. **`public boolean waitForID(int id, long ms) throws InterruptedException`**

In this form the time is also specified. This method waits until all the images with the specified identifier have finished loading, or until the length of time specified in milliseconds by the `ms` argument has passed.

```
/*PROG 22.7 DEMO OF MEDIATRACKER CLASS VER 1 */
```

```
import java.awt.*;
import java.applet.*;
/*
<html>
<applet>
<applet code="mediatracker" width=200 height=200>
</applet>
```

```

</html>
*/
public class mediatracker extends Applet
{
    Image image;
    public void init()
    {
        MediaTracker tracker = new MediaTracker(this);
        image = getImage(getDocumentBase(), "gauv.jpg");
        tracker.addImage(image, 0);
        try
        {
            tracker.waitForAll();
        }
        catch (InterruptedException ex)
        {
            image = null;
        }
    }
    public void paint(Graphics g)
    {
        g.drawImage(image, 10, 10, this);
    }
}

```



Figure 22.6 Output screen of Program 22.7

Explanation: This program is very simple. An instance tracker of `MediaTracker` class is created. The constructor takes default component `this` as argument. The image is added for the tracking purpose using `addImage` method of `MediaTracker` class. The id is assumed to be 0. The method `waitForAll` starts loading all images tracked by this media tracker. This method waits until all the images being tracked have finished loading. When image is loaded, it is displayed in the `paint` method.

```
/*PROG 22.8 DEMO OF MEDIATRACKER CLASS VER 2 */
```

```

import java.awt.*;
import java.applet.*;
/*

```

```
<html>
<applet>
<applet code="mediatracker1" width=200 height=200>
</applet>
</html>
*/
public class mediatracker1 extends Applet implements Runnable
{
    Image img;
    final int MAIN_IMAGE = 0;
    MediaTracker tracker;
    boolean show = false;
    Thread runme;
    String message = "Loading....";
    public void init()
    {
        img = getImage(getDocumentBase(), "gauv2.jpg");
        tracker = new MediaTracker(this);
        tracker.addImage(img, MAIN_IMAGE);
    }
    public void start()
    {
        if (!tracker.checkID(MAIN_IMAGE))
        {
            runme = new Thread(this);
            runme.start();
        }
    }
    public void stop()
    {
        runme.stop();
        runme = null;
    }
    public void run()
    {
        repaint();
        try
        {
            tracker.waitForID(MAIN_IMAGE);
        }
        catch (InterruptedException e)
        {
            message = "Error";
        }
        if (tracker.isErrorID(MAIN_IMAGE))
            message = "Error";
        else
            show = true;
        repaint();
    }
    public void paint(Graphics g){
```

```

        if (show)
            g.drawImage(img, 0, 0, this);
        else
        {
            g.drawRect(0, 0, getSize().width - 1,
                       getSize().height - 1);
            g.drawString(message, 20, 20);
        }
    }
}

```

Explanation: This program is the modified version of the previous one where a separate thread has been used for taking the image. In the `init()` method, `LoadMe` requests its image and creates a `MediaTracker` to manage it. Later, after the applet is started, `LoadMe` starts up a thread to wait while the image is loaded. Note that it is not performed in `init()` because it would be a time-consuming task. It would take up time in an AWT thread that one does not own. In this case, waiting in `init()` would be especially bad because `paint()` would never get called and our ‘loading’ message would not be displayed; the applet would just hang until the image loaded. It is often better to create a new thread for initialization and display a startup message in the interim.

When a `MediaTracker` is constructed, a reference is given to a component (`this`). After creating a `MediaTracker`, images are assigned to it to manage. Each image is associated with an integer identifier that will be used later for checking on its status. Multiple images can be associated with the same identifier, allowing to manage them as a group. The value of the identifier is also used to prioritize loading when waiting on multiple sets of images; lower IDs have higher priority. In this case, only image is wanted to be managed, so we created one identifier called `MAIN_IMAGE` and passed it as the ID for an image in the call to `addImage()`.

In the applet’s `start` method, the `MediaTracker`’s `checkID()` routine is called with the **ID** of the image to see if it is already been loaded. If it has not, the applet fires up a new thread to fetch it. The thread executes the `run()` method, which simply calls the `MediaTracker` `waitForID()` routine and blocks on the image, waiting for it to finish loading. The `show` flag tells `paint()` whether to display the status message or the actual image. A `repaint()` is done immediately upon entering `run()` to display the “**Loading...**” status, and again upon exiting to change the display. Errors are tested for during image preparation with `isErrorID()` and the status message changed if there is one.

This may seem like a lot of work to go through, just to put up a status message while loading a single image. `MediaTracker` is more valuable when many images are being worked with that have to be available before parts of an application can be started. It saves one from implementing a custom `ImageObserver` for every application. Note if the image is retrieved from a local disk or very fast network, this might go by quickly, so pay attention.

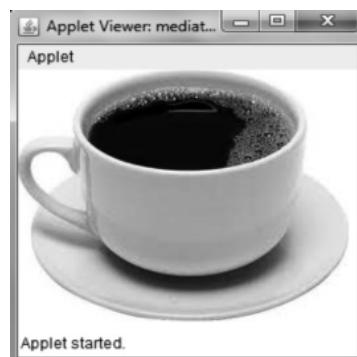


Figure 22.7 Output screen of Program 22.8

For producing an application’s own image data, `ImageProducer` interface needs to be implemented as defined in `java.net.image`. The interface is there for objects which can produce the image data for images. Each image contains an `ImageProducer` which is used to reconstruct the image

22.6 PRODUCING IMAGE DATA

whenever it is needed; for example, when a new size of the image is scaled, a width or height of the image is required.

If one is interested in producing a static image with just one frame, a utility class can be used that acts as an `ImageProducer`. Fortunately, Java provides a class called `MemoryImageSource` that implements `ImageProducer` interface. It is defined within the package `java.awt.image`. Pixel data is given to it in an array and it sends that data to an image consumer. A `MemoryImageSource` can be constructed for a given colour model, with various options to specify the type and positioning of its data. The simplest form will be used, which assumes an **ARGB** (Alpha, Red, Green and Blue) colour model.

The class defines 7 constructors but just one of its form will be used:

```
MemoryImageSource(int width, int height, int pixel[], int offset,
int scan)
```

The constructor constructs an `ImageProducer` object which uses an array of integers in the default RGB `ColorModel` to produce data for an `Image` object. The parameters are as follows:

w: the width of the rectangle of pixels; **H:** the height of the rectangle of pixels; **pix:** an array of pixels; **off:** the offset into the array where the first pixels are stored; **scan:** the distance from one row of pixels to the next in the array (i.e., the width of the scan line which is often the same as the width of the image).

In the default colour model, a pixel is an integer with Alpha, Red, Green and Blue(0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255.

```
/*PROG 22.9 DEMO OF MEMORYIMAGESOURCE VER 1*/
```

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
/*
<html>
<applet>
<applet code="MIS" width=200 height=200>
</applet>
</html>
*/
public class MIS extends Applet{
    Image img;
    int w, h;
    int[] pixData;
    public void init(){
        w = getSize().width;
        h = getSize().height;
        pixData = new int[w*h];
        int i = 0;
        for (int y = 0; y < h; y++)
        {
            int red = (y * 255) / (h - 1);
            for (int x = 0; x < w; x++)
            {
                int green = (x * 255) / (w - 1);
                int blue = 128;
                int alpha = 255;
                pixData[i] = alpha | (red << 16) | (green << 8) | blue;
                i++;
            }
        }
    }
}
```

```

        pixData[i++]=(alpha<<24) | (red<<16) |
                      (green<<8) | blue;
    }
}
img = createImage(new MemoryImageSource(w, h,
                                         pixData, 0, w));
}
public void paint(Graphics g){
    g.drawImage(img, 0, 0, this);
}
}
}

```

Explanation: The size of the image is determined by the size of the applet in the `init` method. The pixel data is created for an image in the `init()` method, and then `MemoryImageSource` is used to create and display the image in `paint()`. The variable `pixData` is a one-dimensional array of integers that holds 32-bit ARGB pixel values. In `init()`, we loop over every pixels in the image and assign it an ARGB value. The alpha (transparency) component is always 255, which means the image is opaque. The blue component is always 128, half its maximum intensity. The red component varies from 0 to 255 along the y axis; likewise, the green component varies from 0 to 255 along the x axis. The line below combines these components into an ARGB value:

```
pixData[i++]=(alpha<<24) | (red<<16) | (green<<8) | blue;
```

This alpha value takes the top bytes of the integer, followed by the red, green and blue values.

When the `MemoryImageSource` is constructed as a producer for this data, five parameters are given to it: the width and height of the image to construct (in pixels), the `pixData` array, an offset into that array and the width of each scan line (in pixels). The array `pixData` has **width*height** elements, which means it has one element for each pixel.

The actual image is created once, in using the `createImage()` method that an applet inherits from `Component`. The `createImage()` is used to generate an image from a specified `ImageProducer`. `createImage()` creates the image object and receives pixel data from the producer to construct the image. Finally the image is displayed in the `paint` method.



Figure 22.8 Output screen of Program 22.9

```
/*PROG 22.10 DEMO OF MEMORYIMAGESOURCE VER 2*/
```

```

import java.awt.*;
import java.awt.image.*;
import java.applet.*;
/*
<html>
<applet>
<applet code="MIS2" width=200 height=200>

```

```
</applet>
</html>
*/
public class MIS2 extends Applet
{
    Image img;
    int w, h;
    int[] pixData;
    public void init()
    {
        w = getSize().width - 10;
        h = getSize().height - 10;
        pixData = new int[w*h];
        int i = 0;
        for (int y = 0; y < h; y++)
        {
            int red = (y * 255) / (h - 1);
            for (int x = 0; x < w; x++)
            {
                int blue = (x * 255) / (w - 1);
                pixData[i++] = (255<<24) | (red<<16)
                                | blue;
            }
        }
        img = createImage(new MemoryImageSource(w, h,
                                                pixData, 0, w));
    }
    public void paint(Graphics g)
    {
        g.drawImage(img, 0, 0, this);
    }
}
```

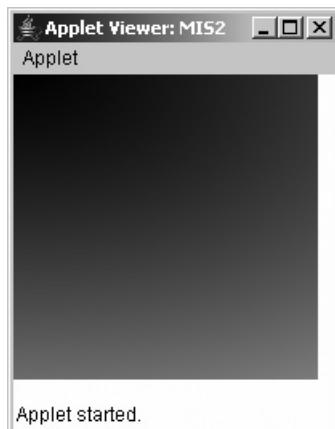


Figure 22.9 Output screen of Program 22.10

Explanation: This program is similar to the previous one with the difference that this time only blue and red components are dealt with.

```
/*PROG 22.11 DEMO OF MEMORYIMAGESOURCE VER 3*/  
  
import java.awt.*;  
import java.awt.image.*;  
import java.applet.*;  
/*  
<html>  
<applet>  
<applet code="MIS3" width=200 height=200>  
</applet>  
</html>  
*/  
public class MIS3 extends Applet  
{  
    Image img;  
    int w, h;  
    int[] pixData;  
    public void init()  
    {  
        w = getSize().width - 10;  
        h = getSize().height - 10;  
        pixData = new int[w*h];  
        int i = 0;  
        for (int y = 0; y < h; y++)  
        {  
            for (int x = 0; x < w; x++)  
            {  
                int red = (x * 2 ^ y * 2) & 0xff;  
                int green = (x * 4 ^ y * 4) & 0xff;  
                int blue = (x * 8 ^ y * 8) & 0xff;  
                pixData[i++] = (255<<24) | (red<<16) |  
                               (green<<8) | blue;  
            }  
        }  
        img = createImage(new MemoryImageSource(w, h,  
                                              pixData, 0, w));  
    }  
    public void paint(Graphics g)  
    {  
        g.drawImage(img, 0, 0, this);  
    }  
}
```

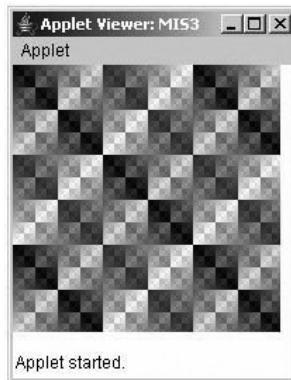


Figure 22.10 Output screen of Program 22.11

Explanation: In this program, inside the inner **for** loop, there are binary **XOR** and **AND** operations for computing the values of red, green and blue components. Try changing/swapping operations and see the colourful squares as shown in the output.

```
/* PROG 22.12 DEMO OF MEMORYIMAGESOURCE VER 4 DOING ANIMATION */

import java.awt.*;
import java.awt.image.*;
import java.applet.*;
/*
<html>
<applet code="MIS4" width=200 height=200>
</applet>
</html>
*/
public class MIS4 extends Applet implements Runnable
{
    int pixData[];
    MemoryImageSource misrc;
    Image image;
    int w, h;
    public void init()
    {
        w = getSize().width;
        h = getSize().height;
        pixData = new int[w*h];
        misrc = new MemoryImageSource(w,h,pixData,0,w);
        misrc.setAnimated(true);
        image = createImage(misrc);
        new Thread(this).start();
    }
}
```

```

public void run()
{
    while (true)
    {
        try
        {
            Thread.sleep(1000 / 20);
        }
        catch (InterruptedException e)
        {
            System.out.println("Thread interrupted");
        }
        for (int x = 0; x < w; x++)
            for (int y = 0; y < h; y++)
            {
                boolean rand = Math.random() > 0.5;
                int col = rand ? Color.black.getRGB():
                    Color.white.getRGB();
                pixData[y * w + x] = col;
            }
        misrc.newPixels(0, 0, w, h);
    }
}
public void paint(Graphics g)
{
    g.drawImage(image, 0, 0, this);
}
}

```

Explanation: `MemoryImageSource` can also be used to generate sequence of images or to update an image dynamically. The program simulates the static on a television screen. It generates successive frames of random black and white pixels and displays each frame when it is complete.

The `init()` method sets up the `MemoryImage` Source that produces the sequence of images. It then creates a `MemoryImageSource` object that produces images the width and height of the display. Pixels are started to be taken from the beginning of the pixel array, and scan lines in the array have the same width as the image. Once the `MemoryImageSource` is created, its `setAnimated()` method is called to tell it that an image sequence will be generated. Then the source is used to create an image that will display the sequence.

Next, a thread is started that generates the pixel data. For every element in the array, a random number is obtained, and the pixel is set to black if the random number is greater than 0.5. Because `pixData` is an `int` array, `Color` objects cannot be assigned to it directly; `getRGB()` is used to extract the color components from the black and white color constants. When the entire array is filled with data, the `newPixels()` method is called, which delivers the new data to the image.

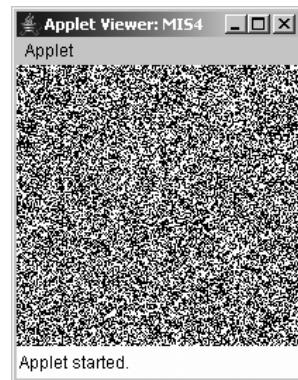


Figure 22.11 Output screen of Program 22.12

Whenever `paint()` is called, the latest collection of static is seen. The image observer, which is the Applet itself, schedules a call to `paint()`, whenever anything interesting has happened to the image.

22.7 CONSUMING IMAGE DATA

Consuming image data is reverse of producing image data that as seen in previous section. For consuming image data, Java provides `ImageConsumer` interface. The interface is for objects expressing interest in image data through the `ImageProducer` interfaces. When a consumer is added to an image producer, the producer delivers all of the data about the image using the method calls defined in this interface. An object that implements the `ImageConsumer` interface is going to create int or byte arrays that represent pixels from an image object.

Java's built-in class `PixelGrabber`, defined in the package `java.awt.image`, implements `ImageConsumer` interface and thus acts as a consumer of image data. The class is opposite of `MemoryImageSource` class; as a consumer it grabs the pixels from the image. To use `PixelGrabber` an array of ints big enough to hold the pixels data is first created, and then a `PixelGrabber` instance is created by passing in the rectangle of interest. Finally, `grabPixels()` is called on that instance.

The class defines three constructors but just one form will be used that is shown below:

```
PixelGrabber(Image imgobj, int left, int top, int width, int height,
int pixel[], int offset, int scan)
```

The constructor creates a `PixelGrabber` object to grab the (x, y, w, h) rectangular section of pixels from the image produced by the specified `ImageProducer` into the given array. The pixels are stored into the array in the default RGB ColorModel. The RGB data for pixel is (i, j) where (i, j) is inside the rectangle (x, y, w, h) stored in the array at $\text{pix}[(j-y)*\text{scansize} + (i-x) + \text{off}]$.

For example:

```
image = getImage(getDocumentBase(), "image.jpg");
int pixels[] = new int[1000*500];
PixelGrabber pg = new PixelGrabber(image, 0, 0, 1000, 500, pixels,
0, 1000);
```

For storing the pixels into array `pixels`, nested `for` loop or a single `for` loop can be used.

The `grabPixels` has the following signature:

```
boolean grabPixels() throws InterruptedException
```

The method requests the `Image` or `ImageProducer` to start delivering pixels and wait for all of the pixels in the rectangle of interest to be delivered.

Check out few programming examples for better understanding of `PixelGrabber` class:

```
/*PROG 22.13 DEMO OF PIXELGRABBER CLASS VER 1*/
```

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;
public class pixgrabl extends Applet
{
    Image image, image2;
    int p;
    public void init()
    {
```

```

        image = getImage(getDocumentBase(), "gay1.jpg");
        int pixData[] = new int[1000*500];
        PixelGrabber pg = new PixelGrabber(image, 0, 0,
                                            1000, 500, pixData, 0, 1000);
        try
        {
            pg.grabPixels();
        }
        catch(InterruptedException e){}
        for(int i=0; i<1000*500;i++)
        {
            p = pixData[i];
            int r = 0xff - (p>>16) & 0xff;
            int g = 0xff - (p>>8) & 0xff;
            int b = 0xff - p & 0xff;
            pixData[i] = (0xff000000|r<<16|g<<8|b);
        }
        image2 = createImage(new MemoryImageSource(1000,
                                                    500, pixData, 0, 1000));
    }
    public void paint(Graphics g)
    {
        g.drawImage(image2, 10, 10, this);
    }
}

```



Figure 22.12 Output screen of Program 22.13

Explanation: The image object represents the image ‘gay1.jpg’. An int array pixData is created of size 1000*500 with width 1000 and height 500. The PixelGrabber constructor creates a PixelGrabber object pg to grab the (x, y, w, h) rectangular section of pixels from the image produced by the image into the array as discussed in the theory part. Next, the grabPixels grabs the pixels into the array pixData.

The main work is done into the for loop. The code works as invert filter. It takes apart the red, green and blue channels and then inverts them by subtracting them from 255. These inverted values are packed back into a pixels value and returned.

At the end of for loop the image is created using the MemoryImageSource passing array pixData as one of its argument. Later the image is displayed in the paint method. Note the output shows inverted image of the original image.

```

/*PROG 22.14 DEMO OF PIXELGRABBER CLASS VER 2 */

import java.awt.*;
import java.applet.*;
import java.awt.image.*;
public class copyImg1 extends Applet
{
    Image image, image2;
    public void init() {
        image = getImage(getDocumentBase(), "gay1.jpg");
        int pixData[] = new int[400 * 400];
        PixelGrabber pg = new PixelGrabber(image, 0, 0,
                                           400, 400, pixData, 0, 400);
        try
        {
            pg.grabPixels();
        }
        catch(InterruptedException e){}
        for(int i =0;i<400*400;i++)
        {
            int p =pixData[i];
            int red = 0xff & (p>>16);
            int green = 0xff & (p>>8);
            int blue = 0xff & p;
            pixData[i]=(0xff000000|red<<16|green<<8|blue);
        }
        image2 = createImage(new MemoryImageSource(400,
                                                   400, pixData, 0, 400));
    }
    public void paint(Graphics g)
    {
        g.drawImage(image2, 10, 10, this);
    }
}

```

Explanation: This program is similar to the previous one with the change on the `for` loop. The coding in the `for` loop performs the copying of one image to another image, pixel by pixel but only of size 400. For this reason, after constructing the `PixelGrabber` object and grabbing the pixels, in the `for` loop each pixel value is taken from `pixData` array and stored in `p`. From this the red, green and blue components are extracted. In the next line, a new pixel value is formed as seen in earlier examples and assigned to `pixData` element. After the end of `for` loop, using `pixData` and `MemoryImageSource`, `image2` object is created and the same is displayed in the `paint` method.



Figure 22.13 Output screen of Program 22.14

```
/*PROG 22.15 DEMO OF PIXELGRABBER CLASS VER 3 */  
  
import java.awt.*;  
import java.applet.*;  
import java.awt.image.*;  
public class grayscale extends Applet  
{  
    Image image, image2;  
    public void init()  
    {  
        image = getImage(getDocumentBase(), "pic.jpg");  
        int pixData[] = new int[400*400];  
        PixelGrabber pg = new PixelGrabber(image, 0, 0,  
                                         400, 400, pixData, 0, 400);  
        try  
        {  
            pg.grabPixels();  
        }  
        catch (InterruptedException e) { }  
        for (int i = 0; i < 400 * 400; i++)  
        {  
            int p = pixData[i];  
            int red = (0xff & (p >> 16));  
            int green = (0xff & (p >> 8));  
            int blue = (0xff & p);  
            int avg = (int)((red + green + blue) / 3);  
            pixData[i] = (0xff000000 | avg << 16 | avg  
                         << 8 | avg);  
        }  
        image2 = createImage(new MemoryImageSource(400,  
                                                 400, pixData, 0, 400));  
    }  
    public void paint(Graphics g)  
    {  
        g.drawImage(image2, 10, 10, this);  
    }  
}
```



Figure 22.14 Output screen of Program 22.15

Explanation: In the `for` loop, the pixels are computed in the grayscale of the same brightness as of the original image. For computing the pixel of grayscale, the red, green and blue components are simply added and the average found. This is stored in `avg`; this `avg` is left stored as an element in the `pixData` array. Note there is just one change in the program from the previous one and the gray scaled picture of the original one is obtained.

22.8 PONDERABLE POINTS

1. An image is simply a rectangular graphical object. It is simply a 2D array of pixels. To a computer, an image is just a set of numbers. The numbers specify the colour of each pixels in the image.
2. The standard class `java.awt.Image` is used to represent images. Images are objects of the `Image` class. The abstract class `Image` is the super class of all classes that represent graphical images.
3. The `Applet` class defines a method `getImage` that can be used for loading images stored in GIF and JPEG files and possibly in other types of imaged files, depending on the version of Java.
4. `ImageObserver` is an update interface for receiving notifications about image information as the image is constructed.
5. The technique of drawing to an off-screen canvas and then quickly copying the canvas to the screen is called double buffering. The use of an off-screen image to reduce flicker is called double buffering, because the screen is considered a buffer for pixels, and the off-screen image is the second buffer, where pixels can be prepared for display.
6. A `MediaTracker` monitors the preparation of an image or a group of images and lets one check on them periodically, or wait until they are completed. `MediaTracker` uses the `ImageObserver` interface internally to receive image update.
7. For producing the application's own image data, `ImageProducer` interface needs to be implemented as defined in `java.awt.image`.
8. The `MemoryImageSource` class implements `ImageProducer` interface.
9. A `MemoryImageSource` can be constructed for a given colour model, with various options to specify the type and positioning of its data. It receives pixel data in an array and sends that data to an image consumer.
10. Consuming image data is reverse of producing image data. For consuming image data, Java provides `ImageConsumer` interface.
11. An object that implements the `ImageConsumer` interface is going to create int or byte arrays that represent pixels from an `Image` object. The `PixelGrabber` class implements this interface.

REVIEW QUESTIONS

1. What are the two forms of loading and image? Explain with example.
2. What is an image observer? Why do we use it?
3. What information can be obtained about an image by overriding `imageUpdate` method?
4. What is double buffering and why it is used?
5. What is the purpose of `ImageConsumer` and `ImageProducer` interface?
6. Explain the purpose of classes `MemoryImageSource` and `PixelGrabber`? Where are they useful?
7. Why do we use `MediaTracker` class? How does it receive image updates?
8. What is the mechanism of loading an image in Java?

Multiple Choice Questions

1. The Applet class defines a method _____ that can be used for loading images stored in GIF and JPEG files.
 - (a) getImage
 - (c) selectImage
 - (b) readImage
 - (d) None of the above
2. Which interface is used for receiving notification about image information as the image is constructed?
 - (a) getImage
 - (c) ImageObserver
 - (b) setImage
 - (d) ImageNotification
3. A _____ monitors the preparation of an image or a group of images.
 - (a) GroupImage
 - (b) ImageObserver
 - (c) getImageInformation
 - (d) MediaTracker
4. Which interface is used for producing our own image data?
 - (a) ImageData
 - (c) MediaTracker
 - (b) ImageProducer
 - (d) None of the above
5. ImageProducer interface is implemented by
 - (a) Image class
 - (b) ImageConsume class
 - (c) MemoryImageSource
 - (d) None of the above
6. The _____ class is used to implement the ImageConsume interface.
 - (a) MemoryImageSource
 - (b) ImageProducer
 - (c) MediaTracker
 - (d) PixelGrabber
7. Which is the correct signature for getImage method
 - (a) public Image getImage(URL url)
 - (b) void Image getImage(URL url)
 - (c) public Image getImage(String name)
 - (d) void Image getImage(String name)
8. Which flag indicates that a static image which was previously drawn is now complete. The x, y, width and height arguments are ignored.
 - (a) SOMEBITS
 - (b) ALLBITS
 - (c) PROPERTIES
 - (d) None of the above
9. The technique of drawing to an off-screen canvas and then quickly copying the canvas to the screen is called
 - (a) Updating image
 - (b) Image copying
 - (c) Double buffering
 - (d) Updating buffer
10. Which of the following is the overloaded form of addImage method?
 - (a) public void addImage(Image image)
 - (b) public void addImage(Image image, int id)
 - (c) public void addImage(Image image, int id, int h)
 - (d) public void addImage(Image image, int id, int w, int h)

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. c 3. d 4. b 5. c 6. d 7. a 8. b 9. c 10. d

Introduction to Swing

23

23.1 WHAT IS SWING?

AWT, layout managers and event handling have been covered in earlier chapters. This chapter deals with refind form of AWT components and other aspects in a form of Swing package. Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT. A number of features of Swing component are given below:

1. *Lightweight*: Not built on native window-system windows.
2. *Much bigger set of built-in controls*: Trees, image, buttons, tabbed panes, sliders, toolbars, color choosers, tables, text areas to display HTML or RTF, etc.
3. *Much more customizable*: Can change look and feel at runtime, or design own look and feel.
4. *Many miscellaneous new features*: Double-buffering built-in, tool tips, dockable tool bars, keyboard accelerators, custom cursors, etc.

Swing is a standard in Java 2. Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform independent. The term ‘lightweight’ is used to describe such elements.

The swing-related classes are contained in `javax.swing` and its subpackages, such as

`javax.swing.table`

In this chapter, some of the components defined by the `javax.swing` and other subpackages will be covered.

The AWT component classes and their equivalent swing components are given in the Table 23.1.

| AWT Component Class | Swing Counterpart Class |
|---------------------|-------------------------|
| Applet | JApplet |
| Button | JButton |
| Label | JLabel |
| Checkbox | JCheckbox |
| CheckboxGroup | JRadioButton |
| TextField | JTextField |
| ScrollPane | JScrollPane |
| List | JList |
| TextArea | JTextArea |
| Panel | JPanel |
| Component | JComponent |

Table 23.1 AWT components and their counterpart Swing components

23.2 THE JAPPLET CLASS

This class is an extended version of `java.applet`. Applet that adds support for the JFC/Swing component architecture. The `Applet` class is the super class of `JApplet` class. For adding components to the `JApplet` window, add method cannot be made use of directly. The `getContentPane` method defined by `JApplet` class has to be made use of. Its signature is shown here:

```
Container getContentPane()
```

The method returns the `contentPane` object for this applet. After obtaining the `contentPane`, components can be added to using `add` method.

23.3 THE IMAGEICON CLASS

This class is an implementation of the icon interface that paints icon from images that are created from a URL, filename or byte array. The class has two commonly used constructors:

```
ImageIcon(String filename)
ImageIcon(URL url)
```

The `filename` denotes name of icon file and `url` specifies complete URL for the file.

23.4 THE JLABEL CLASS

This class is similar to `label` class defined by AWT but icon images can be added along with the text. Some if the constructors are listed below:

- `JLabel()`**

This form of constructor creates a `JLabel` instance with no image and with an empty string for the title.

- `JLabel(Icon image)`**

This form of constructor creates a `JLabel` instance with the specified image.

- `JLabel(Icon image, int h_align)`**

This form of constructor creates a `JLabel` instance with the specified image and horizontal alignment. The `h_align` may be in form `SwingConstants.X`, where `X` may be `LEFT`, `RIGHT`, `CENTER`, `LEADING` or `TRAILING`. The `SwingConstants` is an interface defined in the `javax.swing` package.

- `JLabel(String text, Icon icon, int h_align)`**

This form of constructor creates a `JLabel` instance with the specified text, images and horizontal alignment.

Some of the methods of `JLabel` class are given in the Table 23.2.

| Method Signature | Description |
|--|---|
| <code>Icon getIcon()</code> | Returns the graphics images that the label displays |
| <code>void setIcon(Icon icon)</code> | Defines the icon this component will display |
| <code>String getText()</code> | Returns the text string that the label displays |
| <code>void setText(String text)</code> | Defines the single line of text this component will display |

Table 23.2 Some methods of `JLabel` class

```
/*PROG 23.1 DEMO OF JLABEL1 */

import java.awt.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="JLabel1" width = 200 height = 200>
</applet>
</html>
*/
public class JLabel1 extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        JLabel JL = new JLabel("WELCOME", new
                               ImageIcon("gan.jpg"), JLabel.CENTER);
        contentPane.add(JL);
    }
}
```

Explanation: The program is simple to understand. An iconed label has been created in this program. The icon is created using `ImageIcon` class which takes a single argument (i.e., name of the icon file). The file may be `.bmp`, `.gif` or `.jpeg`. The label is added to the content pane of the applet. The content pane is obtained using the method `getContentPane`.



Figure 23.1 Output screen of Program 23.1

23.5 THE JButton CLASS

The `JButton` class is similar to `Button` class but with a number of new features. The `AbstractButton` class is the super class for `JButton` class. In the button created by `JButton` class, there can be image icons associated with buttons. Some of its constructors are shown below:

```
JButton()
JButton(Icon icon)
JButton(String text, Icon icon)
```

In the second and third form of constructor, button is created with specified icon. The third method defined by `AbstractButton` class can be used for setting the icons for various states for an instance of `JButton`.

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void RolloverIcon(Icon ri)
```

Here, `di`, `pi`, `si` and `ri` are the icons to be used for these different conditions.

The `setToolTipText` can be used for showing a tool tip text for the button. It has the following signature:

```
void setToolTipText(String text)
```

The method registers the text to display in a tool tip. The text displays when the cursor lingers over the component.

```
void setMnemonic(int mnemonic)
```

The method sets the keyboard mnemonic (i.e., shortcut key). A mnemonic must correspond to a single key on the keyboard and should be specified using one of the `VK_XXX` key codes defined in `java.awt.event.KeyEvent`. Also, they are case-insensitive.

The action command can be changed by calling the button's `setActionCommand()` method.

```
void setActionCommand(String actionCommand)
```

The method sets the action command for this button that can be used inside the `actionPerformed` method for recognizing the button. The method is defined by `AbstractButton` class. The method `getActionCommand` returns the action command set using `setActionCommand`.

```
/*PROG 23.2 DEMO OF JButton */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="Jbutton1" width = 200 height = 200>
</applet>
</html>
*/
public class Jbutton1 extends JApplet implements
ActionListener
{
    JButton jb;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        ImageIcon img = new ImageIcon("img1.jpg");
        jb = new JButton(img);
        jb.addActionListener(this);
        jb.setToolTipText("Keep mouse pressed for change
                           image ");
        contentPane.add(jb);
    }
    public void actionPerformed(ActionEvent ae)
    {
        ImageIcon img = new ImageIcon("img2.jpg");
        jb.setPressedIcon(img);
    }
}
```



(a) After running applet

(b) When mouse is pressed on applet

Figure 23.2 Output screen of Program 23.2

Explanation: In this program, an instance of JButton is created with no text but image. The tool tip for the button is set using `setToolTipText` method. When button is pressed, `actionPerformed` method is called. If the button is kept pressed for few seconds, changed image can be seen as it was done in `actionPerformed` method using `setPressedIcon`.

23.6 THE JTEXTFIELD CLASS

JTextField is a lightweight component that allows the ending of a single line of text. The class has JTextField as its base class which in turn inherits JComponent class. This component's capability is not found in the java.awt.TextField class. Some of its constructors are shown below:

```
JTextField()
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

Here, **s** is the string to be presented, and **cols** is the number of columns in the text field.

The method `setEchoChar` and `getEchoChar` are not supported by JTextField class; instead a new class JPasswordField extends JTextField to provide this service.

Once a JPasswordField instance has been created, `setEchoChar` method can be used for obscuring the input.

Some of the methods of JTextField class are given in the Table 23.3.

| Method Signature | Description |
|--|---|
| <code>void setEditable(boolean b)</code> | Sets the specified Boolean to indicate whether or not this text field should be editable. |
| <code>boolean isEditable()</code> | Returns the Boolean indicating whether this text field is editable or not. |
| <code>String getText()</code> | Returns the text contained in this text field. |
| <code>Void setText(String t)</code> | Sets the text of this text field to the specified text. |

Table 23.3 Some methods of JTextField class

```
/*PROG 23.3 DEMO OF JTEXTFIELD AND JPASSWORDFIELD */

import java.awt.*;
import javax.swing.*;
public class Jtext1 extends JApplet
{
    JTextField name;
    JPasswordField pass;
    JLabel Lname, Lpass;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        Lname = new JLabel("Enter your name");
        Lpass = new JLabel("Enter your password");
        name = new JTextField(15);
        pass = new JPasswordField(15);
        pass.setEchoChar('+');
        contentPane.add(Lname);
        contentPane.add(name);
        contentPane.add(Lpass);
        contentPane.add(pass);
    }
}
```

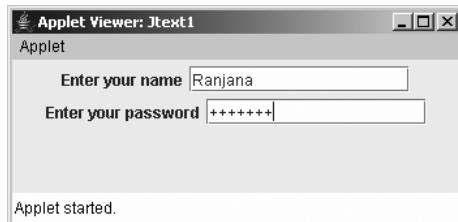


Figure 23.3 Output screen of Program 23.3

Explanation: The program is self-explanatory.

23.7 THE JCHECKBOX CLASS

JCheckBox class is an implementation of a check box, an item that can be selected or deselected, and which displays its state to the user. By convention, any number of check boxes in a group can be selected. The class has JToggelButton as its super class that has AbstractButton as its super class. Some of the constructors of this class are shown below:

```
JCheckBox(Icon ic);
JCheckBox(Icon ic, boolean state)
```

```
JCheckBox(String str)
JCheckBox(String str, boolean state);
JCheckBox(String str, Icon ic)
JCheckBox(String str, Icon ic, boolean state)
```

Here, ic is the icon for the button. The text is specified by str. If state is true, the check box is initially selected else, it is not.

```
/*PROG 23.4 DEMO OF JCHECKBOX */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="JCheckt1" width = 200 height = 200>
</applet>
</html>
*/
public class JCheck1 extends JApplet implements
ActionListener
{
    JLabel caption, hidden;
    JCheckBox cb1, cb2, cb3, cb4;
    JButton jb;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        caption = new JLabel("Which actress you like?");
        cb1 = new JCheckBox("Gauri", new
                           ImageIcon("im1.jpg"));
        cb2 = new JCheckBox("Gaytri", new
                           ImageIcon("im2.jpg"));
        cb3 = new JCheckBox("Dipika", new
                           ImageIcon("im3.jpg"));
        cb4 = new JCheckBox("Katrena", new
                           ImageIcon("im4.jpg"));
        jb = new JButton("Submit");
        hidden = new JLabel(" ");
        hidden.setVisible(false);
        hidden.setBackground(Color.cyan);
        contentPane.add(caption);
        contentPane.add(cb1);
        contentPane.add(cb2);
        contentPane.add(cb3);
        contentPane.add(cb4);
        contentPane.add(jb);
        jb.addActionListener(this);
        jb.setActionCommand("submit");
    }
}
```

```

        contentPane.add(hidden);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String msg = "You Like: ";
        if (cb1.isSelected())
            msg += cb1.getText() + ",";
        if (cb2.isSelected())
            msg += cb2.getText() + ",";
        if (cb3.isSelected())
            msg+=cb3.getText() + ",";
        if (cb4.isSelected())
            msg += cb3.getText() + ",";
        hidden.setText(msg);
        hidden.setVisible(true);
    }
}

```

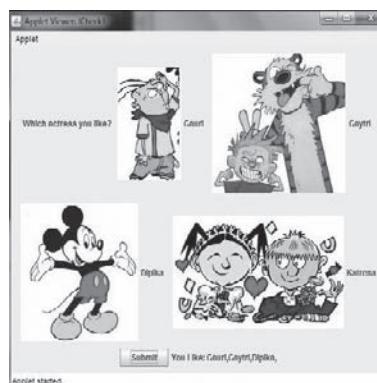


Figure 23.4 Output screen of Program 23.4

Explanation: In this program, four instance of JCheckBox class are created with images on them. One button is also there for showing the selection done by the user. A label is created in the init method. Initially it is hidden. When the user presses the submit button, in the actionPerformed method, state of the check boxes is checked and display string is formed. The display string set as text for the label is made visible.

23.8 THE JRADIOBUTTON CLASS

JRadioButton class is an implementation of a radio button, an item that can be selected or deselected, and which displays its state to the user. It is used with a ButtonGroup object to create a group of buttons in which only one button at a time can be selected. For this reason, a ButtonGroup object is created and its add method used to include the JRadioButton objects in the group. Some of the constructors of this class are shown below:

```

JRadioButton(Icon ic)
JRadioButton(Icon ic, boolean state)
JRadioButton(String str)

```

```
JRadioButton(String str, boolean state)
JRadioButton(String str, Icon ic)
JRadioButton(String str, Icon ic, boolean state)
```

Here, **ic** is the icon for the button. The text is specified by **str**. If state is true, the button is initially selected else, it is not.

```
/*PROG 23.5 DEMO OF RADIOBUTTON */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="JRadio1" width = 200 height = 200>
</applet>
</html>
*/
public class JRadio1 extends JApplet implements
ActionListener
{
    JLabel caption, hidden;
    JRadioButton rb1, rb2, rb3, rb4;
    JButton jb;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        caption = new JLabel("What is your age group?");

        rb1 = new JRadioButton("1-15 Yrs", new
                               ImageIcon("smile1.gif"));
        rb2 = new JRadioButton("16-25 Yrs", new
                               ImageIcon("smile2.gif"));
        rb3 = new JRadioButton("26-40 Yrs", new
                               ImageIcon("smile3.gif"));
        rb4 = new JRadioButton("40>Yrs", new
                               ImageIcon("smile4.gif"));

        ButtonGroup bg = new ButtonGroup();
        bg.add(rb1);
        bg.add(rb2);
        bg.add(rb3);
        bg.add(rb4);

        jb = new JButton("Submit");

        hidden = new JLabel(" ");
        hidden.setVisible(false);
        hidden.setBackground(Color.cyan);
        contentPane.add(caption);
```

```

        contentPane.add(rb1);
        contentPane.add(rb2);
        contentPane.add(rb3);
        contentPane.add(rb4);
        contentPane.add(jb);
        jb.addActionListener(this);
        jb.setActionCommand("submit");
        contentPane.add(hidden);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String msg = "You are in the age group";
        if (rb1.isSelected())
            msg += rb1.getText();
        if (rb2.isSelected())
            msg += rb2.getText();
        if (rb3.isSelected())
            msg += rb3.getText();
        if (rb4.isSelected())
            msg += rb4.getText();

        hidden.setText(msg);
        hidden.setVisible(true);
    }
}

```

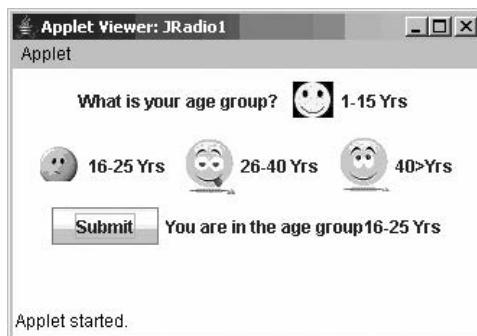


Figure 23.5 Output screen of Program 23.5

Explanation: In this program, radio buttons have been used with images on them. Note, to let the user to select one option at a time, the buttons are made part of the `ButtonGroup` instance. For this reason, an instance group of `ButtonGroup` class is created. The checkboxes are added to this group using `add` method. This adds checkboxes to button group logically. Rest is simple to understand.

23.9 THE JCOMBOBOX CLASS

The `JComboBox` is a component that combines a button or editable field and drop-down list. The user can select a value from the drop-down list, which appears at user's request. By default, only one item is visible in the combo box, but items are selected from a drop-down list which can be displayed when clicked over

down arrow icon in it. To make a combo box editable, it must include an editable field into which the user can type a value.

Some of the constructors of this class are as follows:

1. **JComboBox()**

This form of constructor creates a JComboBox with no elements in it.

2. **JComboBox(Object[] items)**

This form of constructor creates a JComboBox that contains the elements in the specified array.

3. **JComboBox(Vector items)**

This form of constructor creates a JComboBox that contains the elements in the specified vector.

Once a JComboBox instance has been created, items can be added to it using addItem method. Its signature is shown below:

```
void addItem(Object obj)
```

Here, obj is the object to be added to the combo box.

```
/*PROG 23.6 DEMO OF JCOMBOBOX VER 1*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="JCombo1" width = 200 height = 200>
</applet>
</html>
*/
public class JCombo1 extends JApplet implements ItemListener
{
    JLabel JL;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        Object[] arr ={ "C", "C++", "Java", "VC++" } ;
        JComboBox jc = new JComboBox(arr);
        JL = new JLabel(" ");
        jc.addItemListener(this);
        contentPane.add(jc);
        contentPane.add(JL);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        String s = (String)ie.getItem();
        JL.setText("Selected item:" + s);
    }
}
```

Explanation: In this program, an Object array containing four String object is created. This object array is added to the combo box by passing it in the constructor of JComboBox. Now all elements of the array become members of all the combo box. When any of the items is selected in combo box, the itemStateChanged method is called. In the method, selected item is displayed. The getItem returns an Object, so typecasting is a must.

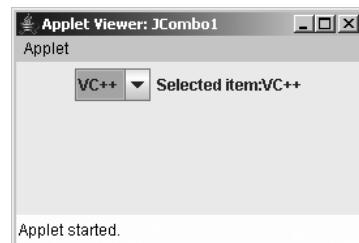


Figure 23.6 Output screen of Program 23.6

```
/*PROG 23.7 DEMO OF JCOMBOBOX VER 2 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="JCombo2" width = 200 height = 200>
</applet>
</html>
*/
public class JCombo2 extends JApplet implements ItemListener
{
    JLabel JL;
    ImageIcon happy, sad, tension, confuse;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JComboBox jc = new JComboBox();
        jc.addItem("happy");
        jc.addItem("sad");
        jc.addItem("confuse");
        jc.addItem("tension");
        jc.addItemListener(this);
        contentPane.add(jc);
        JL = new JLabel(new ImageIcon("im4.jpg"));
        contentPane.add(JL);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        String s = (String)ie.getItem();
        JL.setIcon(new ImageIcon(s+".jpg"));
    }
}
```

Explanation: In this program, when any item from the combo box is selected, the icon of the label changes to that type of image. For this purpose, item names and image file names are kept the same. When any item is selected, `itemStateChanged` method is called in the method; the selected item in string form is stored in `s`. The label icon is changed to `setIcon` method using selected item as file name of the icon to be set.

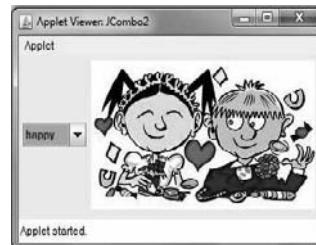


Figure 23.7 Output screen of Program 23.7

23.10 THE JTABBEDPANE CLASS

The `JTabbedPane` class is a component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon. Those who have ever dealt with the system control panel in windows, know what a `JTabbedPane` is. It is a container with the labelled tabs. When a tab is clicked on, a new set of controls is shown in the body of the `JTabbedPane`. In Swing, `JTabbedPane` is simply a specialized container.

Tabs/components are added to a `TabbedPane` object by using the `addTab` and `insertTab` methods. A tab is represented by an index corresponding to the position it was added in, where the first tab has an index equal to 0 and the last tab has an index equal to the tab count minus 1.

There are three constructors defined for the class but only two of them will be discussed as they are simple to use.

1. `JTabbedPane()`

This form of constructor creates an empty `TabbedPane` with a default tab placement of `JTabbedPane.TOP`.

2. `JTabbedPane(int tabPlacement)`

This form of constructor creates an empty `TabbedPane` with the specified tab placement of either `JTabbedPane.TOP`, `JTabbedPane.BOTTOM`, `JTabbedPane.LEFT` or `JTabbedPane.RIGHT`.

For creating a tabbed pane application, first create an instance of `JTabbedPane` class (e.g., `JTB`). Next, add as many tabs as required using `addTab` method of `JTabbedPane` class. Usually each tab is an instance of `Panel` class or one of its subclass defined by the user.

```
/*PROG 23.8 DEMO OF JTABBEDPANE */

import javax.swing.*;
/*
<html>
<applet>
<applet code ="JTab1" width = 200 height = 200>
</applet>
</html>
*/
class ColorsPanel extends JPanel
{
    public ColorsPanel()
    {
        JButton b1 = new JButton("Red");
        add(b1);
        JButton b2 = new JButton("Green");
        add(b2);
    }
}
```

```

        add(b2);
        JButton b3 = new JButton("Blue");
        add(b3);
    }
}
class MoviesPanel extends JPanel
{
    public MoviesPanel()
    {
        JRadioButton cb1 = new JRadioButton("Bheja Fry");
        add(cb1);
        JRadioButton cb2 = new JRadioButton("Yuvraj");
        add(cb2);
        JRadioButton cb3 = new JRadioButton("Masti");
        add(cb3);
        ButtonGroup bg = new ButtonGroup();
        bg.add(cb1);
        bg.add(cb2);
        bg.add(cb3);
    }
}
class FFFPanel extends JPanel
{
    public FFFPanel()
    {
        JComboBox JC = new JComboBox();
        JC.addItem("Pizza");
        JC.addItem("Chowmin");
        JC.addItem("Burgur");
        add(JC);
    }
}
public class JTab1 extends JApplet
{
    public void init()
    {
        JTabbedPane JTP = new JTabbedPane();
        JTP.addTab("Movies", new MoviesPanel());
        JTP.addTab("Colors", new ColorsPanel());
        JTP.addTab("Fast Food", new FFFPanel());
        getContentPane().add(JTP);
    }
}
}

```

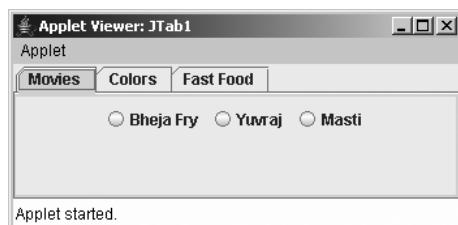


Figure 23.8 Output screen of Program 23.8

Explanation: In this program, three different classes extending JPanel class have been created. On each panel, different types of controls have been added. In the init method of JTab1 class, an instance JTP of JTabbedPane class is constructed. The addTab method of this class adds a new tab. The method takes two parameters: First is the tab name that is displayed onto the tab and second is the argument of Component type. Here, an object of each type MoviesPanel, ColorPanel and FFFPanel is put in three different addTab methods. Note no event handling is done for any of the controls present on any tab pane.

23.11 THE JSCROLLPANE CLASS

A JScrollPane is a container class that can hold one component. In other words, a JScrollPane wraps another component. It presents a rectangular area in which a component may be viewed. By default, if the wrapped component is larger than the JScrollPane itself, the JScrollPane supplies scrollbars. JScrollPane handles the events from the scrollbars and displays the appropriate portion of the contained component.

A JScrollPane has its own layout manager which cannot be changed. It can accommodate only one component at a time. This being the reason, if a lot of stuff are to be put in a JScrollPane, just the components should be put into a JPanel, with whatever layout manager one prefers, and the panel should be put into the JScrollPane.

Constructors of this class are shown below:

```
JScrollPane()
JScrollPane(Component comp)
JScrollPane(int vsb, int hsb)
JScrollPane(Component comp, int vsb, int hsb)
```

Here, the comp is the component to be displayed, and vsb and hsb denote vertical and horizontal policy, respectively. When a JScrollPane is created, the conditions under which its scrollbars will be displayed can be specified in Table 23.4. This is called the scrollbar display policy; a separate policy is used for the horizontal and vertical scrollbars. The following constants can be used to specify the policy for each of the scrollbars:

| Method Signature | Description |
|--------------------------------|---|
| HORIZONTAL_SCROLLBAR_AS_NEEDED | Displays a scrollbar only if the wrapped component does not fit. |
| HORIZONTAL_SCROLLBAR_ALWAYS | Always shows a scrollbar, regardless of the contained component's size. |
| HORIZONTAL_SCROLLBAR_NEVER | Never shows a scrollbar, even if the contained component will not fit. If this policy is used, some other way should be provided to manipulate the JScrollPane. |
| VERTICAL_SCROLLBAR_AS_NEEDED | Displays a scrollbar only if the wrapped component does not fit. |
| VERTICAL_SCROLLBAR_ALWAYS | Always shows a scrollbar, regardless of the contained component's size. |
| VERTICAL_SCROLLBAR_NEVER | Never shows a scrollbar, even if the contained component will not fit. If this policy is used, some other way should be provided to manipulate the JScrollPane. |

Table 23.4 Constants for JScrollPane class

By default, the properties are HORIZONTAL_SCROLLBAR_AS_NEEDED and VERTICAL_SCROLLBAR_AS_NEEDED.

For using JScrollPane in an application/applet, first there must be an instance of JComponent type, and then that component must be added to the scrollpane. In the end, add scrollpane to the content pane.

```

/*PROG 23.9 DEMO OF JSCROLLPANE */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="JSpane" width = 200 height = 200>
</applet>
</html>
*/
class MyPanel extends JPanel
{
    MyPanel()
    {
        setLayout(new GridLayout(20, 10));
        for (int i = 0; i < 19; i++)
        {
            for (int j = 0; j <= 9; j++)
                add(new JLabel(" "+((i*10+j)+1)));
        }
    }
}
public class JSpane extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        int vsb =
            ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int hsb =
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_
NEEDED;
        JScrollPane jsp = new JScrollPane(new MyPanel(),
                                         vsb, hsb);
        contentPane.add(jsp);
    }
}

```

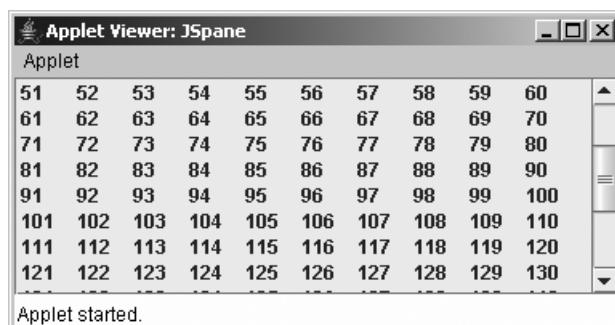


Figure 23.9 Output screen of Program 23.9

Explanation: In the `MyPanel` class constructor, the `GridLayout` of 20 rows and 10 columns is set. Using nested `for` loops, 200 instances of `JLabel` are created with numbers as their caption. In the `init` method of the `JSpace` class, the content pane's layout is set to `BorderLayout`. The `vsb` and `hsb` are initialized to constants discussed earlier. An instance of `MyPanel` class, `vsb` and `hsb`, are passed in the constructor of `JScrollPane` class. The `JScrollPane` instance is then added to the content pane of the applet window. The output is as shown. In the program, scroll bars can be used for viewing the contents of the panel.

23.12 THE JSPLITPANE CLASS

A split pane is a special container that holds two components, each in its own sub-pane. A splitter bar adjusts the sizes of the two panes. In a documents viewer, a split pane should be used to show a table of contents next to a full document. The class `JSplitPane` is used to divide two (and only two) components. The two components are graphically divided based on the look and feel implementation, and they can then be interactively resized by the user.

The two components in a split pane can be aligned left to right using `JSplitPane.HORIZONTAL_SPLIT`, or top to bottom using `JSplitPane.VERTICAL_SPLIT`. The preferred way to change the size of the Components is to invoke `setDividerLocation` where location is either the new `x` or `y` position, depending on the orientation of the `JSplitPane`.

When user resizes the split pane, the new space is distributed between the two components. The class has got four constructors but just one has been shown that is used in the program.

```
JSplitPane(int orient, Component left, Component right)
```

Here, `left` specifies the component to be added left, right to the right of pane and `orient` may take one of the two values stated above.

```
/*PROG 23.11 DEMO OF JSPLITPANE */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
/*
<html>
<applet>
<applet code ="SPane" width = 200 height = 200>
</applet>
</html>
*/
public class SPane extends JApplet
{
    public void init()
    {
        String fileOne = "gau4a.jpg";
        String fileTwo = "gay1v.jpg";
        JLabel JL1 = new JLabel(new ImageIcon(fileOne));
        Component left = new JScrollPane(JL1);

        JLabel JL2 = new JLabel(new ImageIcon(fileTwo));
        Component right = new JScrollPane(JL2);
        SplitPane split = new JSplitPane
```

```

        (JSplitPane.HORIZONTAL_SPLIT,
         left, right);
        split.setDividerLocation(200);
        getContentPane().add(split);
    }
}

```



Figure 23.10 Output screen of Program 23.11

Explanation: In this program, two different images are added onto the left and right side of the split pane. The images are contained within two instances of JScrollPane as done in the previous section. The returned references from the constructors of JScrollPane are stored in the left and right of Component type. In the constructor of the JSplitPane, the first argument is the split orientation, second is the left component and right is the right component. The divider location is set to 200, which is half of the width of the applet window.

23.13 DIALOGS

A dialog is another standard feature of user interface. Dialogs are frequently used to present information to the user to ask a question. Dialogs are so commonly used in GUI applications that Swing includes a handy set of pre-built dialogs. These are accessible from static methods in the JOptionPane class. **JOptionPane** makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something. JOptionPane groups them into three basic types Table 23.5:

| Method Signature | Description |
|--|--|
| Message dialog, method name is:
showMessageDialog | Displays a message to the user, usually accompanied by an OK button. |
| Confirmation dialog, method name is
showConfirmDialog | Asks a question and displays answer buttons, usually Yes, No and Cancel. |
| Option dialogs, method name is
showOptionDialog | The most general type—passes to the application's own components, which are displayed in the dialog. It is the unification of the rest of the three. |

Table 23.5 Types of dialog boxes

All dialogs are modal. Each `showXxxDialog` method blocks the current thread until the user's interaction is complete. A detailed discussion of each of them is as follows:

1. The `ShowMessageDialog` method

The method has three forms:

- `static void showMessageDialog(Component comp, Object msg)`
- `static void showMessageDialog(Component comp, Object msg, String title, int msgType)`
- `static void showMessageDialog(Component comp, Object msg, String title, int msgType, Icon ic)`

In all three forms, `comp` is the `Frame` in which the dialog is displayed; if null, or if the `comp` has no `Frame`, a default `Frame` is used, `msg` is the object to be displayed, `title` is the title string for the dialog, `msgType` is the type of message to be displayed: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` or `PLAIN_MESSAGE`. `ic` is `Icon` to be displayed within the message dialog box.

The following code produces a message dialog:

```
JOptionPane.showMessageDialog(f,"You have mail");
```

The dialog will be centred on the parent component. If null is passed for the parent component, the dialog is centred in the screen.

Here is a slightly fancier message dialog. A title is specified for the dialog and a message type, which affects the icon that is displayed:

```
JOptionPane.showMessageDialog(f,"You are low on memory",
"message","JOptionPane.WARNING_MESSAGE");
```

```
/*PROG 23.12 DEMO OF MESSAGE DIALOG */

import javax.swing.*;
class JPS12
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Title ver 1");
        int msgtype = JOptionPane.INFORMATION_MESSAGE;
        JOptionPane.showMessageDialog(null,"Message dialog
                                         ver 2","Title ver 2", msgtype);
        msgtype = JOptionPane.WARNING_MESSAGE;
        JOptionPane.showMessageDialog(null,"Message dialog
                                         ver 3","Title ver 3", msgtype, new
        ImageIcon("smile4.jpg"));
    }
}
```

Explanation: In the first version, the title is default and message is 'Title ver 1'. In the second version, the title, message and `msgtype` are used. The `msgtype` is set to `INFORMATION_MESSAGE` constant. In the third version, an image is used as last parameter.

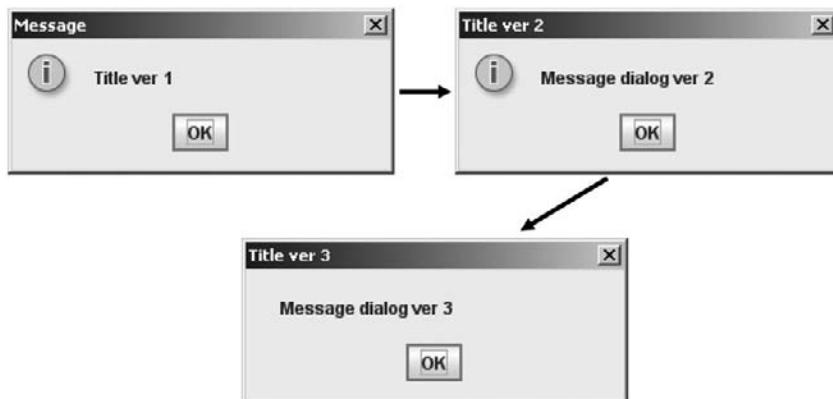


Figure 23.11 Output screen of Program 23.12

2. The `showConfirmDialog` method

The method has four forms:

- `static int showConfirmDialog(Component comp, Object msg)`
- `static void showConfirmDialog(Component comp, Object msg, String title, int optType)`
- `static void showConfirmDialog(Component comp, Object msg, String title, int optType, int msgType)`
- `static void showConfirmDialog(Component comp, Object msg, String title, int optType, int msgType, Icon ic)`

In all four forms, `comp` is the `Frame` in which the dialog is displayed; if null, or if the `comp` has no `Frame`, a default `Frame` is used, `msg` is the object to be displayed, `title` is the title string for the dialog and `msgType` is the type of message to be displayed: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` or `PLAIN_MESSAGE`. `ic` is icon to be displayed within the confirm message dialog box, the `optType` may be one of the followings: `YES_NO_CANCEL_OPTION`, `DEFAULT_OPTION`, `YES_NO_OPTION` and `OK_CANCEL_OPTION`.

The return type may be one of the following: `YES_OPTION`, `NO_OPTION`, `CLOSED_OPTION`, `CANCEL_OPTION` and `OK_OPTION`.

The various constants for dialog boxes are presented in the Table 23.6.

| Constant Name | Integer Value |
|----------------------------------|---------------|
| <code>CANCEL_OPTION</code> | 2 |
| <code>CLOSED_OPTION</code> | -1 |
| <code>DEFAILT_OPTION</code> | -1 |
| <code>ERROR_MESSAGE</code> | 0 |
| <code>INFORMATION_MESSAGE</code> | 1 |
| <code>NO_OPTION</code> | 1 |
| <code>OK_CANCEL_OPTION</code> | 2 |
| <code>OK_OPTION</code> | 0 |

(Continued)

| | |
|----------------------|----|
| PLAIN_MESSAGE | -1 |
| QUESTION_MESSAGE | 3 |
| WARNING_MESSAGE | 2 |
| YES_NO_CANCEL_OPTION | 1 |
| YES_NO_OPTION | 0 |
| YES_NO_OPTION | 0 |

Table 23.6 Constants for dialog boxes

An example of showConfirmDialog follows:

```
int result = JOptionPane.showConfirmDialog(null, "Do you want
to remove Windows now?");
```

In this case, null is passed for the parent component. Special values are returned from showConfirmDialog() to indicate which button has been pressed.

```
/*PROG 23.13 DEMO OF CONFIRM DIALOG VER 1 */

import javax.swing.*;
class JPS13
{
    public static void main(String[] args)
    {
        int msgtype = JOptionPane.INFORMATION_MESSAGE;
        int x = JOptionPane.showConfirmDialog(null,
                "Click any button", "Demo", msgtype);
        switch (x)
        {
            case JOptionPane.YES_OPTION:
                JOptionPane.showMessageDialog(null,
                        "Yes Selected");
                break;
            case JOptionPane.NO_OPTION:
                JOptionPane.showMessageDialog(null, "No
Selected");
                break;
            case JOptionPane.CANCEL_OPTION:
                JOptionPane.showMessageDialog(null,
                        "Cancel Selected");
                break;
            case JOptionPane.CLOSED_OPTION:
                JOptionPane.showMessageDialog(null,
                        "Closed Selected");
                break;
        }
    }
}
```

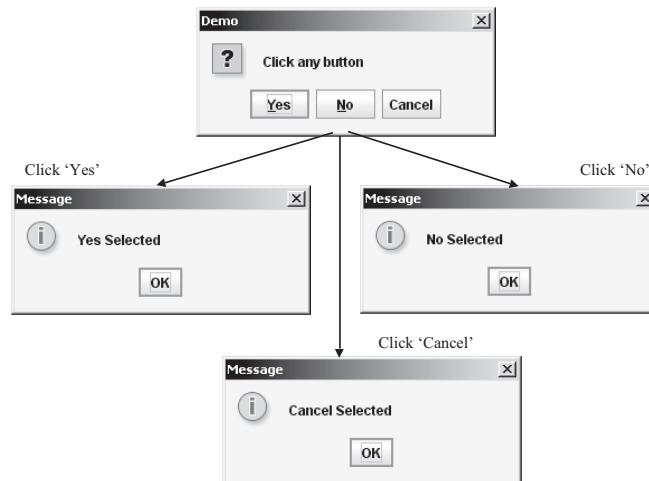


Figure 23.12 Output screen of Program 23.13

Explanation: The program displays a confirm message dialog with Yes-No-Cancel button. The returned value from the method is stored in x and using switch-case compared with the four possible options: YES_OPTION, NO_OPTION, CANCEL_OPTION and CLOSED_OPTION.

```
/*PROG 23.14 DEMO OF CONFIRM DIALOG VER 2 */

import javax.swing.*;
class JPS14
{
    public static void main(String[] args)
    {
        int x = JOptionPane.showConfirmDialog(null,
            "click any button", "Demo", 2, 2);
        switch (x)
        {
            case JOptionPane.OK_OPTION:
                JOptionPane.showMessageDialog(null, "Ok Selected");
                break;
            case JOptionPane.CANCEL_OPTION:
                JOptionPane.showMessageDialog(null,
                    "Cancel Selected");
                break;
            case JOptionPane.CLOSED_OPTION:
                JOptionPane.showMessageDialog(null,
                    "Bye Bye");
                break;
        }
    }
}
```

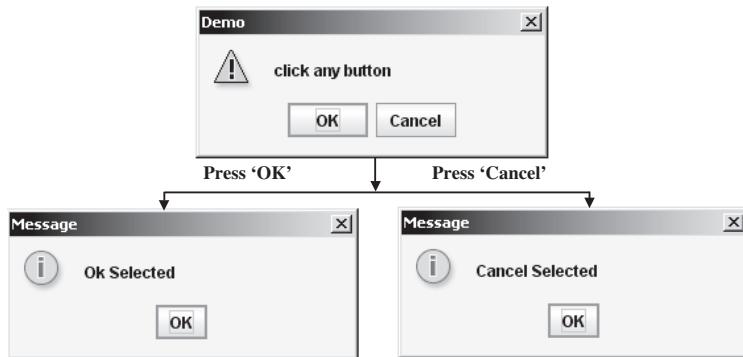


Figure 23.13 Output screen of Program 23.14

Explanation: Due to 4th argument as 2, the method showMessageDialog displays an OK-Cancel dialog box. Rest is simple to understand.

3. The showInputDialog method

The method has six forms:

- static String showInputDialog(Object msg)
- static String showInputDialog(Object msg, object ini_val)
- static String showInputDialog(Component com, Object msg)
- static String showInputDialog(Component com, Object msg, Object ini_val)
- static String showInputDialog(Component comp, Object msg, String ini_val, int msgType)
- static Object showInputDialog(Component com, Object msg, String title, int msgType, Icon ic, Object[] sel_vals, Object ini_sel_val)

Here, msg is usually the string requesting from user, ini_val is the initial value for selection, comp is the parent component, msgType as explained earlier, title is the title for the dialog box, sel_vals specifies values from which user is able to choose a value and ini_sel_val is the initial selection value.

The method is used for taking input from the user. All inputs come in the form of String. The following code puts up a dialog requesting the user's name.

```
String name = JOptionPane.showInputDialog(null, "Please enter
your name");
```

Whatever the user types is returned as a String, or null if the user presses the **Cancel** button. See few examples:

```
/*PROG 23.15 DEMO OF INPUT DIALOG VER 1 */

import javax.swing.*;
class JPS15
{
    static String input(String si)
    {
        String s = JOptionPane.showInputDialog(null, si);
        return s;
    }
}
```

```

    }
    public static void main(String args[])
    {
        String s = "Enter Your Name";
        JOptionPane.showMessageDialog(null, "Hello "
            +input(s));
    }
}

```

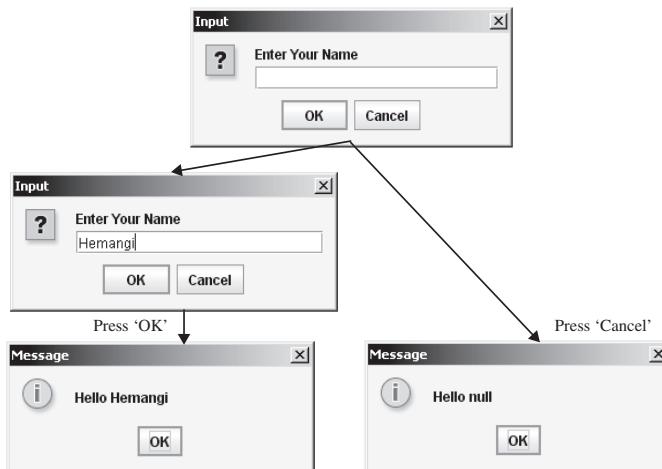


Figure 23.14 Output screen of Program 23.15

Explanation: In this program, using `showInputDialog` method, the user is prompted to enter his/her name. The returned name is prefixed with 'Hello' and displayed.

```

/*PROG 23.16 DEMO OF INPUT DIALOG VER 2 */

import javax.swing.*;
class JPS16
{
    static String input(String si)
    {
        String s = JOptionPane.showInputDialog(null, si);
        return s;
    }
    public static void main(String[] args)
    {
        String s1 = input("Enetr first number");
        String s2 = input("Enter second number");
        int sum = Integer.parseInt(s1)+Integer.parseInt(s2);
        JOptionPane.showMessageDialog(null,"Sum is "+sum);
    }
}

```

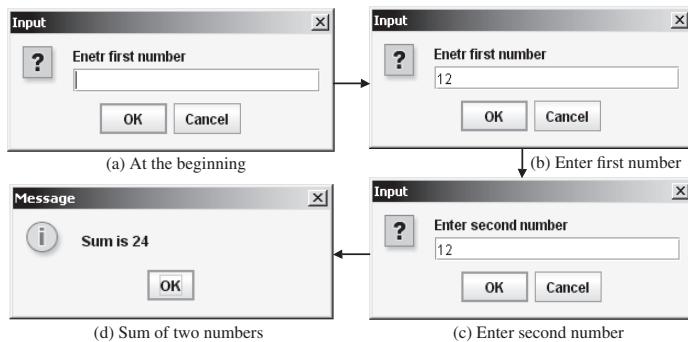


Figure 23.15 Output screen of Program 23.16

Explanation: The program finds sum of two integer numbers. The returned value is String type from showInputDialog so it first must be converted to an integer using parseInt method of Integer class.

```
/*PROG 23.17 DEMO OF INPUT DIALOG VER 3*/
import javax.swing.*;
class JPS17
{
    public static void main(String[] args)
    {
        Object[] vals = {"Trouble Free C", "Trouble Free C++", "Sea of C"};
        Icon ic = new ImageIcon("im1.jpg");
        String d = (String) JOptionPane.showInputDialog
            (null, "What you like", "Demo", 0, ic,
             vals, vals[1]);
        JOptionPane.showMessageDialog(null, "You Like"+d);
    }
}
```

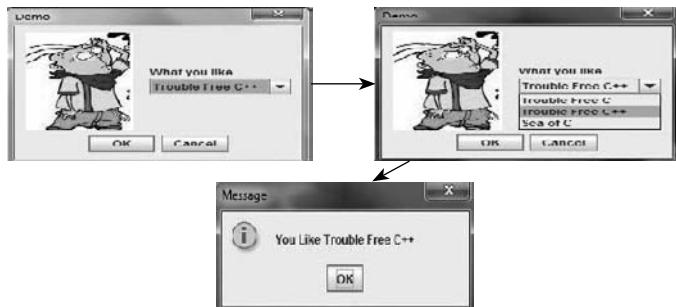


Figure 23.16 Output screen of Program 23.17

Explanation: In this program, the 6th argument in the method showInputDialog is an array of Object class created by the name vals. The next parameter is the default value or initial value selected. The array is displayed in the form of combo.

23.14 FILE SELECTION DIALOG

A `JFileChooser` is a standard file-selection box. `JFileChooser` provides a simple mechanism for the user to choose a file. As with other Swing components, `JFileChooser` is implemented in Java, so it looks and acts the same on different platforms. Some of its constructors are shown below:

- `JFileChooser()`: This form of constructor creates a `JFileChooser` pointing to the user's default directory. It is typically the "My Documents" folder on Windows, and the user's home directory on Unix.
- `JFileChooser(String path)`: This form of constructor creates a `JFileChooser` using the given path.
- `JFileChooser(File f)`: This form of constructor creates a `JFileChooser` using the given file as the path.

Some of its important methods are shown in Table 23.7.

| Method Signature | Description |
|---|---|
| <code>int showOpenDialog(Component parent)</code> | Pops up an 'Open File' file choose dialog. |
| <code>int showSave Dialog(Component parent)</code> | Pops up a 'Save File' file chooser dialog. |
| <code>File getCurrentDirectoy()</code> | Returns the current directory |
| <code>String getName(File f)</code> | Returns the file name. |
| <code>File getSelectedFile()</code> | Returns the selected file. |
| <code>File[]getSelectedFiles()</code> | Returns a list of selected files if the file chooser is set to allow multiple selections. |
| <code>void setCurrentDirectory(File dir)</code> | Sets the current directory. |
| <code>void setFileFilter(FileFilter filter)</code> | Sets the current file filter. The file filter is used by the file chooser to filter out files from the user's view. |
| <code>void setMultiSelectionEnabled(boolean b)</code> | Sets the file chooser to allow multiple file selections. |
| <code>boolean isMultiSelectionEnabled()</code> | Returns true if multiple files can be selected. |

Table 23.7 Some methods of `JFileChooser` class

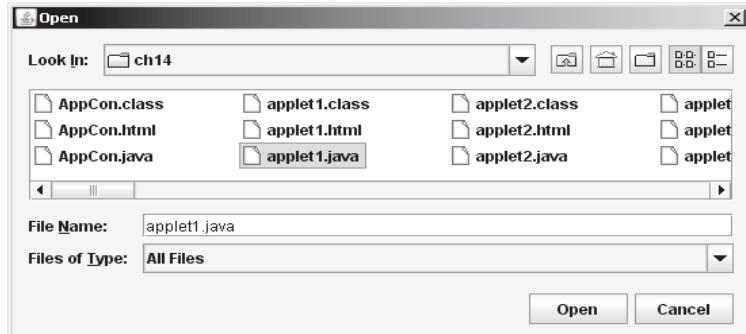
The returns type in the first two methods may be one of the constants defined by `JFileChooser` class `CANCEL_OPTION`, `APPROVE_OPTION` and `ERROR_OPTION`.

```
/*PROG 23.18 DEMO OF JFILECHOOSER CLASS VER 1*/
import javax.swing.*;
class JPS18
{
    public static void main(String[] args)
    {
        JFileChooser jfc = new JFileChooser("c:\\\\JPS");
        jfc.showOpenDialog(null);
        String sf = jfc.getSelectedFile().getName();
```

```

        JOptionPane.showMessageDialog(null, "Selected
file := "+sf);
    }
}

```



(a) Selection of file applet1.java



(b) File selection message after pressing 'open'

Figure 23.17 Output screen of Program 23.18

Explanation: The constructor form `JFileChooser("C:\JPS")`; and `showOpenDialog` together opens a file dialog box with default directory path `C:\JPS`. The `Open` button is the APPROVE button. The selected file is returned using `getSelectedFile` method. This returns a `File` reference. From this file reference, using `getName` method, file name is retrieved. The selected file name is displayed using `showMessageDialog`.

```

/*PROG 23.19 DEMO OF JFILECHOOSER CLASS VER 2 */

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
class MyFrame extends JFrame
{
    MyFrame()
    {
        super("JPS v1.0");
        JEditorPane textPane = new JEditorPane();
        Container content = getContentPane();
        content.add(new JScrollPane(textPane),
                    BorderLayout.CENTER);
    }
}

```

```

JFileChooser JFC = new JFileChooser();
int result = JFC.showOpenDialog(this);
if (result == JFileChooser.CANCEL_OPTION)
    return;
try
{
    File file = JFC.getSelectedFile();
    java.net.URL url = file.toURL();
}
catch (Exception e)
{
    textPane.setText("Could not load file:" + e);
}
setSize(300, 300);
setVisible(true);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}
class JPS19
{
    public static void main(String[] s)
    {
        new MyFrame();
    }
}
}

```

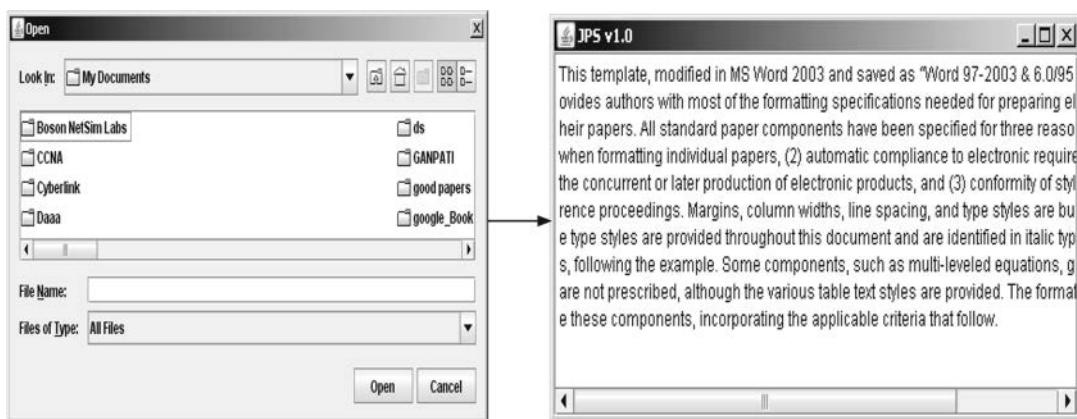


Figure 23.18 Output screen of Program 23.19

Explanation: In this program, an instance of `JEditorPane` is created and this is added as view area for the `JScrollPane` instance. An editor pane contains much more capabilities than simple text area. It lets open a URL directly into it that might be containing text of various style, other control and images.

The selected file is referenced by `File` instance `file`. The file name is then converted into URL using `toURL` method and this returned reference is stored in `URL` instance `url`. This `url` is passed as argument to `setPage` method of `JEditorPane` instance `textpane`.

23.15 THE JCOLORCHOOSEN CLASS

`JColorChooser` is yet another ready-made dialog supplied with Swing; it allows the users to choose colors. Use the `JColorChooser` class to provide users with a palette of colors to choose from. A color chooser is a component that can be placed anywhere within a program's GUI. The `JColorChooser` API also makes it easy to bring up a dialog that contains a color chooser. A brief example in the following shows how easy it is to use `JColorChooser`.

```
/*PROG 23.20 DEMO OF JCOLORCHOOSEN CLASS VER 1*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class JPS20
{
    public static void main(String[] args)
    {
        new JColorChooser().showDialog(null, "Demo",
                                       Color.blue);
    }
}
```

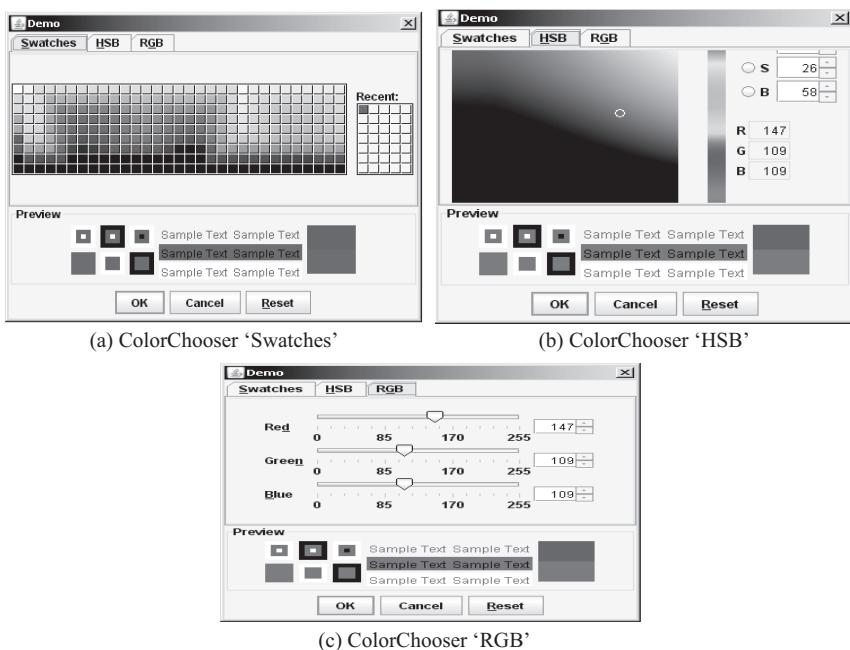


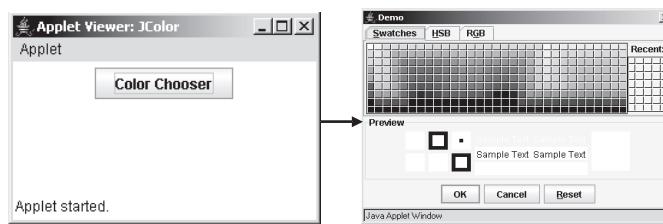
Figure 23.19 Output screen of Program 23.20

Explanation: The program simply creates a `JColorChooser` instance and displays it using `showDialog` method. The first argument in the `showDialog` method is the parent window, second is the text that appears in the title bar of the Color Chooser dialog box and last one is the default color selected.

```
/*PROG 23.21 DEMO OF JCOLORCHOOSEN CLASS VER 2 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="JColor" width = 200 height = 200>
</applet>
</html>
*/
public class JColor extends JApplet implements ActionListener
{
    JButton jb;
    Container contentPane;
    public void init()
    {
        contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        jb = new JButton("Color Chooser");
        jb.addActionListener(this);
        jb.setToolTipText("Let you choose color from
                           Color dialog box");
        contentPane.add(jb);
    }
    public void actionPerformed(ActionEvent ae)
    {
        Color col = JColorChooser.showDialog(this,"Demo",
                                              getBackground());
        contentPane.setBackground(col);
        JButton jb = (JButton)ae.getSource();
        jb.setForeground(col);
    }
}
```



(a) After running the applet (b) After clicking on 'Color Chooser'

Figure 23.20 Output screen of Program 23.21

Explanation: In this program, a button is added. On clicking the button, the color Chooser dialog box appears. The selected color is returned by the method showDialog. That color is used for filling the background of the applet window as well as foreground color of the button.

23.16 THE JTABLE CLASS

The JTable is used to display and edit regular two-dimensional tables of cells. The cursor can be dragged on column boundaries to resize columns. A column can also be dragged to a new position. Even the column's data can be changed. Two of its constructors are as shown below:

➤ **JTable(int nRows, int nCols)**

This form of constructor creates a JTable with nRows and nCols of empty cells. The columns will have names of the form 'A', 'B', 'C', etc.

➤ **JTable(Object[][] rowData, Object[] colNames)**

This form of constructor creates a JTable to display the values in the two-dimensional array, rowData, with column names, colNames. The rowData is an array of rows, so the value of the cell at row 1 and column 5 can be obtained with the following code:

```
rowData[1][5];
```

```
/*PROG 23.22 DEMO OF JTABLE CLASS */

import java.awt.*;
import javax.swing.*;
/*
<html>
<applet>
<applet code ="Jtable1" width = 200 height = 200>
</applet>
</html>
*/
public class Jtable1 extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        final String[] colHeads = {"RegNo","Name","Age","Sex"};
        final Object[][] data = {
            {"RG001", "Azamt", "20", "M"}, 
            {"RG002", "Saurabh", "21", "M"}, 
            {"RG003", "Hemangi", "22", "F"}, 
            {"RG004", "Hari", "22", "M"}, 
            {"RG005", "Deshmukh", "24", "M"}, 
            {"RG006", "Ravi", "21", "M"}, 
            {"RG007", "Aditi", "20", "F"}};
        JTable table = new JTable(data, colHeads);
        int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp = new JScrollPane(table, v, h);
        contentPane.add(jsp,BorderLayout.CENTER);
    }
}
```



Figure 23.21 Output screen of Program 23.22

Explanation: In this program, a `JTable` instance is created by the name `table`. For the header of the table, an `Object` array is created and four headers are stored in it as `String` object. For the data part of the table, a 2-D array data of `Object` class is created. In the constructor of `JTable`, the data is passed as the first argument and `colHeads` as the second argument. The table is added to the scrollpane using constructor of `JScrollPane`. The `JScrollPane` is then added to the content pane of the applet.

23.17 THE JTOOLBAR CLASS

A `JToolBar` is a container that groups several components, usually buttons with icons into a row or column. Often, tool bars provide easy access to function that is also in menus. The user can drag out a tool bar into a separate window (unless the `floatable` property is set to false). For drag-out to work correctly, it is recommended that `JToolBar` instances be added to one of the four ‘sides’ of a container whose layout manager is a `BorderLayout`, and children not be added any of the other four ‘sides’. By default, the user can drag the tool bar to a different edge of its container or out into a window of its own. If it is not wanted from the user’s side, set `Floatable` method of the `JToolbar` class is used with passing a false value as:

```
JT.setFloatable(false);
```

Where JT is an instance of JToolBar class.

```
/*PROG 23.23 DEMO OF JTOOLBAR CLASS VER 1 */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code = "toolbar1" width = 200 height = 200>
</applet>
</html>
*/
public class toolbar1 extends JApplet
{
```

```

public void init()
{
    Container contentPane = getContentPane();
    JToolBar JTB = new JToolBar();
    for(int i = 1;i<=4;i++)
    {
        JTB.add(new JButton("B"+i));
        JTB.addSeparator();
    }
    JTB.add(new JCheckBox("C1"));
    JTB.add(new JCheckBox("C2"));
    contentPane.add(JTB, BorderLayout.NORTH);
}
}

```

Explanation: The program is very simple. A JToolBar instance JTB is created and on it four JButton instances and two JCheckBox instances are added. In the end, the JTB is added to the contentPane on top of the applet window. For this purpose, BorderLayout.NORTH constant is used. No event handling is done in the program.

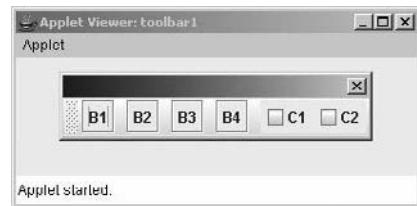


Figure 23.22 Output screen of Program 23.23

```

/*PROG 23.24 DEMO OF JTOOLBAR CLASS VER 2*/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="toolbar" width = 200 height = 200>
</applete>
</html>
*/
public class toolbar extends JApplet implements
ActionListener, ItemListener
{
    JButton JB1 = new JButton("Button 1", new
                                ImageIcon("smile1.jpg"));
    JButton JB2 = new JButton("Button 2", new
                                ImageIcon("smile2.jpg"));
    JComboBox JCom = new JComboBox();
    public void init()
    {
        Container contentPane = getContentPane();
        JToolBar jtoolbar = new JToolBar();
        JB1.addActionListener(this);

```

```

        JB2.addActionListener(this);
        JCom.addItem("Item 1");
        JCom.addItem("Item 2");
        JCom.addItem("Item 3");
        JCom.addItem("Item 4");
        JCom.addItemListener(this);
        jtoolbar.add(JB1);
        jtoolbar.addSeparator();
        jtoolbar.add(JB2);
        jtoolbar.addSeparator();
        jtoolbar.add(JCom);
        contentPane.add(jtoolbar, BorderLayout.NORTH);
    }
    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == JB1)
        {
            showStatus("You Clicked button 1");
        }
        else
        {
            if (e.getSource() == JB2)
            {
                showStatus("You clicked button 2");
            }
        }
    }
    public void itemStateChanged(ItemEvent e)
    {
        showStatus("Selected: " + (String)e.getItem());
    }
}

```

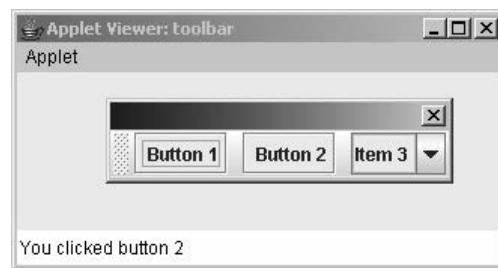


Figure 23.23 Output screen of Program 23.24

Explanation: In this program, onto the JToolBar instance two image buttons are added with one combo box. Listeners are added for the controls so when any of the item is selected, event handlers are invoked. For the button clicked, actionPerformed method is called; and for combo box, itemStateChanged method is called.

As mentioned earlier, the tool bar can be detached from its place and placed anywhere onto the screen. This is illustrated in the figure given below where the tool bar is shown detached and placed outside the applet window. Closing the close button on tool bar brings it back to its original position. Placement of tool bar does not affect its working.

23.18 THE JPROGRESSBAR CLASS

Sometime a task running within a program might take a while to complete. A user-friendly program provides some indication to the user that the task is occurring, how long the task might take and how much work has already been done. A progress bar typically communicates the progress of some work by displaying its percentage of completion and possibly a textual display of this percentage. The JProgressBar is a swing component that does this task.

Some of the constructors of this class are as follows:

- `JProgressBar()` : This form of constructor creates a horizontal progress bar that displays a border but no progress string. The initial and minimum values are 0, and the maximum is 100.
- `JProgressBar(int orient)` : This form of constructor creates a progress bar with the specified orientation, which can be either `JProgressBar.VERTICAL` or `JProgressBar.HORIZONTAL`.
- `JProgressBar(int min, int max)` : This form of constructor creates a horizontal progress bar with the specified minimum and maximum.
- `JProgressBar(int orient, int min, int max)` : This form of constructor creates a progress bar using the specified orientation, minimum and maximum.

Some of the methods are shown in the Table 23.8.

| Method Signature | Description |
|---|---|
| <code>void addChnageListener(ChangeListener L)</code> | Adds the specified ChangeListener to the progress bar. |
| <code>int getMaximum()</code> | Returns the progress bar's maximum value. |
| <code>int getMinimum()</code> | Returns the progress bar's minimum value. |
| <code>int getOrientation()</code> | Returns <code>JProgressBar.VERTICAL</code> or <code>JProgressBar.HORIZONTAL</code> , depending on the orientation of the progress bar. |
| <code>int getValue()</code> | Returns the progress bar's current value. |
| <code>void paintBorder(Graphics g)</code> | Paints the progress bar's border if the borderPainted property is true. |
| <code>void setBorderPainted(Boolean b)</code> | Sets the borderPainted property, which is true if the progress bar should paint its border. |
| <code>void setMaximum(int n)</code> | Sets the progress bar's maximum value. |
| <code>void setMinimum(int n)</code> | Sets the progress bar's minimum value. |
| <code>void setOrientation(int new Orientation)</code> | Sets the progress bar's orientation to newOrientation, which must be <code>JProgressBar.VERTICAL</code> or <code>JProgressBar.HORIZONTAL</code> |
| <code>void setString(String s)</code> | Sets the value of the progress string. |
| <code>void setValue(int n)</code> | Sets the progress bar's current value. |

Table 23.8 Some methods of `JProgressBar` class

In order to receive events from the `JProgressBar` class, the class must implement `ChangeListener` and a change listener must be added for the `JProgressBar` instance using `addChangeListener` method. See some examples:

```
/*PROG 23.25 DEMO OF JPROGRESSBAR CLASS VER 1 */
```

```
import java.awt.*;
import javax.swing.*;
/*
```

```
<html>
<applet>
<applet code ="Pbar1" width = 200 height = 200>
</applet>
</html>
*/
public class Pbar1 extends JApplet
{
    JProgressBar JPB1, JPB2, JPB3, JPB4;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        //First
        JPB1 = new JProgressBar();
        JPB1.setValue(50);
        contentPane.add(JPB1);
        //Second
        JPB2 = new JProgressBar();
        JPB2.setMinimum(100);
        JPB2.setMaximum(200);
        JPB2.setValue(180);
        JPB2.setStringPainted(true);
        JPB2.setForeground(Color.green);
        contentPane.add(JPB2);
        //Third
        JPB3 = new JProgressBar();
        JPB3.setForeground(Color.magenta);
        JPB3.setValue(50);
        JPB3.setStringPainted(true);
        JPB3.setBorder(BorderFactory.createEtchedBorder());
        contentPane.add(JPB3);
        //Fourth
        JPB4 = new JProgressBar();
        JPB4.setOrientation(JProgressBar.VERTICAL);
        JPB4.setStringPainted(true);
        JPB4.setString("Hello from Progress Bar");
        JPB4.setValue(90);
        contentPane.add(JPB4);
    }
}
```

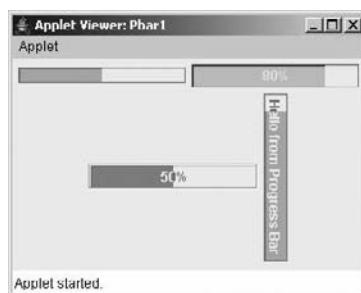


Figure 23.24 Output screen of Program 23.25

Explanation: The program uses various methods of ProgressBar and displays it in a number of forms. In the first form, a simple progress bar is created and its value is set at position 50. In the second form, the minimum value is set at 100 and maximum value at 200. The current value set is at 180. The `setStringPainted` method makes percentage string that is to be displayed within progress bar. In the third form, the border is set for the progress bar using `BorderFactory` class. The class contains a number of methods for setting the border around any component. Here, etched border is used.

```
/*PROG 23.26 DEMO OF JPROGRESSBAR CLASS VER 2 */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
/*
<html>
<applet>
<applet code ="PBar2" width = 200 height = 200>
</applet>
</html>
*/
public class PBar2 extends JApplet implements ActionListener,
ChangeListener
{
    JProgressBar JPB;
    JButton Jbut;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JPB = new JProgressBar();
        Jbut = new JButton("Clicke Me");
        contentPane.add(Jbut);

        JPB.setForeground(Color.green);
        contentPane.add(JPB);
        Jbut.addActionListener(this);
        JPB.addChangeListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        JPB.setValue(JPB.getValue() + 5);
    }
    public void stateChanged(ChangeEvent e)
    {
        showStatus("Max: " + JPB.getMinimum() + " Min: "
                  + JPB.getMaximum() + " Value: "
                  + JPB.getValue());
    }
}
```

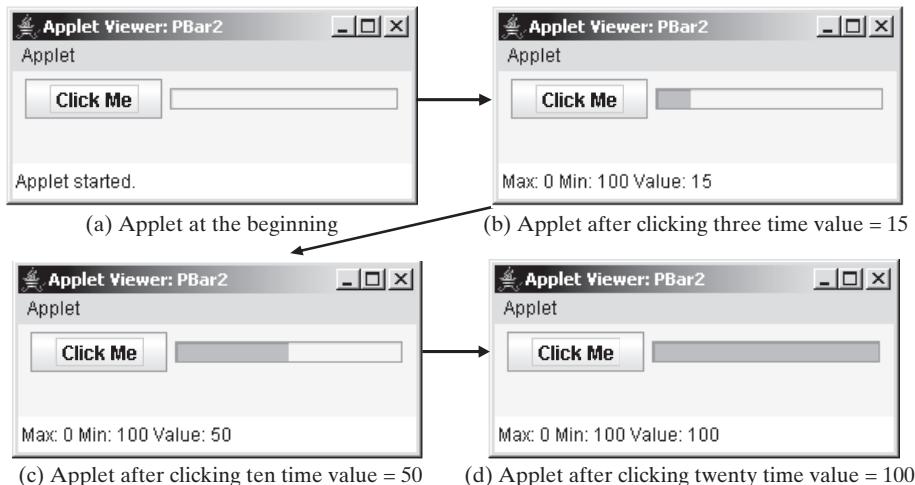


Figure 23.25 Output screen of Program 23.26

Explanation: In this program, the current value of the progress bar is incremented by 5 on every button click. When the button is clicked, current value of the progress bar is taken using `getValue` method and after adding 5 to it, current value is set using `setValue` method. When the value in progress bar changes, `stateChanged` method is called.

```
/*PROG 23.27 DEMO OF JPROGRESSBAR CLASS VER 3 */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="PBar3" width = 200 height = 200>
</applet>
</html>
*/
public class PBar3 extends JApplet implements ActionListener
{
    JProgressBar JPB;
    JButton Jbut;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JPB = new JProgressBar();
        Jbut = new JButton("Click Me");
    }
}
```

```

JPB.setForeground(Color.blue);
contentPane.add(Jbut);
contentPane.add(JPB);

JPB.setStringPainted(true);
Jbut.addActionListener(this);

}

public void actionPerformed(ActionEvent e)
{
    JPB.setValue(JPB.getValue() + 10);
    showStatus(JPB.getString() + " Completed");
}

}

```

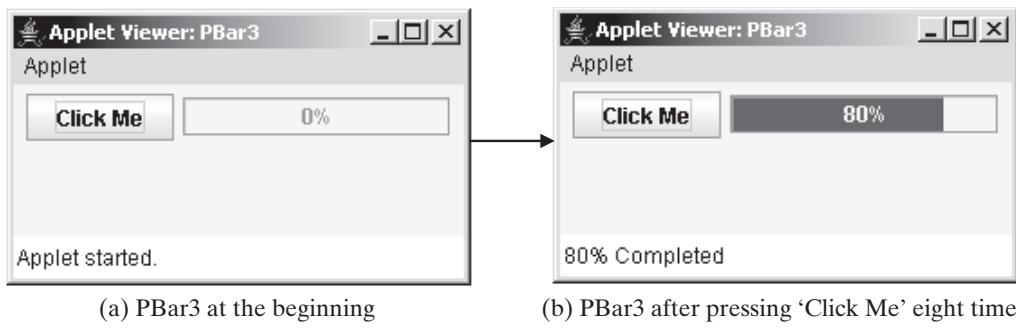


Figure 23.26 Output screen of Program 23.27

Explanation: There is a small variation in this program from the previous one. Here the value for the progress bar is set in the `actionPerformed` method and percentage completion is shown in the applet status window using `getString` method. Note to show the percentage string in the progress bar, the `setStringPainted` must be used with true argument.

23.19 THE JSLIDER CLASS

An instance of `JSlider` class is a component that lets the user graphically select a value by sliding a knob within a bounded interval. The slider can show both major and minor tick marks between them. Some of the constructors of this class are as shown below:

- `JSlider()` : This form of constructor creates a horizontal slider with the range 0 to 100 and an initial value of 50.
- `JSlider(int orient)` : This form of constructor creates a slider using the specified orientation with the range 0 to 100 and an initial value of 50.
- `JSlider(int min, int max)` : This form of constructor creates a horizontal slider using the specified min and max with an initial value equal to the average of the min plus max.

| Method Signature | Description |
|--------------------------------------|--|
| boolean getInverted() | Returns true if the value range shown for the slider is reversed. |
| int getMajorTickSpacing() | Returns the major tick spacing. |
| int getMaximum() | Returns the maximum value supported by the slider. |
| int getMinimum() | Returns the minimum value supported by the slider. |
| int getMinorTick(boolean b) | Returns the minor tick spacing. |
| void setInverted(boolean b) | Specifies true to reverse the value range shown for the slider and false to put the value range in the normal order. |
| void setLabelTable(Dictionary label) | Specifies what label will be drawn at any given value. |
| void setMajorTickSpacing(int n) | Sets the major tick spacing. |
| Void setMaximum(int maximum) | Sets the maximum value for slider. |
| void setMinimum(int minimum) | Sets the minimum value for slider. |
| void setMinorTickSpacing(int n) | Sets the minor tick spacing. |
| void setOrientation(int orientation) | Sets the scrollbar orientation to either VERTICAL or HORIZONTAL. |
| void setPaintLabels(boolean b) | Determines whether labels are painted on the slider. |
| void setPaintTicks(boolean b) | Determines whether tick marks are painted on the slider. |

Table 23.9 Some methods of JSlider class

```
/*PROG 23.28 DEMO OF JSLIDER CLSS VER 1 */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
/*
<html>
<applet>
<applet code =slider width = 200 height = 200>
</applete>
</html>
*/
public class slider extends JApplet implements ChangeListener
{
    JSlider JS;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JS = new JSlider(SwingConstants.HORIZONTAL, 0,
                         100, 0);
        JS.addChangeListener(this);
    }
}
```

```

        JS.setPaintTicks(true);
        JS.setPaintLabels(true);
        JS.setMajorTickSpacing(20);
        JS.setMinorTickSpacing(4);
        contentPane.add(JS);
    }
    public void stateChanged(ChangeEvent e)
    {
        JSlider Js = (JSlider) e.getSource();
        String s = "Min: "+Js.getMinimum();
        s+= " Max: "+Js.getMaximum();
        s+= " Value: "+Js.getValue();
        showStatus(s);
    }
}

```

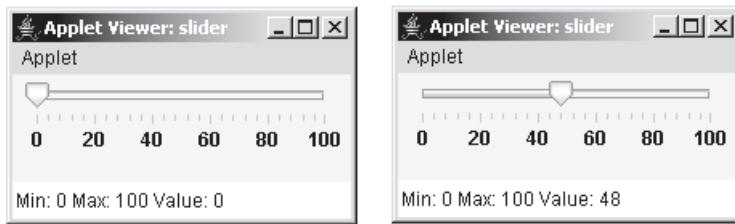


Figure 23.27 Output screen of Program 23.28

Explanation: In this program, an instance of JSlider is created. The orientation of slider is horizontal, min value is 0, max value is 100 and initial value is 0. The class implements ChangeListener interface that is required for slider component. The setPaintTicks and setPaintLabels are used for displaying the tick marks and labels for the slider. When the slider is moved, listeners are notified and stateChanged method is called. In the method using getSource method, reference to slider instance is obtained. Recall to getSource returns the reference of Object type so typecasting by JSlider is a must. In the method minimum, maximum and current value of the slider are displayed.

```
/*PROG 23.29 DEMO OF JSLEIDER CLASS VER 2*/
```

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.util.*;
/*
<html>
<applet>
<applet code ="slider2" width = 200 height = 200>
</applet>
</html>
*/
public class slider2 extends JApplet

```

```

{
    JSlider JS;
    final int S_Min = 0;
    final int S_Max = 150;
    final int S_Ini = 0;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JS = new JSlider(SwingConstants.VERTICAL, S_Min,
                          S_Max, S_Ini);
        JS.setPaintTicks(true);
        Hashtable Ltab = new Hashtable();

        Ltab.put(new Integer(0), new JLabel("stop"));
        Ltab.put(new Integer(S_Max / 5), new
                 JLabel("Slow"));
        Ltab.put(new Integer(S_Min / 3 + 10), new
                 JLabel("Medium"));
        Ltab.put(new Integer(S_Max), new JLabel("Fast"));

        JS.setLabelTable(Ltab);
        JS.setPaintLabels(true);
        JS.setPaintLabels(true);
        JS.setMajorTickSpacing(30);

        contentPane.add(JS);
    }
}

```

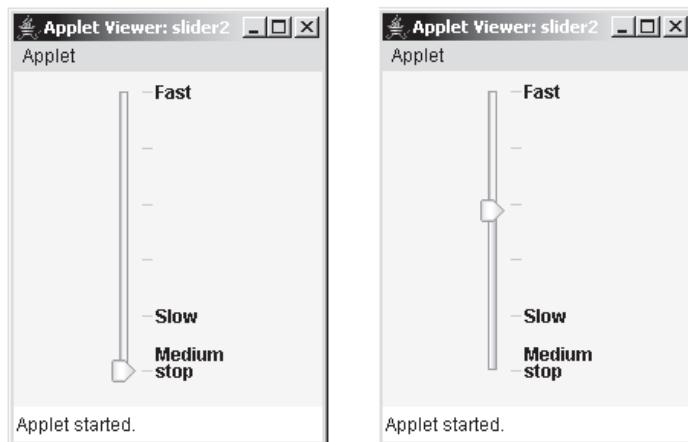


Figure 23.28 Output screen of Program 23.29

Explanation: The program is similar to the previous one but here it has been shown how to set the labels at various positions. This has been done by creating an instance of Hashtable and adding entry as key-value

pair. Each key-value pair in the hashtable specified with `setLabelTable` gives the position and the value of one label. The hashtable key must be an integer and a value within the slider's range at which to place the label. The hashtable value associated with each key must be a Component. This program uses `JLabel` instance with text only.

23.20 THE JTREE CLASS

The `JTree` is a control that displays a set of hierarchical data as an outline. A `JTree` object does not actually contain the data; it simply provides a view of the data. Like any non-trivial Swing component, the tree gets data by querying its data model.

As the figure shows, `JTree` displays its data vertically. Each row displayed by the tree contains exactly one item of data, which is called a node. Every tree has a root node from which all nodes descend. By default, the tree displays the root node, but it can be decreed otherwise. A node can either have children or not. Those nodes are referred to that can have children—whether or not they currently have children—as branch nodes. Nodes that cannot have children are leaf nodes.

Branch nodes can have any number of children. Typically, the user can expand and collapse branch nodes—making their children visible or invisible—by clicking them. By default, all branch nodes except the root node start out collapsed. A program can detect changes in the expansion state of branch nodes by listening for tree expansion or tree-will-expand events.

Two of its constructors are shown below:

- `JTree ()` : This form of constructor returns a `JTree` with a sample model.
- `JTree (TreeModel newModel)` : This form of constructor returns an instance of `JTree` which displays the root node—the tree is created using the specified data model.

Due to its complexity, only two examples of `JTree` will be given and its events are not discussed.



```

/*PROG 23.30 DEMO OF JTREE CLASS VER 1 */

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.*;
public class TreeDemo1
{
    public static void main(String args[])
    {
        JFrame frame = new JFrame("Tree Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JTree tree = new JTree();
        JScrollPane pane = new JScrollPane(tree);
        frame.getContentPane().add(pane);
        frame.setSize(250, 250);
        frame.setVisible(true);
    }
}
  
```

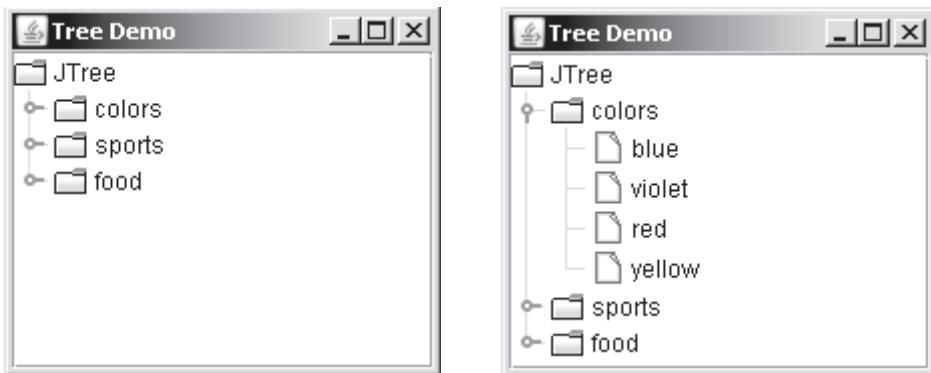


Figure 23.29 Output screen of Program 23.30

Explanation: In this program, a simple tree containing only a few items is created that has no ability to add or remove items. A scrollbar pane is included. The icons/tree nodes shown in this example are defaults provided by Swing. An item is not actually inserted in the tree. When no specific data model has been specified, JTree reverts to a default example model. When the program is executed, notice that the tree is initially collapsed down to the root level. Opening the root expands it, showing the next level of indentation. Notice the change in icon for the lowest element of the tree. This is the default operation for the simple tree.

```
/*PROG 23.31 DEMO OF JTREE CLASS VER 2 */

import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
public class TreeDemo2 extends JFrame
{
    JTree m_tree = null;
    DefaultTreeModel m_model = null;
    public TreeDemo2()
    {
        super("Tree Demo2");
        DefaultMutableTreeNode top = new
            DefaultMutableTreeNode("College");
        DefaultMutableTreeNode parent = top;
        DefaultMutableTreeNode node = new
            DefaultMutableTreeNode("Administration");
        parent.add(node);
        node = new DefaultMutableTreeNode("Security");
        parent.add(node);
        node = new DefaultMutableTreeNode("Deptt");
        parent.add(node);
        parent = node;
    }
}
```

```
node = new DefaultMutableTreeNode("Computer");
parent.add(node);
parent = node;
node = new DefaultMutableTreeNode("Staff");
parent.add(node);
parent = node;
node = new
DefaultMutableTreeNode("Asst. Professor");
parent.add(node);

node = new DefaultMutableTreeNode("Lecturers");
parent.add(node);
node.add(new DefaultMutableTreeNode("Prof. N.S.
Choubey"));
node.add(new DefaultMutableTreeNode("Prof.
H.M. Pandey"));
node.add(new DefaultMutableTreeNode("Prof.
Hemangi Patil"));
node.add(new DefaultMutableTreeNode("Prof. Varsha
Nemade"));
node.add(new DefaultMutableTreeNode("Prof. Bipin
Jadhav"));
node.add(new DefaultMutableTreeNode("Prof.
Vishalli Patil"));
node.add(new DefaultMutableTreeNode("Prof. Yogesh
Choudhary"));
node = new DefaultMutableTreeNode("Professor");

parent.add(node);
node = new
DefaultMutableTreeNode("Techinicians");
parent.add(node);
m_model = new DefaultTreeModel(top);
m_tree = new JTree(m_model);
JScrollPane s = new JScrollPane(m_tree);
getContentPane().add(s, BorderLayout.CENTER);
setSize(300, 270);
setVisible(true);
}
public static void main(String argv[])
{
    TreeDemo2 td = new TreeDemo2();

    td.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}
```

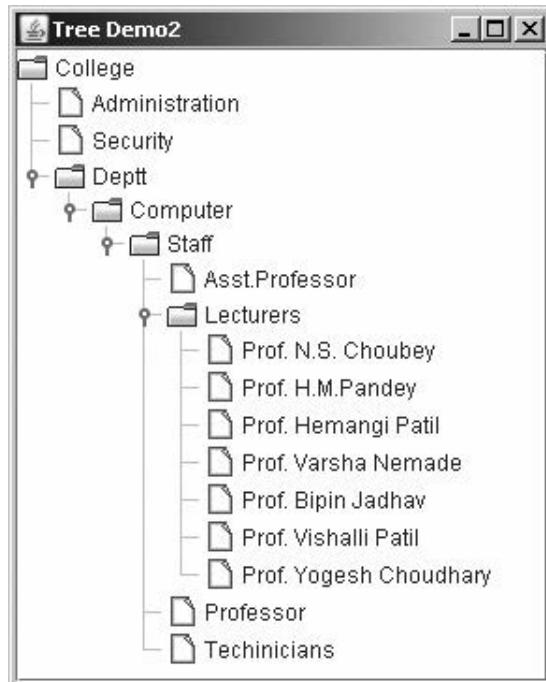


Figure 23.30 Output screen of Program 23.31

Explanation: The `DefaultMutableTreeNode` is a class defined in the package `javax.swing.tree`. Instances of this class are general-purpose node in a tree data structure. `DefaultMutableTreeNode` provides operations for examining and modifying a node's parent and children and also operations for examining the tree that the node is a part of. A node's tree is the set of all nodes that can be reached by starting at the node and following all the possible links to parents and children. A node with no parent is the root of its tree; a node with no children is a leaf. A tree may consist of many subtrees, each node acting as the root for its own subtree.

In this program and from the output, observe how various nodes have been added to the tree and subtrees. Note the root node is referenced by top. Using the following lines:

```
m_model = new DefaultTreeModel(top);
m_tree = new JTree(m_model);
```

A `DefaultTreeModel` is created in which `top` is passed as argument. The `DefaultTreeModel` creates a tree in which any node can have children. The `m_model` is then passed to the constructor of `JTree` class and reference returned is stored in `m_tree`.

The `JTree` reference `m_tree` is then passed to the constructor of `JScrollPane` class as done in the previous program. The `JScrollPane` instance `s` is then added onto the frame's content pane. Coding in main is simple to understand.

23.21 EXAMPLES OF MENUS

Creation of menus is similar to the way seen earlier in the chapter dealing with AWT. Here, few programs are presented that contain some new features provided by swing.

```
/*PROG 23.32 DEMO OF CREATING MENUS USING SWING VER 1*/
```

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="menuRB1" width = 200 height = 200>
</applet>
</html>
*/
public class menuRB1 extends JApplet implements ItemListener
{
    ImageIcon icon;
    int i;
    JRadioButtonMenuItem JRBM[];
    public void init()
    {
        Container contentPane = getContentPane();
        icon = new ImageIcon("im4.jpg");
        JRBM = new JRadioButtonMenuItem[4];
        ButtonGroup group = new ButtonGroup();

        JMenuBar jmenubar = new JMenuBar();

        JMenu jmenu = new JMenu("Demo_Menu");
        for (i = 0; i < JRBM.length; i++)
        {
            JRBM[i]=new JRadioButtonMenuItem("Item"+(i+1),
                icon);
            group.add(JRBM[i]);
            jmenu.add(JRBM[i]);
            JRBM[i].addItemListener(this);
        }
        jmenubar.add(jmenu);
        setJMenuBar(jmenubar);
    }
    public void itemStateChanged(ItemEvent e)
    {
        JMenuItem JMI = (JMenuItem)e.getSource();
        showStatus("Selected item:" + JMI.getText());
    }
}
```

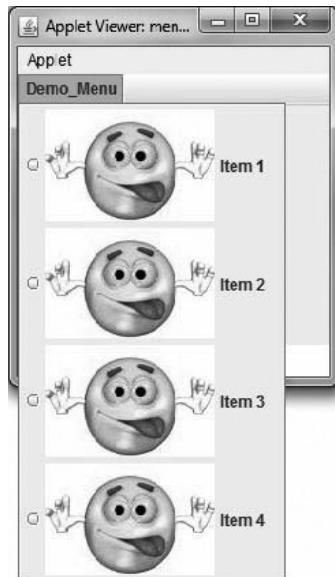


Figure 23.31 Output screen of Program 23.32

Explanation: This program demonstrates how radio button menu items can be added to the menu. The `JRadioButtonMenuItem` class can be used for this purpose. An array `JRBM` of size 4 of `JRadioButtonMenuItem` class is created. Using for loop menu, items of `JRadioButtonMenuItem` is created with name along with .jpg image ‘im4.jpg’. Before for loop, an instance each of `JMenuBar` and `JMenu` is created. Menu items created within for loop are added to the `JM`. Outside the for loop, `JMB` is added to the window using `setJMenuBar` method. As all menu items are placed within group, only one radio button menu item is selected at a time. When any of the menu items is selected, `itemStateChanged` method is called as listeners are added for every menu item. In the method, which item was selected is displayed using `showStatus` method.

```
/*PROG 23.33 DEMO OF CREATING MENUS USING SWING VER 2 */
```

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="menuIMG1" width = 200 height = 200>
</applete>
</html>
*/
public class menuIMG1 extends JApplet implements
ActionListener
{
    public void init()
```

```
{  
    JMenuBar JMB = new JMenuBar();  
  
    JM = new JMenu("File");  
    ImageIcon icon = new ImageIcon("smile3.jpg");  
    JMenuItem JMI1 = new JMenuItem("New.....",icon);  
    JMenuItem JMI2 = new JMenuItem("Open....",icon);  
  
    JM.add(JMI1);  
    JM.addSeparator();  
    JM.add(JMI2);  
  
    JMI1.setActionCommand("You selected New");  
    JMI2.setActionCommand("You selected Open");  
  
    JMI1.addActionListener(this);  
    JMI2.addActionListener(this);  
  
    JMB.add(JM);  
    setJMenuBar(JMB);  
}  
public void actionPerformed(ActionEvent e)  
{  
    JMenuItem JMI = (MenuItem)e.getSource();  
    showStatus(JMI.getActionCommand());  
}  
}
```

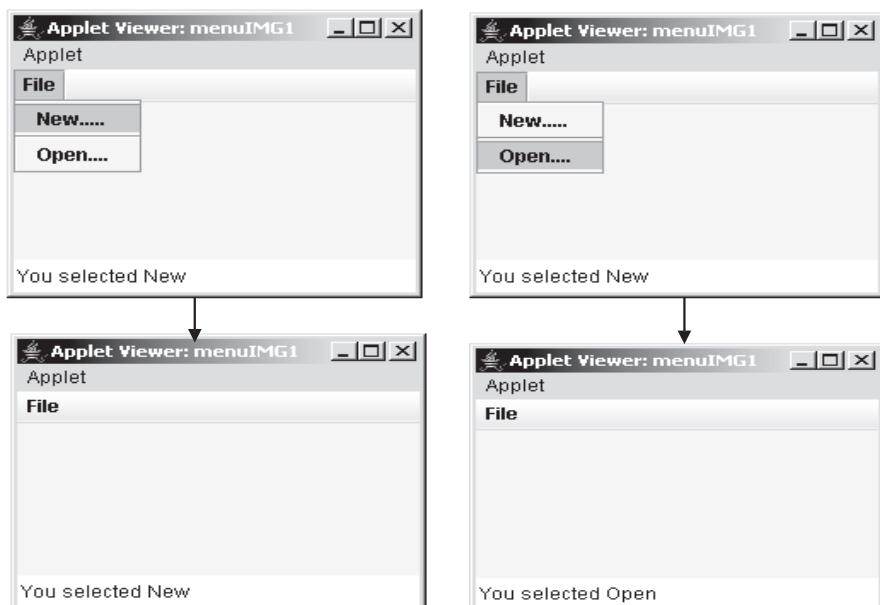


Figure 23.32 Output screen of Program 23.33

Explanation: This program is run for creating a simple menu consisting of two menu items but along with the text, images can also be displayed in the menu items. For this purpose, constructor of JMenuItem class is used that takes second argument of Icon type. The setActionCommand and getActionCommand are inherited from the AbstractButton class, which have already been discussed.

```
/*PROG 23.34 DEMO OF CREATING MENUS USING SWING VER 3 */

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*
<html>
<applet>
<applet code ="menuUP" width = 200 height = 200>
</applet>
</html>
*/
public class menuUP extends JApplet implements ActionListener
{
    JMenuBar JMB = new JMenuBar();
    JMenu JM = new JMenu("Update");
    JMenuItem JMI1 = new JMenuItem("Add item");
    JMenuItem JMI2 = new JMenuItem("Remove item");
    JMenuItem JMI3 = new JMenuItem("New item");
    public void init(){
        JM.add(JMI1);
        JM.add(JMI2);

        JMI1.addActionListener(this);
        JMI2.addActionListener(this);

        JMB.add(JM);
        setJMenuBar(JMB);
    }
    public void actionPerformed(ActionEvent e){
        JMenuItem JMI = (JMenuItem)e.getSource();
        if (JMI == JMI1){
            JM.add(JMI3);
            showStatus("Item added");
        }
        if (JMI == JMI2){
            JM.remove(JMI3);
            showStatus("Item removed");
        }
    }
}
```

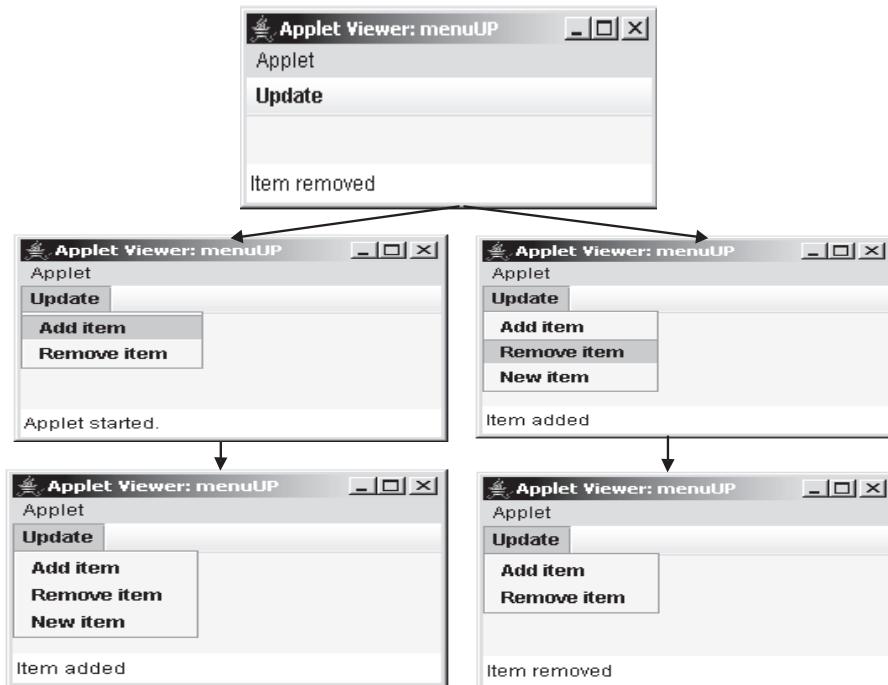


Figure 23.33 Output screen of Program 23.34

Explanation: When the first menu item labelled ‘**Add item**’ is selected, actionPerformed method is called and a new menu item is added using the add method. The menu item was already created in the beginning of the class. Similarly when second menu item labelled ‘**Remove item**’ is selected, the newly added item is removed, from the menu using the remove method.

23.22 PONDERABLE POINTS

1. A swing is a set of classes that provides more powerful and flexible components than is possible with the AWT. It is defined within the package javax.swing.
2. As compared to AWT components, swing components are known as lightweight components.
3. The JApplet class is an extended version of java.applet. Applet that adds support for the JFC/Swing component architecture.
4. The class ImageIcon is an implementation of the Icon interface that paints Icons from Image created from a URL, filename or byte array.
5. The SwingConstants is an interface defined in the javax.swing package. It defines various constants related with the shape and alignment of the component.
6. All the swing components can make use of images as icon as they are drawn.
7. The JTabbedPane class is a component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon.
8. A JScrollPane is a container that can hold one component. In other words, a JScrollPane wraps another component. It presents a rectangular area in which a component may be viewed. By default, if the wrapped component is larger than the JScrollPane itself, the JScrollPane supplies scrollbars.
9. A split pane is a special container that holds two components, each in its own sub-pane. A splitter bar adjusts the sizes of the two sub-panes.

10. Dialogs are frequently used to present information to the user to ask a question. These are accessible from static methods in the `JOptionPane` class. `JOptionPane` makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something.
11. A `JFileChooser` is a standard file-selection box. It provides a simple mechanism for the user to choose a File.
12. The `JColorChooser` class allows an application to choose colors. It is used to provide users with a palette of colors to choose from.
13. The `JTable` is used to display and edit regular two-dimensional tables of cells. The cursor can be dragged on column boundaries to resize columns. A column can also be dragged to a new position.
14. A `JToolBar` is a container that groups several components, usually buttons with icons into a row or columns.
15. A progress bar typically communicates the progress of some work by displaying its percentage of completion and possibly a textual display of this percentage. The `JProgressBar` is a swing component that does this task.
16. An instance of `JSlider` class is a component that lets the user to graphically select a value by sliding a knob within a bounded interval.
17. The `JTree` is a control that displays a set of hierarchical data as an outline. A `JTree` object does not actually contain the data; it simply provides a view of the data.

REVIEW QUESTIONS

1. What is the difference between AWT and Swing?
2. In what way JButton is better than Button Class? Explain with an example.
3. Write a program to display the names of the Java files in C:
4. Design a text editor similar to Notepad implementing Swing, event and IO File handling.
5. Write a Java program to display the following 3×3 magic square (total = 15) using `JTable`:
6. Write a program to display the month names by `JList` and display the Days by `JComboBox`.
7. Write Java application using slider control to change the current Fahrenheit to centigrade and display the result in `JLabel`.
8. How can you create a button with an image loaded on it and how do you set a tool tip to a button control?
9. Explain any two event Listener and its methods.

| | | |
|---|---|---|
| 2 | 9 | 4 |
| 7 | 5 | 3 |
| 6 | 1 | 8 |

Multiple Choice Questions

1. The swing related classes are contained in
 - `javax.swing`
 - `javax.awt`
 - `javax.Swing`
 - None of the above
2. Which method is used to return if the value range shown for the slider is reversed?
 - `void setLabelTable(Dictionary labels)`
 - `boolean getInverted()`
 - `void setInverted(boolean b)`
 - None of the above
3. The constructor of `JTree` class is
 - `public void JTree()`
- (b) `public JTree()`
 (c) `JTree(TreeModel newModel)`
 (d) `JTree(boolean b)`
4. Which method is used to return the progress bar's current value?
 - `void getValue(int val)`
 - `public void getValue()`
 - `int getValue()`
 - `public int getValue()`
5. The `setFloatable` method belong to
 - `JToolBar` class
 - `JTable` class
 - `JProgressBar` class
 - None of the above

6. Which is the constructor of JComboBox?
 (a) JComboBox(int items)
 (b) JComboBox(Vector items)
 (c) JComboBox(int [] items)
 (d) None of the above
7. Which is the correct signature for JScrollPane class construction?
 (a) JScrollPane(int vsb, int hsb)
 (b) public JScrollPane(Object vsb, Object hsb)
 (c) public void JScrollPane(int vsb, int hsb)
 (d) public void JScrollPane(Object vsb, Object hsb)
8. Which is the correct form for showMessageDialog method?
 (a) void showMessageDialog(Component comp)
 (b) public void showMessageDialog(Component comp, Object msg)
- (c) static void showMessageDialog(Component comp, Object msg)
 (d) public static void showMessageDialog(Component comp, Object msg)
9. Which method is used to set the file chooser to allow multiple file selections?
 (a) void setMultiSelectionEnabled(boolean b)
 (b) boolean isMultiSelectionEnabled()
 (c) public void setMultiSelectionEnabled(boolean b)
 (d) None of the above
10. The method void setValue(int n) is used for
 (a) Setting the current value of progress bar
 (b) Setting the current value of combo box
 (c) Setting the current value of Text box
 (d) None of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. a 2. b 3. c 4. c 5. a 6. b 7. b 8. c 9. a 10. a

24

Introduction to Virtual Machine and API Programming

24.1 INTRODUCTION

The programming API is an API that let users to write scripts and programs to manipulate virtual machines. It is a high-level interface which is easy to use and practical for both script developers and application programmers.

The Java platform has two main parts, the Java virtual machine and the Java API, as shown in Figure 24.1.

1. Java virtual machine

The Java virtual machine is a ‘soft’ computer that can be implemented in software or hardware. It is an abstract machine designed to be implemented on top of existing processors. The porting interface and adapters enable it to be easily ported to new operating system without being completely rewritten.

2. Java API

The Java API forms a standard interface to applets and applications, regardless of the underlying operating system. The Java API is the essential framework for application development. This API specifies a set of essential interfaces in a growing number of key areas that developers will use to build their Java-powered applications.

- *The Java base API:* It provides the very basic language, utility, I/O, network, GUI and applet services; OS companies that have licensed Java have contracted to include them in any Java platform they deploy.

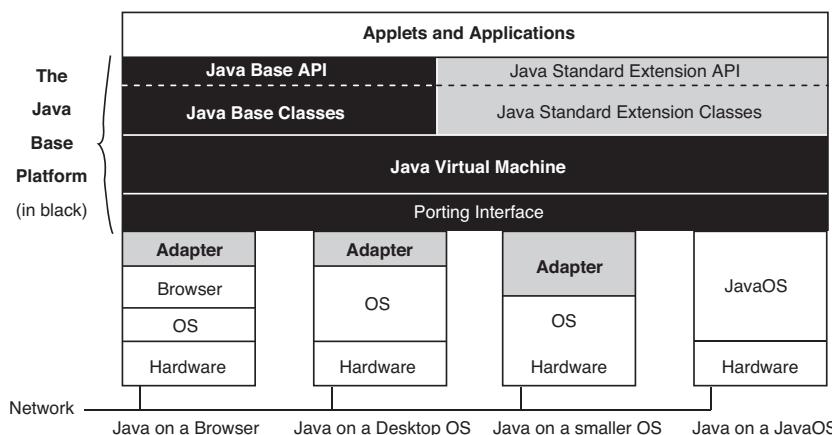


Figure 24.1 The Java base platform is uniform across all operating systems

- *The Java standard extension API:* It extends the capabilities of Java beyond the Java base API. Some of these extensions will eventually migrate to the Java base API. Other non-standard extension APIs can be provided by the applet, application or underlying operating system. As each new extension API specification is published, it will be made available for industry review and feedback before it is finalized.

In Figure 24.1, the Java base platform is the part shown in black, including the blocks labelled Adapter (platform dependent). The Java API includes both the Java base API and Java standard extension API. The classes are the implementation of that API. The Java virtual machine is at the core of the platform. The porting interface lies between the Java virtual machine and the operating system (OS) or browser. This porting interface has a platform independent part as shown in black and a platform dependent part shown as adapters. The OS and JavaOS provide the window, filling and network functionality. Different machines can now be connected by a network.

The Java API framework is open and extensible. The API is organized by groups or sets. Each of the API sets can be implemented as one or more packages which are called as namespaces. Each package groups together a set of classes and interfaces that define a set of related fields constructors and methods.

24.2 VIRTUAL MACHINE

The Java virtual machine is key to the independence of the underlying operating system and hardware; it is a platform that hides the underlying operating system from Java-powered applets and applications. And it is very easy to port the virtual machine to a browser or another operating system.

In addition, the virtual machine defines a machine independent format for binary files which is called as '.class' file format. This format includes instructions for a virtual computer in the form of bytecodes. The bytecode representation of any Java language program is symbolic in that it offsets and indexes into methods that are not constants, but are instead given symbolically as string names. The first time a method is called, it is searched by name in the class file format, and its offset numeric value is determined at that time for quicker access at subsequent lookups. Therefore, any new or overriding method can be introduced late at runtime anywhere in the class structure and it will be referred to symbolically, and properly accessed without breaking the code.

```
/* PROG 24.1 SUM OF TWO NUMBERS (DEMO OF VIRTUAL MACHINE) */

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code = "button4" width = 280 height = 140>
</applet>
*/
public class JPS2 extends Applet implements ActionListener
{
    Button but1;String result=" ";
    boolean in=false;
    int num1, num2;
    TextField tf1, tf2;
    Label L1, L2;
    public void init()
    {
```

```

        setBackground(Color.blue);
        L1=new Label("Enter first number:");
        add(L1);
        tf1=new TextField(10);
        add(tf1);
        L2=new Label("Enter second number:");
        add(L2);
        tf2=new TextField(10);
        add(tf2);
        but1=new Button("Click for maximum of two no.");
        add(but1);
        but1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent AE)
    {
        try
        {
            num1=Integer.parseInt(tf1.getText());
            num2=Integer.parseInt(tf2.getText());
            int ans = num1 > num2 ? num1 : num2;
            result=result.valueOf(ans);
            result=("Sum of two number:="+result);
        }
        catch(Exception E)
        {
            result="Error";
        }
        repaint();
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.drawString(result,120,120);
    }
}

```

Now, on the basis of the example given in the above Java application program named JPS2.java, the working of Java virtual machine will be traced. The Java virtual machine (simple virtual machine) can be categorized into five fundamental pieces as shown in figure 24.2.

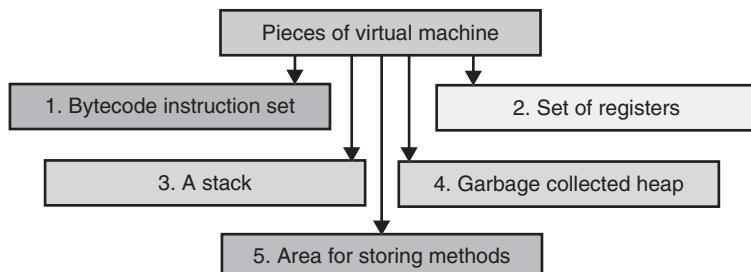


Figure 24.2 Five fundamental pieces for Java virtual machine

Some of these pieces might be implemented by using an interpreter ‘java’ or native binary code compiler—Java native interface (JNI) or by a hardware chip.

The remarkable point here is that the memory areas required by the Java virtual machine are not just required at any particular place in the memory (primary or secondary), or to be in any fixed order. However, all but the method area must be able to represent align 32-bit values (Java stack is 32-bit wide).

Now, the pieces of Java virtual machine can be shown in Figure 24.2.

Java Bytecodes

To get the bytecode for any application implemented in Java, the Java application will first be compiled using ‘javac’. For getting the concept of byte code compile the above file (JPS2.java) using javac.

After compiling the Java program using ‘javac’ the ‘.class’ file format is obtained which contains the bytecodes which makes the program platform independent. To see the bytecodes the javap will be used with option ‘-c’.

```
javap -c filename>filename.bc
```

Note: Here extension .bc is used for bytecode.

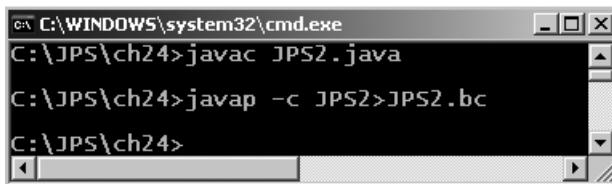


Figure 24.3 javap used to get bytecode for JPS2.java

A Java virtual machine starts execution by invoking the method `main` of some specified class, passing it a single argument, which is an array of strings (`public static void main(String args[])`). This causes the specified class to be loaded, linked to other types that it uses and initialized. The method `main` must be declared **public**, **static**, and **void**. Below the file ‘**JPS2.class**’ is given that contains the bytecode as given below:

```
/*FILE:"JPS2.BC" CONTAINS BYTECODE FOR THE FILE "JPS2.CLASS" */

Compiled from "JPS2.java"
public class JPS2 extends java.applet.Applet implements
    java.awt.event.ActionListener{
    java.awt.Button but1;
    java.lang.String result;
    boolean in;
    int num1;
    int num2;
    java.awt.TextField tf1;
    java.awt.TextField tf2;
    java.awt.Label L1;
    java.awt.Label L2;
    public JPS2();
```

```

Code:
 0:  aload_0
 1:  invokespecial    #1;  //Method java/applet/
                           Applet."<init>":()V
 4:  aload_0
 5:  ldc      #2;  //String
 7:  putfield   #3;  //Field result:Ljava/lang/String;
10:   aload_0
11:   iconst_0
12:   putfield   #4;  //Field in:Z
15:   return
public void init();
Code:
 0:  aload_0
 1:  getstatic   #5;  //Field java.awt.Color.blue:Ljava/
                      awt/Color;
 4:  invokevirtual #6;  //Method setBackground:(Ljava/
                      awt/Color;)V
 7:  aload_0
 8:  new      #7;  //class java.awt.Label
11:  dup
12:  ldc      #8;  //String Enter first number:
14:  invokespecial #9;  //Method java/
                      awt/Label."<init>":(Ljava/lang/String;)V
17:  putfield   #10; //Field L1:Ljava.awt.Label;
20:  aload_0
21:  aload_0
22:  getfield   #10; //Field L1:Ljava.awt.Label;
25:  invokevirtual #11; //Method add:(Ljava/
                      awt/Component;)Ljava.awt.Component;
28:  pop
29:  aload_0
30:  new      #12;  //class java.awt.TextField
33:  dup
34:  bipush 10
36:  invokespecial #13; //Method java/
                      awt/TextField."<init>":(I)V
39:  putfield   #14; //Field tf1: Ljava.awt/
                      TextField;
42:  aload_0
43:  aload_0
44:  getfield   #14; //Field tf1:Ljava.awt/
                      TextField;
47:  invokevirtual #11; //Method add:(Ljava/
                      awt/Component;)Ljava.awt.Component;
50:  pop
51:  aload_0

```

```
52:      new      #7; //class java.awt.Label
55:      dup
56:      ldc      #15; //String Enter second number:
58:      invokespecial #9; //Method java
           /awt/Label."<init>":(Ljava/lang/String;)V
61:      putfield    #16; //Field L2:Ljava.awt.Label;
64:      aload_0
65:      aload_0
66:      getfield    #16; //Field L2:Ljava.awt.Label;
69:      invokevirtual #11; //Method add:(Ljava
           /awt/Component;)Ljava.awt.Component;
72:      pop
73:      aload_0
74:      new      #12; //class java.awt.TextField
77:      dup
78:      bipush 10
80:      invokespecial #13; //Method java
           /awt/TextField."<init>":(I)V
83:      putfield    #17; //Field tf2:Ljava.awt/
           TextField;
86:      aload_0
87:      aload_0
88:      getfield    #17; //Field tf2:Ljava.awt/
           TextField;
91:      invokevirtual #11; //Method add:(Ljava
           /awt/Component;)Ljava.awt.Component;
94:      pop
95:      aload_0
96:      new      #18; //class java.awt.Button
99:      dup
100:     ldc      #19; //String Click for maximum of two
           no.
102:     invokespecial #20; //Method java
           /awt/Button."<init>":(Ljava/lang/String;)V
105:     putfield    #21; //Field but1:Ljava.awt/
           Button;
108:     aload_0
109:     aload_0
110:     getfield    #21; //Field but1:Ljava.awt/
           Button;
113:     invokevirtual #11; //Method add:(Ljava
           /awt/Component;)Ljava.awt.Component;
116:     pop
117:     aload_0
118:     getfield    #21; //Field but1:Ljava.awt/
           Button;
121:     aload_0
```

```
122:     invokevirtual #22; //Method java.awt
           /Button.addActionListener:(Ljava.awt.event/
           ActionListener;)V
125:     return
public void actionPerformed(java.awt.event.ActionEvent);
Code:
 0:  aload_0
 1:  aload_0
 2:  getfield    #14;  //Field tf1:Ljava.awt/TextField;
 5:  invokevirtual #23; //Method java
           /awt/TextField.getText:()Ljava.lang/String;
 8:  invokestatic #24; //Method java/lang
           /Integer.parseInt:(Ljava.lang/String;)I
11:   putfield    #25; //Field num1: I
14:   aload_0
15:   aload_0
16:   getfield    #17; //Field tf2:Ljava.awt/
           TextField;
19:   invokevirtual #23; //Method java
           /awt/TextField.getText:()Ljava.lang/String;
22:   invokestatic #24; //Method java
           /lang/Integer.parseInt:(Ljava.lang/String;)I
25:   putfield    #26; //Field num2:I
28:   aload_0
29:   getfield    #25; //Field num1:I
32:   aload_0
33:   getfield    #26; //Field num2:I
36:   if_icmpne 46
39:   aload_0
40:   getfield    #25; //Field num1:I
43:   goto 50
46:   aload_0
47:   getfield    #26; //Field num2:I
50:   istore_2
51:   aload_0
52:   aload_0
53:   getfield    #3; //Field result:Ljava.lang/
           String;
56:   pop
57:   iload_2
58:   invokestatic #27; //Method java
           /lang/String.valueOf:(I)Ljava.lang/String;
61:   putfield    #3; //Field result:Ljava.lang/
           String;
64:   aload_0
65:   new      #28; //class java.lang.StringBuilder
68:   dup
```

```

69:      invokespecial    #29;  //Method java
          /lang/StringBuilder."<init>":()V
72:      ldc      #30;  //String Sum of two number:=
74:      invokevirtual    #31;  //Method java/lang/
          StringBuilder.append:(Ljava/lang/String;)V
          Ljava/lang/StringBuilder;
77:      aload_0
78:      getfield     #3;   //Field result:Ljava/lang/
          String;
81:      invokevirtual    #31;  //Method java/lang/
          StringBuilder.append:(Ljava/lang/String;)V
          Ljava/lang/StringBuilder;
84:      invokevirtual    #32;  //Method java/lang/
          StringBuilder.toString:()Ljava/lang/String;
87:      putfield     #3;   //Field result:Ljava/lang/
          String;
90:      goto 100
93:      astore_2
94:      aload_0
95:      ldc      #34;  //String Error
97:      putfield     #3;   //Field result:Ljava/lang/
          String;
100:     aload_0
101:     invokevirtual    #35;  //Method repaint:()V
104:     return
Exception table: from to target type
  0   90   93   Class java/lang/Exception
public void paint(java.awt.Graphics);
Code:
  0:  aload_1
  1:  getstatic     #5;   //Field java.awt.Color.
          blue:Ljava.awt/      Color;
  4:  invokevirtual    #36;  //Method java/
          awt/Graphics.setColor:(Ljava.awt/Color;)V
  7:  aload_1
  8:  aload_0
  9:  getfield     #3;   //Field result: Ljava/lang/String;
12:  bipush       120
14:  bipush       120
16:  invokevirtual    #37;  //Method java/
          awt/Graphics.drawString:(Ljava/lang/String;II)V
V
19:  return
}

```

The bytecodes are a high-level representation of the program so that optimization and machine code generation (via a **just-in-time compiler**) can be performed at that level. In addition, garbage collection can occur inside the virtual machine, since it holds variables in stacks in the Java platform address space.

A bytecode instruction consists of a one-byte opcode that serves to identify the instruction involved and zero or more operands, each of which may be more than one byte long, that encodes the parameters the opcode requires.

Set of Registers

For the execution of the byte code, Java virtual machine use a set registers which is very similar to the registers given inside a real computer system. Java virtual machine use registers for performing three major tasks: keeping the machine's state, affecting the operation of virtual machine and updating after each bytes-code is executed.

There are mainly four types of registers given for Java virtual machine. These registers are basically 32-bit wide.

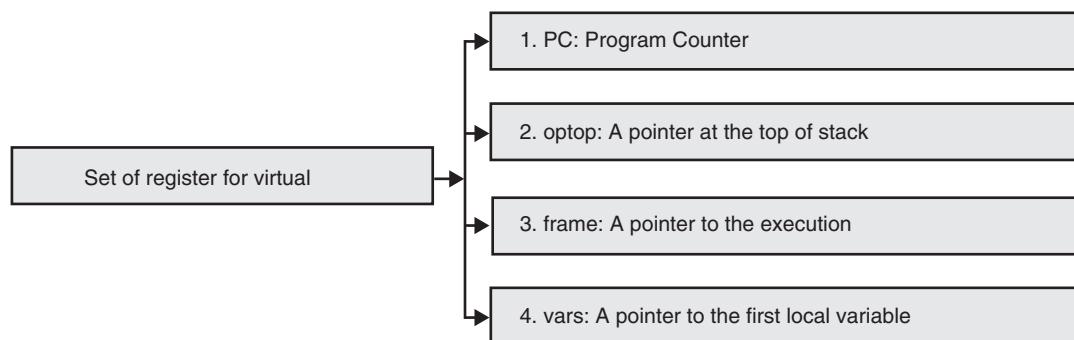


Figure 24.4 Set of register for Java virtual machine

Since Java virtual machine is completely stack based, it does not use any registers (pc, otop, frame and vars) for passing or receiving any type of arguments.

The Stack

Stack plays an important role in the working of Java virtual machine. A Java stack frame is very similar to the stack frame used for a conventional programming language—it holds the state for a single method call. The role of stack in Java virtual machine is to supply parameters to bytecodes and methods, and to receive result back from them.

To understand what has been given for the stack, the above example can be considered once again. In the program JPS2.java method `public void paint(Graphics g)` is given. This method accepts object of `Graphics` class.

```

public void paint(Graphics g)
{
    g.setColor(Color.blue);
    g.drawString(result,120,120);
}
  
```

The byte code for the above method is:

```

public void paint(java.awt.Graphics);
Code:
0:  aload_1
1:  getstatic   #5;  //Field java/awt/Color.blue:Ljava/awt/
                   Color;
4:  invokevirtual #36; //Method java/awt/Graphics.
                   setColor:(Ljava/awt/Color;)V
  
```

```

7:  aload_1
8:  aload_0
9:  getfield   #3;  //Field result: Ljava/lang/String;
12:    bipush     120
14:    bipush     120
16:    invokevirtual #37; //Method java/
                      awt/Graphics.drawString:(Ljava/lang/String;II)V
19:    return
}

```

Explanation: In the bytecode given above for paint method, `aload_1` loads object reference from the local variable. Local variable in the current Java frame must contain a return address or reference to an object (Object `g` of class `Graphics`). Line-2 shows a `getstatic` field from class. The general form for `getstatic` is:

```
getstatic ...., =>..., value-word1, value-word2
```

It is used to construct an index into the constant pool of the current class and the constant pool item will be a field reference to a static field of a class.

```
1:  getstatic #5; //Field java.awt.Color.blue:Ljava/awt/Color;
```

In the above `getstatic` field construct an index into the constant pool for the `Color` class. The constant pool item with nothing but the `Color.blue` will be a field reference to a static field of a class `Color`. The value of that field (`Color.blue`) is placed on the top of the stack. The first stack picture is for 32-bit wide and the second will be for 64-bit. The noticeable point here is if the specified field is not a static field, an **IncompatibleClassChangeError** is thrown by the system.

Now, the next line of the bytecode is for `invokevirtual`, which is used to invoke an instance method based on runtime type. The general code for `invokevirtual` is:

```
invokevirtual ...., handle, [arg1][arg2...].=>.....
```

In the program `JPS2.java`, `invokevirtual` has been used to invoke the instance method for `paint` method.

```
4:  invokevirtual #36; //Method java.awt.Graphics.setColor:(Ljava/
                      awt/Color;)V
```

The operand stack must maintain a **reference/handle** to an object (object `g` for `Graphics` class) and some number of arguments. It use an index into the constant pool of the current class. Item at some particular index in the constant pool contains the complete method signature. The reason why the constant pool contains the method signature is that, `invokevirtual` manages a pointer to the object which has been used to invoke the method table is to be retrieved from the object reference. The method signature is then verified from the method table. Method signature is guaranteed to get an exact match for one of the method signatures in the table. If value of handle is null, it throws a `NullPointerException`. And if during method invocation a stack overflow is found, a `StackOverflowError` is thrown. The role of `getfield` is to fetch field from object. There is a handle which must be a reference to an object. `bipush` pushes one byte signed integer, as shown in the above bytecode at line-12 and line-14.

```

12:  bipush     120
14:  bipush     120

```

The value then expanded to an `int` and pushed onto the operand stack. The method `return` has used to return from the method. That is after getting return Java interpreter returns control to its caller. The value is pushed onto the operand stack of the previous execution environment.

The Garbage Collected Heap

The technique through which a program can obtain space in the RAM during the execution of the program and not during compilation is called dynamic memory allocation. In this method, space for the variables (array, strings, etc.) is allocated from a free memory region which is known as heap. This area is not fixed in nature and keeps changing as the memory is allocated and de-allocated from this region. The entire runtime view of memory for a program is shown in Figure 24.5.

The heap is often assigned a large, fixed sized when the Java runtime system is started, but on systems that support virtual memory, it can grow according to the requirement, in an unbounded manner. Since it is well known that Java supports the concepts of garbage collection to collect the object automatically there is no need de-allocate the memory manually from programmers side.

Area for Storing Methods

Java virtual machine supports basically two different areas for storing purposes. The first one is called as Method area and second Constant pool.

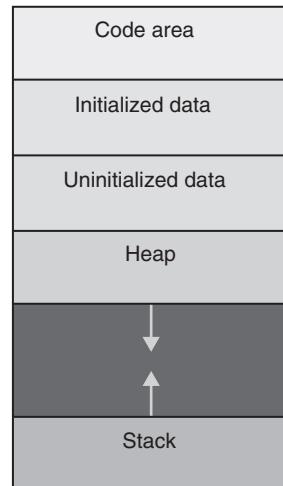


Figure 24.5 Runtime memory area of a Java application program

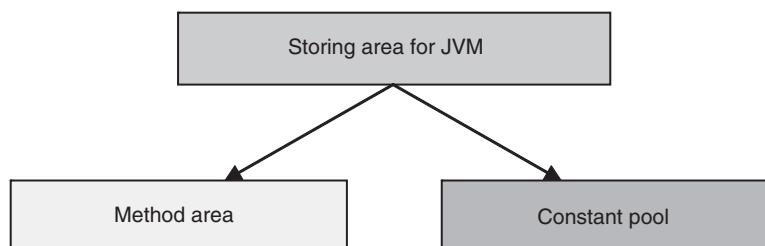


Figure 24.6 Area used for storing by Java Virtual Machine

- *Method area:* It is very similar to the compiled code area of conventional programming language environment. It keeps the bytecodes used to implement almost all the methods used for Java application development by Java system.
- *Constant pool:* Constant pool is basically given for classes used in the Java system. Since, it is well known that heap is a part of memory from which newly created instance are allocated, each class in the heap has a constant pool attached to it. Constant pool is usually created by javac at the compile time. These constants encode all the names like variables and methods used by any method in a class.

24.3 VIRTUAL MACHINE ERRORS

When any Java program is compiled and interpreted, most of the time problem is faced in loading, linking and initialization of the program ('.bc' or '.class') with virtual machine. A Java virtual machine throws an object that is an instance of a subclass of the class `VirtualMachineError` when an internal error or resource limitation prevents it from implementing the semantics of the Java language. The Java virtual machine specification cannot predict where resource limitations or internal errors may be encountered.

and does not mandate precisely when they can be reported. Thus, any of the virtual machine errors listed as subclasses of `VirtualMachineError` may be thrown at any time during the operation of the Java virtual machine.

24.4 INSTRUCTION SET FOR VIRTUAL MACHINE

Java virtual machine instructions are represented in this chapter by entries of the form shown in Table 24.1.

| Operation | Description | | | | |
|--------------------|--|----------|----------|----------|-------|
| Format | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>mnemonic</td></tr> <tr><td>Operand1</td></tr> <tr><td>Operand2</td></tr> <tr><td>-----</td></tr> </table> | mnemonic | Operand1 | Operand2 | ----- |
| mnemonic | | | | | |
| Operand1 | | | | | |
| Operand2 | | | | | |
| ----- | | | | | |
| Forms | <code>mnemonic = opcode</code> | | | | |
| Stack | <code>....., value1, value2 =></code>
<code>....., vlaue3</code> | | | | |
| Description | A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the result, and so on. | | | | |
| Linking Exceptions | If any linking exception is thrown by the execution of this instruction, they are set off one to a line in a way that they must be thrown. | | | | |
| Runtime Exceptions | If any runtime exceptions can be thrown by the execution of an instruction, they are set off one to a line in a way that they must be thrown.
Other than the linking and runtime exceptions, if any, listed for an instruction, the instruction must not throw any runtime exceptions except for instances of <code>VirtualMachineError</code> or its subclasses. | | | | |
| Notes | Comments not strictly part of the specification of an instruction are set aside as note at the end of the description. | | | | |

Table 24.1 An example of instruction page

24.5 INVOKING METHODS USING VIRTUAL MACHINE

The normal method invocation for a Java instance method dispatches on the runtime type of the object (they are virtual, in C++ terms). Such an invocation is implemented using the `invokevirtual` instruction, which takes as its argument an index to a constant pool entry giving the fully qualified name of the class type of the object, the name of the method to invoke, and that method's descriptor. To invoke the `area_rectangle` method, defined as an instance method, the code can be written as shown below:

```
/*PROG 24.2 INVOKING METHODS DEFINED AS AN INSTANCE METHOD
USING VIRTUAL MACHINE */
```

```
import java.io.*;
import java.util.*;
```

```

class JPS3
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int area_rect;
        System.out.println("\nEnter a number");
        //num = sc.nextDouble();
        area_rect = area15n10();
        System.out.println("\nSquare of "+area_rect);
    }
    static int area15n10()
    {
        return area_rectangle(150, 100);
    }
    static int area_rectangle(int l, int w)
    {
        return (l * w);
    }
}

```

Now, the complete bytecode is given for the program JPS3.java.

```

/*FILE:"JPS3.BC" CONTAINS BYTECODE FOR THE FILE "JPS3.CLASS" */

Compiled from "JPS3.java"
class JPS3 extends java.lang.Object{
JPS3();
Code:
  0: aload_0
  1: invokespecial    #1;  //Method java/lang/
   Object."<init>":()V
  4: return
public static void main(java.lang.String[]);
Code:
  0: new      #2;  //class java/util/Scanner
  3: dup
  4: getstatic     #3;  //Field java/lang/System.in:Ljava/
   io/InputStream;
  7: invokespecial    #4;//Method java/util/
   Scanner."<init>":(Ljava/io/InputStream;)V
 10:   astore_1
 11:   getstatic     #5;  //Field
   java/lang/System.out:Ljava/io/PrintStream;
 14:   ldc      #6;  //String \nEnter a number
 16:   invokevirtual    #7;  //Method
   java/io/PrintStream.println:(Ljava/lang/String;)V

```

```

19:      invokestatic      #8;  //Method area15n10:()I
22:      istore_2
23:      getstatic       #5;  //Field
        java/lang/System.out:Ljava/io/PrintStream;
26:      new      #9;  //class java/lang/StringBuilder
29:      dup
30:      invokespecial    #10; //Method
        java/lang/StringBuilder."<init>":() V
33:      ldc      #11; //String \nSquare of
35:      invokevirtual    #12; //Method java/lang/
        StringBuilder.append:(Ljava/lang/String;)Ljava/
        lang/StringBuilder;
38:      iload_2
39:      invokevirtual    #13; //Method java/lang/
        StringBuilder.append:(I)Ljava/lang/
        StringBuilder;
42:      invokevirtual    #14; //Method java/lang/
        StringBuilder.toString():()Ljava/lang/String;
45:      invokevirtual    #7;  //Method java/io/
        PrintStream.println:(Ljava/lang/String;)V
48:      return
static int area15n10();
Code:
 0: sipush 150
 3: bipush 100
 5: invokestatic      #15;  //Method area_rectangle:(II)I
 8: ireturn
static int area_rectangle(int, int);
Code:
 0: iload_0
 1: iload_1
 2: imul
 3: ireturn
}

```

Explanation: To know the method invocation, the detail of bytecode is first looked into for the method which is shown below:

```

static int area150n100();
Code:
 0: sipush 150
 3: bipush 100
 5: invokestatic      #15;  //Method area_rectangle:(II)I
 8: ireturn

```

sipush is given as `sipush 100` pushes two byte signed integer where byte 1 and byte 2 are combined into a signed 16-bit value. Later on, this value is expanded to a certain extent and pushed onto the stack. `bipush` is also given to push the data element but it pushes one byte signed integer. This value is expanded to an int and pushed onto the operand stack. The general signature for `sipush` and `bipush` is:

```
sipush ..... ==> ....., value
bipush ..... ==> ....., value
```

The next statement of the bytecode for the area15n10() is:

```
5: invokestatic #15; //Method area_rectangle:(II)I
```

The byte1 and byte2 are available in the operand stack. These numbers work as arguments. Both the byte1 and byte2 are used to construct and index into the constant pool of the current class. The item which is in the constant pool at a certain index keeps the signature for the method and class. The general form for the invokestatic is:

```
invokestatic ..... , [a1, [a2,...]], .....=> .....
```

First method invocation starts by pushing the reference of the current instance onto the stack. Then the values are pushed into the stack using `sipush` and `bipush` statement. When the frame for the `area_rectangle` method is created the arguments passed to the method `area_rectangle` (150, 100) becomes the initial values of the new frame's local variables. Finally, `area_rectangle` is invoked using `invokestatic`. When `area_rectangle` returns the value, its int value is then pushed onto the stack. The return value is thus put in place to be immediately returned to the invoker of `int area150n10()`.

The last statement of the bytecode for the method `int area150n10()` is `ireturn` instruction. The `ireturn` is used for the value to be pushed onto the stack of the previous execution environment. The general form for `ireturn` is given here:

```
ireturn ..... , value => [empty]
```

The Java virtual machine provides distinct return instructions for many of its numeric and reference data types, as well as a `return` instruction for methods with no return value. The same set of return instructions is used for all varieties of method invocations. The various types of return instruction with their general form used in bytecode to return from method are listed below:

- (a) `ireturn` (General form: `ireturn , value => [empty]`)
- (b) `lreturn` (General form: `lreturn , value-word1, value-word2 => [empty]`)
- (c) `freturn` (General form: `freturn , value => [empty]`)
- (d) `dreturn` (General form: `dreturn , value-word1, value-word2 => [empty]`)
- (e) `areturn` (General form: `areturn , value => [empty]`)

24.6 CLASS INSTANCE AND VIRTUAL MACHINE

Java virtual machine class instances are created using the Java virtual machine's `new` instruction. Once the class instance and its instance variables have been created, and those of the class and all of its **superclasses** have been initialized to their default values, an instance initialization method of the new class instance (`<init>`) is invoked. (Recall that at the level of the Java virtual machine, a constructor appears as a method with the special compiler-supplied name `<init>`. This special method is known as the instance initialization method. Multiple instance initialization methods, corresponding to multiple constructors, may exist for a given class.) For example:

```
/*PROG 24.3 DEMO OF CLASS INSTANCE AND VIRTUAL MACHINE VER 1*/
class demo_class_vm
{
    private int cx, cy;
```

```

void input(int x, int y)
{
    cx = x;
    cy = y;
}
void show()
{
    System.out.println("\ncx=" + cx);
    System.out.println("ncy=" + cy);
}
class JPS5
{
    public static void main(String args[])
    {
        demo_class_vm d = new demo_class_vm();
        d.input(100, 200);
        d.show();
    }
}
OUTPUT:
C:\JPS\ch24>javac JPS5.java
C:\JPS\ch24>java JPS5
cx=100
cy=200
C:\JPS\ch24>
```

```

/* FILE: JPS5.bc */

Compiled from "JPS5.java"
class JPS5 extends java.lang.Object{
JPS5();
Code:
  0: aload_0
  1: invokespecial    #1; //Method java/lang/
          Object."<init>":()V
4: return
public static void main(java.lang.String[]);
Code:
  0: new      #2; //class demo_class_vm
  3: dup
  4: invokespecial    #3; //Method demo_class_
          vm."<init>":()V
  7: astore_1
  8: aload_1
```

```

9: bipush      100
11:    sipush      200
14:    invokevirtual #4; //Method demo_class_
           vm.input:(II)V
17:    aload_1
18:    invokevirtual #5; //Method demo_class_
           vm.show:()V
21:    return
}

```

From the byte code given above, it can be seen that the Java class `demo_class_vm` starts with the new instruction which is basically created by Java virtual machine.

```

0: new      #2; //class demo_class_vm
3: dup
4: invokespecial #3; //Method demo_class_vm."<init>":()V
7: astore_1
8: aload_1
9: bipush 100
11:    sipush 200
14:    invokevirtual #4; //Method demo_class_
           vm.input:(II)V
17:    aload_1
18:    invokevirtual #5; //Method demo_class_vm.show:()
V
21:    return
}

```

Class instances are passed and returned (as `reference` types) very much like numeric values, although type `reference` has its own complement of instructions:

```

/* PROG 24.4 CLASS INSTANCE AND VIRTUAL MACHINE VER 2 */

class JPS7          //Class declaration JPS7
{
    int i;        // An instance variable
    JPS7 example()
    {
        JPS7 o = new JPS7();
        return silly(o);
    }
    JPS7 silly(JPS7 o)
    {
        if (o != null)
        {
            return o;
        }
    }
}

```

```

        else
        {
            return o;
        }
    }
}

```

Becomes like the below given code:

```

JPS7 example();
Code:
 0:   new      #2; //class JPS7
 3:   dup
 4:   invokespecial #3; //Method "<init>":()V
 7:   astore_1
 8:   aload_0
 9:   aload_1
10:   invokevirtual #4; //Method silly:(LJPS7;)
           LJPS7;
13:   areturn

```

The method given in the above program `JPS7.java: silly(JPS7 o)`

```

Method: JPS7 silly(JPS7);
Code:
 0:   aload_1
 1:   ifnull 6
 4:   aload_1
 5:   areturn
 6:   aload_1
 7:   areturn

```

The fields of a class instance (instance variables) are accessed using the `getfield` and `putfield` instructions. If `i` is an instance variable of type `int`, the methods `setIt` and `getIt` are defined as:

```

void setIt(int value)
{
    i = value;
}
int getIt()
{
    return i;
}

```

Method void setIt(int)

0 aload_0

```

1 iload_1
2 putfield    #4 // Field Example.i I
5 return

Method int getIt()

0 aload_0
1 getfield    #4 // Field Example.i I
4 ireturn

```

As with the operands of method invocation instructions, the operands of the `putfield` and `getfield` instructions (the constant pool index #4) are not the offsets of the fields in the class instance. The Java compiler generates symbolic references to the fields of an instance, which are stored in the constant pool. Those constant pool items are resolved at runtime to determine the actual field offset.

Basically `getfield` and `putfield` are used for manipulating object fields. The `getfield` fetches field object. The constant pool item will be a field reference to a class name and a field name. The general form for `getfield` is given as:

```

getfield .......,handle => ......., value
getfield .......,handle => .......,value-word1, value-word2

```

Here, handle must be reference to an object. The value at offset into the object referenced by handle replaces handle on the top of the stack. If the specified field is a static field, an `IncompatibleClassChangeError` is thrown.

The `putfield` is also used for manipulating object fields. This instruction sets field in object. The constant pool item is a field reference to a class name and a field name. The general form for `putfield` is:

```

putfield ......., handle, value => .......
Putfield ......., handle, value-word1, value-word2 => .......

```

If handle is null, a `NullPointerException` is thrown and if the specified field is a static field, an `IncompatibleClassChangeError` is thrown.

24.7 ARRAY AND VIRTUAL MACHINE

Arrays are also served as an object for Java virtual machine. To create an array of a numeric type the `newarray` instruction is used in the bytecode.

```

/*PROG 24.5 DEMO ARRAYS AND VIRTUAL MACHINE USING NEWARRAY */

class JPS5
{
    public static void main(String args[])
    {
        int[] arr ={ 0, 1, 2, 3, 4 };
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}

```

The above code might be compiled to

```
/* FILE: JPS5.BC FOR DEMO ARRAYS AND VIRTUAL MACHINE USING NEW  
ARRAY */
```

```
Compiled from "JPS5.java"  
class JPS5 extends java.lang.Object  
{  
JPS5();  
    Code:  
        0: aload_0  
        1: invokespecial    #1; //Method java/lang/  
           Object."<init>":()V  
        4: return  
  
public static void main(java.lang.String[]);  
    Code:  
        0: iconst_5  
        1: newarray int  
        3: dup  
        4: iconst_0  
        5: iconst_0  
        6: iastore  
        7: dup  
        8: iconst_1  
        9: iconst_1  
       10:     iastore  
       11:     dup  
       12:     iconst_2  
       13:     iconst_2  
       14:     iastore  
       15:     dup  
       16:     iconst_3  
       17:     iconst_3  
       18:     iastore  
       19:     dup  
       20:     iconst_4  
       21:     iconst_4  
       22:     iastore  
       23:     astore_1  
       24:     iconst_0  
       25:     istore_2  
       26:     iload_2  
       27:     aload_1  
       28:     arraylength  
       29:     if_icmpge 65  
       32:     getstatic      #2; //Field java/lang/System.out:  
                      Ljava/io/PrintStream;  
       35:     new         #3; //class java/lang/StringBuilder
```

```

38:      dup
39:      invokespecial    #4; //Method java/lang/
           StringBuilder." <init>":()V
42:      aload_1
43:      iload_2
44:      iaload
45:      invokevirtual   #5; //Method java/lang/
           StringBuilder.append:(I)Ljava/lang/StringBuilder;
48:      ldc     #6; //String
50:      invokevirtual   #7; //Method java/lang/
           StringBuilder.append:(Ljava/lang/String;)Ljava/
           lang/StringBuilder;
53:      invokevirtual   #8; //Method java/lang/
           StringBuilder.toString:()Ljava/lang/String;
56:      invokevirtual   #9; //Method java/io/PrintStream
           .print:(Ljava/lang/String;)V
59:      iinc 2, 1
62:      goto 26
65:      return
}

```

Explanation: The above bytecode is given for the Java application program where array has been used. At line 0 of `public static void main(java.lang.String[]);` `iconst_5` is given which is a pushing instruction, used to push the five integer values onto the stack. There are six of these bytecodes one for each of the integers 0–5 (`iconst_0`, `iconst_1`, `iconst_2`, `iconst_2`, `iconst_3`, `iconst_4` and `iconst_5`) given by line 5, 12, 13, 16, 17, 20 and 21. The general form `iconst` is given below:

`iconst_....., =>`

The next instruction given in the above bytecode is nothing but the instruction used for creating an array of numeric type. The general form of this instruction is:

`newarray , size => result`

The `newarray` instruction is given to manage the array. It is used to allocate new array. Its size must be an int. It represents the number of elements in the new array. The `newarray` instruction is an attempt to allocate a new array of the indicated type, which is capable of holding `size` elements. If `size` is less than zero, a `NegativeArraySizeException` is thrown and if there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown. The instruction `dup` is given at line-3 used for stack operation. The `dup` is used to duplicate the top word on the stack. The general form for `dup` instruction is:

`dup, any =>, any, any`

Now the next instruction given in the above bytecode for JPS5 will be discussed, which is `iastore` given at line-6. The general form of `iastore` is given below:

`iastore, array, index, value =>`

The `iastore` stores the value into the array. Array must be an array of int, index must be an integer. If array is null, a `NullPointerException` is thrown and if index is not within the bounds of array, an `ArrayIndexOutOfBoundsException` is thrown.

The next instruction discussed here is `astore`, given at line-23 in the bytecode shown above.

`23: astore_1`

The `astore` stores the object reference into the local variable. It uses a handle. Handle must be a return address or a reference to an object. The general form for the `astore` instruction is given as:

```
astore_<A> ..... , handle => .....
```

There are four types of these bytecodes, one for each of the integers starting from 0 to 3 as:

```
astore_0, astore_1, astore_2 and astore_3.
```

Now, the working of `istore` given at line-25 will be discussed. The `istore` is used to store stack values into local variables.

```
25: istore_2
```

In the instruction `istore` 'i' is given for integer. That is the reason `istore` instruction is used to store int into the local variable. The general signature for the instruction is given as:

```
istore ..... , value => .....
```

The next new instruction given in the bytecode is `iload` given at line-26

```
26: iload_2
```

The `iload` instruction is nothing but the instruction given for the Java virtual machine to load the local variables onto the stack. The signature for `iload` instruction is:

```
iload_ ..... => ..... , value
```

The role of `iload` is to load int from local variable onto the operand stack. The local variable (`<i>load_2` shown in the byte code) in the current Java frame must contain an int. There are four different flavours of `iload` instruction, one for each of the integer starting from 0 to 3: `iload_0`, `iload_1`, `iload_2` and `iload_3`.

Now a bit about the `aload` instruction will be discussed. The `aload` instruction is shown in the above bytecode at line-27 as:

```
27: aload_1
```

There are mainly two types of `aload` instructions. The signature of both the `aload` instructions is given below:

```
aload ..... => ..... , value
```

It is used to load object reference from local variable.

```
aload_<A> ..... => ..... , value
```

It is used to load object reference from local variable. Here local variable `<A>` in the current Java frame must contain a return address or reference to an object. There are four different bytecodes for `aload_<A>`, one for each of the integer 0–3: `aload_0`, `aload_1`, `aload_2` and `aload_3`.

The next instruction given in the bytecode file `JPS5.bc` is `if_icmpge` which is nothing but used for transferring control. The instruction `if_icmpge` is given at line-29 as shown below:

```
29: if_icmpge 65
```

For the description of `if_icmpge` instruction, the signature of the instruction is looked into which is given as:

```
if_icmpge ..... , value1, value2 => .....
```

Here for the instruction `if_icmpge` execution proceeds at that offset where `value<rel>0` is true in the first set of bytecodes Table 24.2, `value1<rel>value2` is true in the second set, or value is null or not null in the third; byte1 and byte2 are used to construct a signed 16-bit offset from the `pc` (register). Here `<rel>` is nothing but one of eq, ne, lt, gt, le, and ge which represents the following:

| Bytecode for <code>if_icmpge</code> | Meaning |
|-------------------------------------|-----------------------|
| <code>eq</code> | Equal |
| <code>ne</code> | Not equal |
| <code>lt</code> | Less than |
| <code>gt</code> | Greater than |
| <code>le</code> | Less than or equal |
| <code>ge</code> | Greater than or equal |

Table 24.2 Option used for `if_icmpge` instruction

Now `getstatic` instruction can be discussed which is used for manipulating object fields. The `getstatic` field is shown in the above code at line-32

```
32: getstatic      #2;  //Field java/lang/System.out: Ljava/
    io/PrintStream;
```

The instruction `getstatic` is given to get static field from class. There are two flavours of `getstatic` instruction. The general form of both the `getstatic` type is given below:

```
getstatic ..... , => ..... , value
getstatic ..... , => ..... , value-word1, value-word2
```

For `getstatic` instruction byte1 and byte2 are used to construct an index into the constant pool of the current class. The constant pool item will be field reference to a static field of a class. The value of that field is placed on the top of the operand stack. Here it is important to note that the first stack picture is for 32-bit and the second for 64-bit-wide fields.

The next instruction to be focussed is `new` instruction. In the above bytecode `new` is used at line-35 as shown below:

```
35: new      #3;  //class java/lang/StringBuilder
```

The `new` instruction is mainly used to create new object. The general form for `new` instruction is given below:

```
new ..... => ...., handle
```

Byte1 and byte2 are used to construct an index into the constant pool of the current class. The item at that index must be a class name that can be resolved to a class pointer, class.

The `iaload` instruction is the next instruction which will be discussed now. It is given in the above bytecode at line-44.

```
44: iaload
```

The signature for the `iaload` instruction is given below:

```
iaload ..... , array, index => ...., value
```

The `iaload` instruction is used to load `<type>` from array. For the array, index must be integer type. The `<type>` value at position number index in array is retrieved and pushed onto the top of the stack.

The `ldc` instruction is used to push the item from the constant pool. In the byte code `ldc` instruction has been given at line-48

```
48: ldc      #6;  //String
```

The `iinc` instruction is given to increment local variable by constant. The `iinc` instruction is given at line-59 in the above bytecode given for `JPS5.bc`

```
59: iinc 2, 1
```

Value of `iinc` instruction is incremented by the value `byte2`, where `byte2` is treated as a signed 8-bit quantity.

The last instruction of the above bytecode is `goto` instruction, which is given at line-62 as given below:

62: goto 26

There are two versions of goto statement: goto -no change and goto -no change.

The `goto` instruction is also used for transferring control. Byte1 and byte2 are used to construct a signed 16-bit/32-bit offset. For `goto` statement, execution starts at that particular offset from the register pc.

24.8 SWITCH STATEMENTS AND VIRTUAL MACHINE

There are two different types of instruction to compile switch statement in Java: `tableswitch` instruction and `lookupswitch` instruction.

To see the working of both the instructions given for handling switch statement in Java an example is given below.

```
/*PROG 24.6 DEMO (TABLESWITCH INSTRUCTION) OF SWITCH-CASE AND  
VIRTUAL MACHINE */
```

The bytecode for the above program can be generated:

```
/*FILE: JPS6.BC CONTAINS BYTECODE */

Compiled from "JPS6.java"
class JPS6 extends java.lang.Object{
JPS6();
Code:
  0: aload_0
  1: invokespecial    #1;  //Method java/lang/Object.
                           "<init>":()V
  4: return
public static void main(java.lang.String[]);
Code:
  0: new      #2;  //class java/util/Scanner
  3: dup
  4: getstatic   #3;  //Field java/lang/System.in:
                      Ljava/io/InputStream;
  7: invokespecial  #4;  //Method java/util/Scanner.
                        "<init>":(Ljava/io/InputStream;)V
 10:   astore_1
 11:   getstatic   #5;  //Field java/lang/System.out:
                      Ljava/io/PrintStream;
 14:   ldc      #6;  //String \n Enter 0, 1, or 2 :=
 16:   invokevirtual #7;  //Method java/io/
                      PrintStream. print:(Ljava/lang/String;)V
 19:   aload_1
 20:   invokevirtual #8;  //Method java/util/
                      Scanner. nextInt:()I
 23:   istore_2
 24:   iload_2
 25:   tableswitch{ //0 to 2
 0: 52;
 1: 63;
 2: 74;
 default: 85}
 52:   getstatic   #5;  //Field java/lang/System.out:
                      Ljava/io/PrintStream;
 55:   ldc      #9;  //String \n U entered zero\n
 57:   invokevirtual #10; //Method java/io/
                      PrintStream. println:(Ljava/lang/String;)V
 60:   goto 93
 63:   getstatic   #5;  //Field java/lang/System.out:
                      Ljava/io/PrintStream;
 66:   ldc      #11; //String \n U entered one\n
 68:   invokevirtual #10; //Method java/io/
                      PrintStream. println:(Ljava/lang/String;)V
 71:   goto 93
 74:   getstatic   #5;  //Field java/lang/System.out:
                      Ljava/io/PrintStream; 77:   ldc      #12; //
String \n U enterd two\n
```

```

79:      invokevirtual    #10;  //Method java/io/
     PrintStream. println:(Ljava/lang/String;)V
82:      goto   93
85:      getstatic     #5;   //Field java/lang/System.out:
     Ljava/io/PrintStream;
88:      ldc      #13;  //String \n U entered other than
0,
     1,or 2\n
90:      invokevirtual    #10;  //Method java/io/
     PrintStream.
     println:(Ljava/lang/String;)V
93:      return
}

```

Explanation: The `tableswitch` instruction is used when the cases of the `switch` can be efficiently represented as indices into a table of target offsets. The important point for `tableswitch` instruction is that it operates only on integer data. The `tableswitch` instruction is efficient in terms of space complexity. In the above bytecode `tableswitch` instruction is given at line-25 as shown below:

```

25: tableswitch{ //0 to 2
  0: 52;
  1: 63;
  2: 74;
  default: 85}

```

To understand the working of `tableswitch` instruction, the general form given for the instruction can be considered:

```
tableswitch .....index => ....
```

Only three cases are given under `switch` statement in the program. And the cases given in `switch` statement are case 0, case 1 and case 2; therefore the index value will start from 0 to 2 as shown in the bytecode. The `tableswitch` is a variable-length bytecode. Immediately after the `tableswitch` opcode, zero to three 0 bytes are inserted as a padding so that the next byte starts at an address that is a multiple of 4. After the padding, a series of signed 4-byte quantities will come into picture which is nothing but default-offset, low, high, and then high-low+1. These offsets are treated as a 0-based jump table. The important thing behind the index is, if the index is less than low or if it is greater than high, default-offset is added to the pc (register).

In the above section, a lot has been discussed for `tableswitch` instruction. Now, `lookupswitch` instruction will be discussed which is also used for `switch` statement. It is a variable-length bytecode. To understand the working of `lookupswitch` the example `JPS7.java` is given below:

```
/*PROG 24.7 DEMO (LOOKUPSWITCH) OF SWITCH-CASE AND VIRTUAL
MACHINE */
```

```

import java.io.*;
import java.util.*;
class JPS7
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);

```

```

        System.out.print("\n Enter 0, 1, or 2 :=");
        int num = sc.nextInt();
        switch (num)
        {
            case -5:
                System.out.println("\n U entered minus
                                   five\n");
                break;
            case 0:
                System.out.println("\n U entered
                                   zero\n");
                break;
            case 5:
                System.out.println("\n U entered
                                   five\n");
                break;
            default:
                System.out.println("\n U entered other
                                   than -5,0,or 5\n");
        }
    }
}

```

Now, the program will be compiled to get the .class file (JPS7.class) which contains the byte code. Also javap will be used to get the bytecode for the same file.

The bytecode file for JPS7.java is shown below:

```

/*FILE: JPS7.BC CONTAINS BYTECODE */

Compiled from "JPS7.java"
class JPS7 extends java.lang.Object{
JPS7();

Code:
 0:  aload_0
 1:  invokespecial #1;  //Method java/lang/Object.
                      "<init>":()V
 4:  return
public static void main(java.lang.String[]);

Code:
 0:  new      #2;  //class java/util/Scanner
 3:  dup
 4:  getstatic   #3;  //Field java/lang/System.in:
                      Ljava/io/InputStream;
 7:  invokespecial #4;  //Method java/util/Scanner.
                      "<init>":(Ljava/io/InputStream;)V
10:   astore_1
11:   getstatic   #5;  //Field java/lang/System.out
                      Ljava/io/PrintStream;
14:   ldc      #6;  //String \n Enter 0, 1, or 2 :=

```

```

16:      invokevirtual    #7;  //Method java/io/
           PrintStream. print:(Ljava/lang/String;)V
19:      aload_1
20:      invokevirtual    #8;  //Method java/util/
           Scanner. nextInt:()I
23:      istore_2
24:      iload_2
25:      lookupswitch{ //3
       -5: 60;
       0: 71;
       5: 82;
     default: 93 }
60:      getstatic      #5;  //Field java/lang/System.out
                           :Ljava/io/PrintStream;
63:      ldc      #9;  //String \n U entered minus five\n
65:      invokevirtual    #10; //Method java/io/
           PrintStream. println:(Ljava/lang/String;)V
68:      goto 101
71:      getstatic      #5;  //Field java/lang/System.
           out:Ljava/io/PrintStream;
74:      ldc      #11; //String \n U entered zero\n
76:      invokevirtual    #10; //Method java/io/
           PrintStream. println:(Ljava/lang/String;)V
79:      goto 101
82:      getstatic      #5;  //Field java/lang/System.out
                           :Ljava/io/PrintStream;
85:      ldc      #12; //String \n U enterd five\n
87:      invokevirtual    #10; //Method java/io/
           PrintStream. println:(Ljava/lang/String;)V
90:      goto 101
93:      getstatic      #5;  //Field java/lang/System.out
                           :Ljava/io/PrintStream;
96:      ldc      #13; //String \n U entered other than -
      5,0,or 5\n
98:      invokevirtual    #10; //Method java/io/
           PrintStream .println:(Ljava/lang/String;)V
101:     return
}

```

Explanation: The `lookupswitch` instruction is used when the cases of the `switch` statement is not efficiently represented as indices into a table of target offsets. The important point for `lookupswitch` instruction is that it also operates only on integer data like `tableswitch` opcode. The bytecode for `lookupswitch` opcode is given in the above bytecode at line-25.

```

25:      lookupswitch{ //3
       -5: 60;
       0: 71;
       5: 82;
     default: 93 }

```

The general form given for `lookupswitch` opcode is shown as:

```
lookupswitch ..... , key => ....
```

If the general signature is compared with the `lookupswitch` instruction given in the bytecode, it can be said that the value of key is 3. Immediately after the `lookupswitch` opcode, zero to three 0-bytes is inserted as padding so that the next byte starts that is a multiple of 4. And immediately, after the padding is a series of pairs of signed 4-byte quantities. The first pair is special, because it contains the default offset and the number of pairs that follow. The special thing for `lookupswitch` is that the key on the operand stack must be an int. This key is basically compared with each of the matches. If it is equal to one of them, the corresponding offset is added to the pc register. And if the key does not match any of the matches, the default offset is added to the pc. Even though, the `lookupswitch` instruction must search its keys for a match rather than simply perform a bounds check and index into a table like `tableswitch`. Thus, a `tableswitch` instruction is probably more efficient than a `lookupswitch` where space considerations permit a choice.

24.9 THROWING AND HANDLING EXCEPTIONS, AND VIRTUAL MACHINE

This section reveals the inner working of exception handling mechanism that Java supports. Several exception handling mechanisms like `try`, `catch` and `throw` are discussed. Therefore to understand the working of exception handling mechanism, an example can be given here.

```
/*PROG 24.8 DEMO OF EXCEPTION HANDLING AND VIRTUAL MACHINE */
```

```
class demo1 extends Exception
{
}
class demo2 extends Exception
{
}
class demo3 extends Exception
{
}
class JPS11
{
    public static void main(String[] args)
    {
        int num = 10;
        for (num = 10; num <= 30; num += 10)
        {
            try
            {
                if (num == 20)
                    throw new demo1();
                else if (num < 20)
                    throw new demo2();
                else if (num > 20)
                    throw new demo3();
            }
            catch (Exception E)
            {

```

```
        System.out.println("Caught an  
            exception");  
    }  
}  
}  
}
```

The bytecode for the above Java application program is given below:

```
/*FILE: JPS11.BC CONTAINS BYTECODE */

Compiled from "JPS11.java"
class JPS11 extends java.lang.Object{
JPS11();
Code:
 0: aload_0
 1: invokespecial #1; //Method java/lang/
                   Object."<init>":()V
 4: return
public static void main(java.lang.String[]);
Code:
 0: bipush      10
 2: istore_1
 3: bipush      10
 5: istore_1
 6: iload_1
 7: bipush      30
 9: if_icmpgt 72
12:   iload_1
13:   bipush      20
15:   if_icmpne 26
18:   new      #2; //class demo1
21:   dup
22:   invokespecial #3; //Method demo1."<init>":()V
25:   athrow
26:   iload_1
27:   bipush      20
29:   if_icmpge 40
32:   new      #4; //class demo2
35:   dup
36:   invokespecial #5; //Method demo2."<init>":()V
39:   athrow
40:   iload_1
41:   bipush      20
43:   if_icmple 54
46:   new      #6; //class demo3
49:   dup
50:   invokespecial #7; //Method demo3."<init>":()V
53:   athrow
```

```

54:      goto 66
57:      astore_2
58:      getstatic     #9;  //Field java/lang/System.
                           out:Ljava/io/PrintStream;
61:      ldc      #10; //String Caught an exception
63:      invokevirtual   #11; //Method java/io/
                           PrintStream.println:(Ljava/lang/String;)V
66:      iinc 1,      10
69:      goto 6
72:      return

Exception table:
  from    to    target  type
    12     54      57    Class java/lang/Exception
}

```

Explanation: For digging the matter deep about the working of exception handling mechanism, there has to have the detailed ideas about the opcode given in the above bytecode file JPS11.bc. Hence, discussion will begin with `if_icmpgt` instruction given at line-9 for `public static void main(java.lang.String[]);`

```
9: if_icmpgt 72
```

The opcode `if_icmpgt` is nothing but the instruction for transfer control. The general signature for `if_icmpgt` is given as:

```
if_icmpgt ...., value1, value2 => .....
```

It is used to jump if one integer is greater than another. In the above shown bytecode first 10 is pushed onto the stack by `bipush` opcode at line-0 and stored by using `istore_1` as shown at line-1. The value 10 has been assigned in the variable `num` which is of integer type. Then again in `for` loop the value 10 in `num` is once again assigned; for this purpose once again `bipush` and `istore` instruction has been used at line-3 and line-5, respectively. After that the instruction `iload_1` is given to load the value 10 in the operand stack. Again `bipush` is given to push the value 30 onto the stack as boundary condition for the `for` loop; that is if the value will be less than or equal to 30, `for` loop will work otherwise control will move out from the loop. To achieve this condition (if value of `num` is more than 30) in the next line `if_icmpgt` is given. The role of `if_icmpgt` is just to jump if one integer is greater than another. From the bytecode we can see that at line-9, `if_icmpgt` opcode is given, which has the value 72 and at line-72 `return` statement is given. The next instruction discussed here is `if_icmpne` given at line-15.

```
15: if_icmpne 26
```

This instruction is also a transfer control instruction used to transfer the control if two integers are not equal. The working of `if_icmpne` is that it first pops the top two values of integer type from the operand stack and then compares the values. If the two integers are not equal, execution branches to the address (pc: program counter). But if the integers are equal, execution continues at the next instruction. The general signature for `if_icmpne` opcode is:

```
if_icmpne ...., value1, value2 => .....
```

The first condition (if-statement) given in the try block of JPS11.java is:

```
if (num == 20)
    throw new demo1();
```

Since, initially value of num is 10 and in the if-statement the condition has been given for checking the equality of num with 20 which is not correct, therefore it will throw an exception. Diagrammatically it can be represented as:

| Operand stack | | Operand stack | |
|---------------|-------|---------------|-------|
| Before | After | Before | After |
| Value1 | | 10 | |
| Value2 | | 20 | |
| | | | |

```
value1 = 10 (num)
value2 = 20
```

Since num is not equal to 20 exception is obtained. But if the exception is not obtained, the bytecode given below will execute:

```
18:     new      #2;  //class demo1
21:     dup
22:     invokespecial    #3;  //Method demo1."<init>": ()V
25:     athrow
```

The new opcode is given to create new object for the class demo1. The role of dup opcode is to duplicate the top word given on the stack. Invokespecial is a method invocation opcode which is used to invoke method (demo1 for the program JPS11.java) belonging to a specific class. The athrow opcode is an exception handling opcode. The signature of athrow opcode is:

```
athrow .....handle => [undefined]
```

Rest of the portion is self-explanatory.

24.10 JAVA BASE API

Currently the Java base API is defined to be the Java applet API, described in the next sections. Over time, as the platform develops, this base will grow, as some of the standard extension API migrate into the Java base API. The Java base API is also known as the Java core API.

24.10.1 Java Applet API

The Java applet API, also known as the Java base API, defines the basic building blocks for creating fully functional Java-powered applets and applications. It includes all the classes in the Java package: java.lang, java.util, java.io, java.net, java.awt and java.applet.

24.11 JAVA STANDARD EXTENSION API

In this section, the Java standard extension API is described. These extensions are ‘standard’ in that they form a published, uniform, open API that anyone can implement. Once defined, they can be added to maintain backward compatibility, not changed in a way that calls to them would fail. Over time, new extensions will be added. In addition, some of these extensions will migrate into the Java base API. The current plan for this migration is shown in the Table 24.3.

| Java API | |
|--------------------------|-----------------------------------|
| Migrate to Java Base API | Remain as Java Standard Extension |
| Java 2D | Java 3D |
| Audio | Video, MIDI |
| Java Media Framework | Java Share |
| Java Animation | Java Telephony |
| Java Enterprise | Java Server |
| Java Commerce | Java Management |
| Java Security | |

Table 24.3 Migration from standard extension API to Java base API

An implementation should do whatever is appropriate for the platform it runs on, within its hardware constraints. For example, desktop operating systems that have access to speaker will produce audio; however, a mainframe or network operating system that has no speaker is permitted to do the equivalent of a no-op or some other well-defined behaviour, such as throwing an exception.

24.11.1 Java Security API

The Java security API is a new Java core API, built around the `java.security` package (and its subpackages). This first release includes primarily cryptography functionality, which can be incorporated into Java-based applications. Future releases of this API will include more primitives supporting system security and secure distributed computing.

The Java security API is a framework for developers to easily and securely include security functionality in their applets and applications. This functionality includes cryptography, with digital signatures, encryption and authentication.

The cryptography framework in the Java security API is designed so that a new algorithm can be added later on without much difficulty and can be utilized in the same fashion as existing algorithms. For example, although DSA is the only built-in digital signature algorithm in this release, the framework can easily accommodate another algorithm such as RSA.

APIs for data encryption and other functionalities, together with their implementations, will be released separately in a ‘Java Cryptography Extension’ (JCE) as an add-on package to JDK. These APIs include block and stream cipher, symmetric and asymmetric encryption, and support for multiple modes of operation and multiple encryption.

Java security includes an abstract layer that applications can call. This layer, in turn, makes calls to Java security packages that implement the actual cryptography. This allows third-party developers specializing in providing cryptographic functionality to write packages for Java security. Java security also includes system support for key management, including a secure database, certificate facility, and so on.

This architecture provides for replaceable, upgradeable security. Whenever a strong algorithm or a faster implementation becomes available, modules can be replaced in the platform, in a manner completely transparent to applications.

24.11.2 Java Media API

To support various interactive media on and off the Web Java supports the Java media API. The Java media API mainly defines the multimedia classes given in Java. It is highly extensible, which is composed of various different components. The component of Java media APIs includes audio, video, 2D and 3D animation, telephony, Java speech and Java collaboration. The main focus of Java media API is to provide interfaces to Java programmers to use a wide range of media types within their applications and applets.

The components of the Java media APIs are as follow:

1. **Java 2D API:** It mainly includes line art, images, color, transforms and compositing. It provides an abstract imaging model which extends the functionality of 1.02 AWT package. Java 2D API also provides an extension mechanism to support various arrays of different presentation devices, image formats, color spaces and encoding mechanism.
2. **Java Media Framework (JMF) API:** It specifies a unified architecture, messaging protocol and programming interface for media players, capturing and conferencing. It mainly handles time critical media, such as audio, video and MIDI where synchronization is required. Java media framework will be published as three APIs.

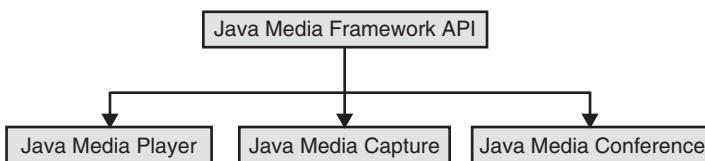


Figure 24.7 Java Media Frame APIs

- *Java media player:* The Java media player will be published first. It is the API used for synchronization, controlling, processing and presenting of compressed streaming and stored timed media, which includes video and audio.
- *Java media capture:* It is used to capture the Web pages.
- *Java media conference:* Especially for conference Web pages and API specification are not yet published.
- *Java collaboration API:* Provides interactive two-way, multiparty communication over a variety of dedicated networks. There will be two versions of Java collaboration.
 - Ver1: Given for sharing of collaboration-aware applications.
 - Ver2: Given for sharing of collaboration-unaware applications.
- *Java telephony API:* It mainly works to integrates telephones with computers. Java telephony gives the idea of '**how to control the phone calls**'. It provides basic functionality for first-party and third-party call control.
- *Java speech:* It provides Java based speech recognition and speech synthesis system.
- *Java animation API:* It provides Java based system for motion and transformation of 2D objects. It uses Java media framework for synchronization, composition and timing.
- *Java 3D API:* It has been used specially for 3D objects. To handle 3D object, Java 3D provides an abstract, interactive imaging model. The 3D API is closely integrated with audio, video, MIDI and animation area.

24.11.3 Java Enterprise API

Java enterprise APIs are mainly given to support connectivity to enterprise databases. There are various types of database architecture like centralized, client-server, multiprocessor and distributed. With the help of Java enterprise APIs, corporate developers are building distributed client/server applets and applications in Java that run on any operating system or hardware platform in the enterprise. There are currently three groups for connectivity: JDBC (Java database connectivity), interface definition language and remote method invocation.

JDBC: JDBC is Java database connectivity, a standard SQL database access interface, which is used for providing access to a wide range of relational databases. JDBC is also used to provide a common base on which higher level tools and interfaces can be developed.

Interface Definition Language (IDL): It is developed to the OMG interface definition language specification. It is a language-neutral way to specify and interface between an object and its client on different platform. Java interface definition language (IDL) is implemented completely in Java. Mainly Java IDL components are used to provide mapping of its method, packages, etc., to IDL operations and features.

Remote Method Invocation: It is given for invocation between peers, or between client and server. RMI performed, serves its purpose when applications at both the ends of the invocation are written in Java.

24.11.4 Java Commerce API

It will bring a methodology for secure purchasing and financial management on Web. The initial component of the Java commerce API is the Java wallet, which defines and implements a client-side framework for credit card, and electronic cash transactions. For secure purchasing and financial management, Java wallet keeps the information about the shopper, payment instruction and details of transactions. It also uses strong cryptographic services that makes the complement process secure.

24.11.5 Java Server API

Java server API provides a uniform and consistent access to the server and administrative system resources. It is an extensible framework that enables and ease the development of a whole spectrum of Java-powered Internet and Intranet servers. Java serves API its own Java ‘servlets’ executable program that users upload to run on networks or servers. Servlets are platform-independent, server-side programming. They can reside on the server.

24.11.6 Java Management API

It provides a wide range of extensible Java objects and methods for building applets that can manage an enterprise network over Internets. It is composed of several different components, each associated with an aspect of the total management space. Since Java management API is a collection of Java classes it provides the building blocks for integrated management. The components of Java management APIs are shown in the diagram below:

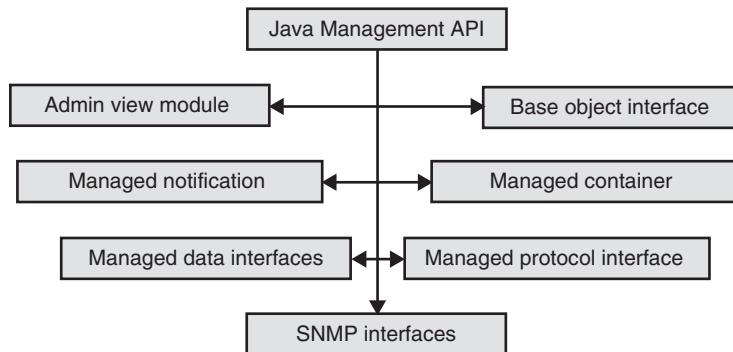


Figure 24.8 Components of Java management APIs

24.12 THE JAVA API PACKAGE

All the core API packages defined by Java and their functionality are summarized in the Table 24.4.

| Package | Description |
|---------------------------|--|
| java.applet | Provides support for applets construction. |
| java.awt | Provides capability of graphical user interfaces. |
| java.awt.color | Provides support for color spaces and profiles. |
| java.awt.datatransfer | Transfers data to and from the system clipboard. |
| java.awt.font | Represent various types of font. |
| java.awt.geom | Facilitate working with geometric shapes. |
| java.awt.im | Allow input of Chinese, Japanese and Korean characters to text editing components. |
| java.awt.dnd | Support for drag-and-drop operations. |
| java.awt.event | Handle events. |
| java.awt.im.spi | Support for alternative input devices. |
| java.awt.image | Process images. |
| java.awt.image.renderable | Support rendering-independent images. |
| java.awt.print | Provide general print capabilities. |
| java.beans | Allows programmer to build software component. |
| java.beans.beancontext | Creates an execution environment for Beans. |
| java.io | Provides input and output files. |
| java.lang | Provides core functionality. |
| java.lang.annotation | Provides support for annotations. |
| java.lang.instrument | Gives support for program instrumentation. |
| java.lang.management | Provides support for management of the execution environment. |
| java.lang.ref | Enables some interaction with the garbage collector. |
| java.lang.reflect | Analyzes code at runtime. |
| java.math | Handles large integers and decimal numbers. |
| java.net | Supports networking. |
| java.nio | Top-level package for the NIO classes and encapsulates buffers. |
| java.nio.channels | Encapsulates channels, used by the NIO system. |
| java.nio.channels.spi | Supports service providers for channels. |
| java.nio.charset | Encapsulates character sets. |
| java.nio.charset.spi | Supports service providers' character set. |
| java.rmi | Provides remote method invocation. |
| java.rmi.activation | Activates persistent objects. |
| java.rmi.dgc | Manages distributed garbage collection. |
| java.rmi.registry | Maps names to remote object references. |
| java.rmi.server | Supports remote method invocation. |
| java.security | Handles certificates, keys, digests, signatures and other security functions. |

(Continued)

| | |
|--------------------------|---|
| java.security.acl | Manages access control lists. |
| java.security.cert | Parses and manages certificates. |
| java.security.interfaces | Defines interfaces for DSA (Digital Signature Algorithm) keys. |
| java.security.spec | Specifies keys and algorithm parameters. |
| java.sql | Communicates with a SQL (Structured Query Language) database. |
| java.util | Contains the common utilities. |
| java.util.concurrent | Supports the concurrent utilities. |
| java.util.jar | Creates and reads JAR files. |
| java.util.logging | Supports logging of information relating to user preference. |
| java.util.prefs | Encapsulates information relating to user and system preferences. |
| java.util.regex | Supports regular expression processing. |
| java.util.spi | Supports service providers for the utility classes in java.util. |
| java.util.zip | Reads and writes compressed and uncompressed ZIP files. |

Table 24.4 The core Java API packages

24.13 PONDERABLE POINTS

1. The Java platform has two main parts: the Java virtual machine and the Java API.
2. The Java API specifies a set of essential interfaces in a growing number of key areas that developers will use to build their Java-powered applications.
3. Java API basically has two parts: Java base API and Java standard extension API.
4. The Java virtual machine is key to the independence of the underlying operating system and hardware.
5. The Java base API is defined to be the Java applet API.
6. The Java applet API, also known as the Java base API, defines the basic building blocks for creating Java-powered applets.
7. Java applet API includes all the classes in the Java package: java.lang, java.util, java.io, java.net, java.awt and java.applet.
8. The Java security API is a framework for developers to easily and securely include security functionality in their applets.
9. Java security includes an abstract layer that applications can call.
10. The Java media API defines the multimedia classes that support a wide range of rich, interactive media on and off the Web.
11. The Java media API includes audio, video, 2D and 3D animation, telephony and collaboration.
12. Java 2D API provides graphics and imaging capabilities beyond those available with the Java applet API.
13. Java media framework API handles traditional time critical media such as audio, video and MIDI.
14. Java animation API supports 2D animations.
15. Java share API provides the basic abstraction for live, two-way, multi-party communication between objects over a variety of networks and transport protocols.
16. Java 3D API provides high-performance, interactive, 3D graphics support.
17. Java enterprise API supports three groups for connectivity: JDBC (Java database connectivity), interface definition language and remote method invocation.

18. Java server API is an extensible framework that enables and eases the development of a whole spectrum of Java-powered Internet and intranet server.
19. Java management API is a collection of Java classes that provides the building blocks for integrated management.

REVIEW QUESTIONS

1. What is the Java platform?
2. Explain the term Java base platform and embedded Java platform.
3. Explain the term Java API. Also explain Java base API and the Java standard extension API.
4. Explain the functionality of Java security API with example.
5. Why do we use Java media API? Also explain the components of the Java APIs.
6. Explain the term Java enterprise API.
7. What do you understand by Java commerce API? Also explain the functionality of the Java wallet.
8. Explain the working of Java server API.
9. Give the functionality of the components of Java management APIs.
10. Describe the virtual machine in detail.
11. What do you understand by instruction set for virtual machine.
12. Give the procedure for invoking method using virtual machine.
13. What is class instance?
14. What is cryptographic package provider used for API?
15. Give the detailed procedure for debugging in API environment.
16. Explain the term ‘remote debugging’ for API.
17. Explain Java debugger for API environment.
18. Explain the Java classes used for debugging APIs. Also explain the data types associated with Java debugging API.

Multiple Choice Questions

1. Java platform has the part
 - (a) Java virtual machine (c) (a) and (b)
 - (b) Java API (d) None of the above
2. The virtual machine defines a machine independent format for binary files which is called as:
 - (a) .exe file (c) .java file
 - (b) .class file (d) .dat file
3. To see the bytecode of Java’s program we will use
 - (a) javac -c filename>filename.bc
 - (b) javah -c filename>filename.bc
 - (c) javap -c filename>filename.bc
 - (d) java -c filename>filename.bc
4. Any of the virtual machine errors listed as subclasses of _____, may be thrown at any time during the operation of the Java virtual machine.
 - (a) VirtualMachineError
 - (b) MachineError
 - (c) MachineExecutionError
 - (d) None of the above
5. In Java, method invocation is done by
 - (a) invokevirtual instruction
 - (b) bipushvirtual instruction
6. For the method


```
int sum10and20 ()
{
    return addTwoValue(10, 20);
}
```

This compiles

```
0  aload_0
1  bipush 10
3  bipush 20
8  ireturn
```

Here ireturn is used to

 - (a) just to return the value by default
 - (b) take int value by addTwoValue
 - (c) return value to sum
 - (d) none of the above
7. The invokespecial instruction must be used to
 - (a) invoke variables
 - (b) invoke stack storage area
 - (c) invoke instance initialization <init> method
 - (d) none of the above

8. Java virtual machine class instances are created using the Java virtual machine by
 - (a) Using invokespecial instruction
 - (b) Using <init> instruction
 - (c) Using aload_0 instruction
 - (d) Using new instruction
9. Which of the following is the method of DebuggerCallBack class?
 - (a) threadDeathEvent()
 - (b) void threadDeathEvent(int a)
10. For getting the code position when we are using RemoteStackFrame we will use
 - (a) getRemoteClass()
 - (b) getLocalVariable()
 - (c) getpc()
 - (d) None of the above

KEY FOR MULTIPLE CHOICE QUESTIONS

1. c 2. b 3. c 4. a 5. a 6. b 7. c 8. d 9. a 10.c

Sample Project

S.1 INTRODUCTION

Many a times need arises to reduce the workload of any institutional library. To achieve this a software for library management system called ‘Library System’ has been implemented. This software has an architecture that provides several facilities for the employees of the library. Also, this software works efficiently for the members in issuing, giving details about the types of books available in the library and returning the books.

The project covers the details about the following functions:

- Borrowing books
- Transaction records
- Add members
- Add books
- Search the title
- Returning books
- Important notes
- Designed by

S.2 CODING

```
/*FILE : TIMERTIEM.JAVA*/
```

```
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.JLabel;
import java.util.Calendar;
import java.util.Date;
import java.text.DateFormat;
import javax.swing.JLabel;
import java.awt.*;
import java.awt.event.*;
public class TimerTime
```

```
{ Timer timer;
  Calendar cal;
  Date date;
  DateFormat dateFormatter;
  /* Constructor used for running example main.*/
  public TimerTime(boolean visible,JLabel label,int seconds)
  {
    timer = new Timer();
    timer.scheduleAtFixedRate(new DoTime(visible,label),0,
                             seconds*1000);
  }
  /* Constructor used for creating class in another
   application */
  public TimerTime(int seconds)
  {
    this(new JLabel(),seconds);
  }
  public TimerTime()
  {
    this(new JLabel(),1);
  }
  public TimerTime(JLabel label)
  {
    this(label, 1);
  }
  public TimerTime(JLabel label, int seconds)
  {
    timer = new Timer();
    timer.scheduleAtFixedRate(new DoTime(label),0, seconds*1000);
  }
  class DoTime extends TimerTask
  {
    JLabel label = new JLabel();
    boolean visible = false;
    public DoTime(){}
    public DoTime(boolean visible, JLabel label)
    {
      this.label = label;
      this.visible = visible;
    }
    public DoTime(JLabel label){this.label = label;
    }
    public void run()
    {
      cal = Calendar.getInstance();
      date = cal.getTime();
      dateFormatter = DateFormat.getTimeInstance();
      label.setText(dateFormatter.format(date));
    }
    public void timerCancel()
```

```
{  
    timer.cancel();  
}  
public void timerCancel()  
{  
    timer.cancel();  
}  
}
```

```
/*FILE: MEMBER.JAVA */
```

```
public class Member  
{  
    String MemberName, MemberID, MemAddress, MemberIC;  
    int MemberAge, MemPhone, BookCount;  
  
    Member(String Name, int Age, String IC, String  
            Address, int Telephone, int Count)  
    {  
        MemberName = Name;  
        MemberAge = Age;  
        MemberIC = IC;  
        MemAddress = Address;  
        MemPhone = Telephone;  
        MemberID = IC;  
        BookCount = Count;  
    }  
    public String getName()  
    {  
        return MemberName;  
    }  
    public int getAge()  
    {  
        return MemberAge;  
    }  
    public String getIC()  
    {  
        return MemberIC;  
    }  
    public String getAddress()  
    {  
        return MemAddress;  
    }  
    public int getPhone()  
    {  
        return MemPhone;  
    }  
    public String getMemberID()  
    {
```

```
        return MemberID;
    }
    public int getBookCount()
    {
        return BookCount;
    }
    public void IncreaseBookCount()
    {
        BookCount++;
    }
    public void DecreaseBookCount()
    {
        BookCount--;
    }
}
```

```
/*FILE: FACTORYTRANSACTION.JAVA */
```

```
import javax.swing.*;
import java.awt.*;
import java.text.*;
public class FactoryTransaction
{
    Transaction[] t;
    int i = 0;
    int j;
    DecimalFormat df = new DecimalFormat("##0.00");
    public FactoryTransaction()
    {
        t = new Transaction[1000];
    }
    public void CreateBookTrans(String MemberID, int
        BookID, int BDate, int RDate, double Cost)
    {t[getTransIDSize()] = new Transaction(MemberID,
        BookID, BDate, RDate, Cost);
    }
    public int getTransIDSize()
    {
        while (t[i] != null)
        {
            i++;
        }
        return i;
    }
    public void getIDDDetails (String ID)
    {
        JPanel p1 = new JPanel();
        p1.setLayout(new GridLayout(28,0));
    }
}
```

```

        for(int j = 0; j <= getTransIDSize()-1; j++)
        {
            if(ID.equals(t[j].getTransMemberID()))
            {
                p1.setBorder(BorderFactory.createTitledBorder
                    ("Transaction Details:"));

                JLabel ddd = new JLabel("===== =====");
                =====;

                JLabel tid = new JLabel("Transaction ID : "+(j+1));
                JLabel mid = new JLabel("Member ID : "+t[j].
                    getTransMemberID());
                JLabel bid = new JLabel("Book ID : "+t[j].
                    getTransMBookID());
                JLabel dob = new JLabel("Date of Borrow : "+t[j].
                    getBDate());
                JLabel dod = new JLabel("Date of Return : "+t[j].
                    getRDate());

                p1.add(tid);
                p1.add(mid);
                p1.add(bid);
                p1.add(dob);
                p1.add(dod);

                if(t[j].getRDate() == 0)
                {
                    JLabel bs = new JLabel("Book Has Not been
                        Returned");
                    p1.add(bs);
                }
                else
                {
                    if (t[j].calculateCost(t[j].getRDate()) < 0.00)
                    {
                        JLabel f = new JLabel("No Fines Aquired");
                        p1.add(f);
                    }
                    else
                    {
                        JLabel f1 = new JLabel("Fine Issued : $"
                            +df.format(t[j].calculateCost(t[j].getRDate())));
                        p1.add(f1);
                    }
                }
                p1.add(ddd);
            }
        }

        JOptionPane.showMessageDialog(null,p1,"Transaction
            Record",JOptionPane.INFORMATION_MESSAGE);
    }

    public String UpdateBookTransaction(int BookID,int
        DateReturned)
}

```

```

{
    JPanel p2 = new JPanel();
    p2.setLayout(new GridLayout(6,6));
    for(int j = 0; j <= getTransIDSsize()-1; j++)
    {
        if(BookID==t[j].getTransMBookID() &&t[j].getRDate()==0)
        {
            p2.setBorder(BorderFactory.createTitledBorder
                ("Transaction Details of Transaction ID:"+(j+1)));
            JLabel tid1 = new JLabel("Transaction ID :" +(j+1));
            JLabel mid1 = new JLabel("Member ID : "+t[j].
                getTransMemberID());
            JLabel bid1 = new JLabel("Book ID : "+t[j].
                getTransMBookID());
            JLabel dob1 = new JLabel("Date of Borrow : "+t[j].
                getBDate());
            JLabel dod1 = new JLabel("Date of Return : "+t[j].
                setRDate(DateReturned));
            p2.add(tid1);
            p2.add(mid1);
            p2.add(bid1);
            p2.add(dob1);
            p2.add(dod1);
            if (t[j].calculateCost(DateReturned) < 0.50)
            {
                JLabel fine = new JLabel("No Fines Aquired");
                p2.add(fine);
            }
        }
        else
        {
            JLabel fine1 = new JLabel("Fine Issued : $" + df.
                format(t[j].calculateCost(DateReturned)));
            p2.add(fine1);
        }
    }
}
 JOptionPane.showMessageDialog(null,p2,"Transaction
 Record",JOptionPane.INFORMATION_MESSAGE);
 return t[j].getTransMemberID();
}
}
}

```

/*FILE: TRANSACTION.JAVA */

```

import java.text.*;
public class Transaction
{
    String MemberID2 ;
    int BookID2, DateB, DateR;
    double Cost1;
    public Transaction(String MemberID, int BookID, int

```

```

        BDate, int RDate, double Cost) {
    MemberID2 = MemberID;
    BookID2 = BookID;
    DateB = BDate;
    DateR = RDate;
    Cost1 = Cost;
}
public String getTransMemberID()
{
    return MemberID2;
}
public int getTransMBookID()
{
    return BookID2;
}
public int getBDate()
{
    return DateB;
}
public double getCost()
{
    return Cost1;
}
public int setRDate(int ReturnedDate)
{
    DateR = ReturnedDate;
    return DateR;
}
public int getRDate()
{
    return DateR;
}
public double calculateCost(int ReturnedDate)
{
    Cost1 = (((ReturnedDate-getBDate())-14)*0.10);
    return Cost1;
}
}

```

/*FILE: FACTORYMEMBER.JAVA */

```

import javax.swing.*;
import java.awt.*;
public class FactoryMember
{
    Member[] m;
    private int i = 0;
    private int y = 0;
    private int z;
    public FactoryMember()

```

```

{
    m = new Member[50];
    m[0]=new Member("Hari Mohan Pandey",27, "FF0209",
        "SVKM's NMIMS University Shirpur", 7542473, 0);
    m[1]=new Member("Shreedhar Deshmukh", 30, "FF0210",
        "SVKM's NMIMS University Shirpur", 8997598, 0);
    m[2] = new Member("Sonali Borse", 23, "FF0211",
        "SVKM's NMIMS University Shirpur", 5611300, 0);
    m[3] = new Member("Soniya Relan", 25, "FF0212",
        "SVKM's NMIMS University Shirpur", 2438973, 0);
    m[4] = new Member("Malvika Sharma", 35, "FF0213",
        "SVKM's NMIMS University Shirpur", 3103109, 0);
    m[5] = new Member("Yogesh Choudhary", 26, "FF0214",
        "SVKM's NMIMS University Shirpur", 7861194, 0);
    m[6] = new Member("Priya Nath", 20, "FF0215",
        "SVKM's NMIMS University Shirpur", 5844182, 0);
    m[7] = new Member("Drisha Trivedi",20, "FF0216",
        "SVKM's NMIMS University Shirpur", 7486555, 0);
    m[8] = new Member("Apoorva Sharma", 20, "FF0217",
        "SVKM's NMIMS University Shirpur", 97489129, 0);
    m[9] = new Member("Manali Kapadia", 20, "FF0218",
        "SVKM's NMIMS University Shirpur", 2866027, 0);
}
public int getMemberSize()
{
    while (m[i] != null) {
        i++;
    }
    return i;
}
public void getStatsUsingName(String Name)
{
    for(int j = 0; j <= getMemberSize()-1; j++)
    {
        if(Name.equalsIgnoreCase(m[j].getName()))
        {
            JPanel p1 = new JPanel();
            p1.setLayout(new GridLayout(7,7));
            JLabel mnl = new JLabel("Name : ");
            JTextField mnt = new JTextField
                (""+m[j].getName());
            mnt.setEditable(false);
            JLabel mal = new JLabel("Age : ");
            JTextField mat = new JTextField
                (""+m[j].getAge());
            mat.setEditable(false);
            JLabel micl = new JLabel("IC No : ");
            JTextField mict = new JTextField
                (""+m[j].getIC());
            mict.setEditable(false);
        }
    }
}

```

```

JLabel madl = new JLabel("Address : ");
JTextField madt = new JTextField
        (""+m[j].getAddress());
madt.setEditable(false);
JLabel mtl=new JLabel("Telelphone number ");
JTextField mtt = new JTextField
        (""+m[j].getPhone());
mtt.setEditable(false);
JLabel midl = new JLabel("Member ID : ");
JTextField midt = new JTextField
        (""+m[j].getMemberID());
midt.setEditable(false);
JLabel mbcl=new JLabel("Book Counts : ");
JTextField mbct = new JTextField
        (""+m[j].getBookCount());
mbct.setEditable(false);
p1.add(mnl);
p1.add(mnt);
p1.add(mal);
p1.add(mat);
p1.add(micl);
p1.add(mict);
p1.add(madl);
p1.add(madt);
p1.add(mtl);
p1.add(mtt);
p1.add(midl);
p1.add(midt);
p1.add(mbcl);
p1.add(mbct);
p1.add(mbct);

JOptionPane.showMessageDialog(null,p1,"Member Record",
                JOptionPane.INFORMATION_MESSAGE
            }
        }
    }

public void getStatsUsingIC (String IC)
{
    for(int j = 0; j <= getMemberSize()-1; j++)
    {
        if(IC.equalsIgnoreCase(m[j].getIC()))
        {
            JPanel p2 = new JPanel();
            p2.setLayout(new GridLayout(7,7));
            JLabel mnl2 = new JLabel("Name : ");
            JTextField mnt2 = new JTextField(""+m[j].
                    getName());
            mnt2.setEditable(false);
            JLabel mal2 = new JLabel("Age : ");
            JTextField mat2 = new JTextField(""+m[j].
                    getAge());
            mat2.setEditable(false);
        }
    }
}

```

```

JLabel micl2 = new JLabel("IC No : ");
JTextField mict2 = new JTextField(""+m[j].
                                         getIC());
mict2.setEditable(false);
JLabel madl2 = new JLabel("Address : ");
JTextField madt2 = new JTextField
                    ("."+m[j].getAddress());
madt2.setEditable(false);
JLabel mt12 = new JLabel("Telephone
                           number : ");
JTextField mtt2 = new JTextField(""+m[j].
                                         getPhone());
mtt2.setEditable(false);
JLabel midl2 = new JLabel("Member ID : ");
JTextField midt2 = new JTextField
                    ("."+m[j].getMemberID());
midt2.setEditable(false);
JLabel mbcl2 = new JLabel("Book Counts : ");
JTextField mbct2 = new JTextField
                    ("."+m[j].getBookCount());
mbct2.setEditable(false);
p2.add(mn12);
p2.add(mnt2);
p2.add(mal2);
p2.add(mat2);
p2.add(mic12);
p2.add(mict2);
p2.add(madl2);
p2.add(madt2);
p2.add(mt12);
p2.add(mtt2);
p2.add(midl2);
p2.add(midt2);
p2.add(mbcl2);
p2.add(mbct2);

 JOptionPane.showMessageDialog(null,p2,"Member Record",
                               JOptionPane.INFORMATION_MESSAGE);
}
}

public void addMember(String Name, int Age, String IC, String
                      Address, int Telephone)
{
    m[getMemberSize()]=new Member (Name,Age,IC,Address,
                                  Telephone,0);
}

public int getMemberBookCount(String MemberID)
{
    for(int j = 0; j <= getMemberSize()-1; j++)
    {

```

```
        if(MemberID.equals(m[j].getIC())))
    {
        y = m[j].getBookCount();
    }
}
return y;
}
public void UpdateBookCount (String MemberID)
{
    for(int j = 0; j <= getMemberSize()-1; j++)
    {
        if(MemberID.equals(m[j].getIC()))
        {
            m[j].IncreaseBookCount();
        }
    }
}
public void ReturnBookCount (String MemberID)
{
    for(int j = 0; j <= getMemberSize()-1; j++)
    {
        if(MemberID.equals(m[j].getIC()))
        {
            m[j].DecreaseBookCount();
        }
    }
}
```

```
/*FILE: FACTORYBOOK.JAVA*/
```

```
B[3]= new Books("Trouble Free C", "Hari Mohan  
Pandey"  
, "University Science Press", 4, "Available");  
B[4]= new Books("Trouble Free C", "Hari Mohan  
Pandey"  
, "University Science Press", 5, "Available");  
B[5]= new Books("Trouble Free C", "Hari Mohan  
Pandey"  
, "University Science Press", 4, "Available");  
B[6] = new Books("Data Structure And Algorithms",  
"Hari Mohan Pandey", "University Science  
Press" , 7, "Available");  
B[7] = new Books("Data Structure And Algorithms",  
"Hari Mohan Pandey", "University Science  
Press", 8, "Available");  
B[8] = new Books("Data Structure And Algorithms",  
"Hari Mohan Pandey", "University Science  
Press", 9, "Available");  
B[9] = new Books("Trouble Free C++", "Hari Mohan  
Pandey", "Ane Book ", 10, "Available");  
B[10] = new Books("Trouble Free C++", "Hari Mohan  
Pandey", "Ane Book ", 11, "Available");  
B[11] = new Books("Trouble Free C++", "Hari Mohan  
Pandey", "Ane Book ", 12, "Available");  
B[12] = new Books("Sea of Java", "Hari Mohan Pandey",  
"Pearson Education", 13, "Available");  
B[13] = new Books("Sea of Java", "Hari Mohan Pandey",  
"Pearson Education", 14, "Available");  
B[14] = new Books("Sea of Java", "Hari Mohan Pandey",  
"Pearson Education", 13, "Available");  
B[15] = new Books("Java How To Program", "Deitel,  
Harvey M./ Deitel, Paul J.", "Prentice Hall ",  
16, "Available");  
B[16] = new Books("Java How To Program", "Deitel,  
Harvey M./ Deitel, Paul J.", "Prentice Hall ",  
17, "Available");  
B[17] = new Books("Java How To Program", "Deitel,  
Harvey M./ Deitel, Paul J.", "Prentice Hall ",  
18, "Available");  
B[18] = new Books("C++ How To Program", "Deitel, H.  
M./ Deitel, P. J.", "Prentice Hall ", 19,  
"Available");  
B[19] = new Books("C++ How To Program", "Deitel, H.  
M./ Deitel, P. J.", "Prentice Hall ", 20,  
"Available");  
B[20] = new Books("C++ How To Program", "Deitel, H.  
M./ Deitel, P. J.", "Prentice Hall ", 21,  
"Available");  
B[21] = new Books("Flash 5 Visual Jumpstart",  
"Hartman, Patricia A./ Hartman", "Sybex",  
22, "Available");
```

```

B[22] = new Books("Flash 5 Visual Jumpstart",
                  "Hartman, Patricia A./ Hartman", "Sybex",
                  23, "Available");
B[23] = new Books("Flash 5 Visual Jumpstart",
                  "Hartman, Patricia A./ Hartman", "Sybex",
                  24, "Available");
B[24] = new Books("Inside the Adobe Photoshop 6
                  Studio", "Mullin, Eileen", "Prima
                  Publications ", 25, "Available");
B[25] = new Books("Inside the Adobe Photoshop 6
                  Studio", "Mullin, Eileen", "Prima
                  Publications ", 26, "Available");
B[26] = new Books("Inside the Adobe Photoshop 6
                  Studio", "Mullin, Eileen", "Prima
                  Publications ", 27, "Available");
B[27] = new Books("Adobe Photoshop 6.0 Web Design ",
                  "Baumgardt, Michael", "Adobe Press", 28,
                  "Available");
B[28] = new Books("Adobe Photoshop 6.0 Web Design ",
                  "Baumgardt, Michael", "Adobe Press", 29,
                  "Available");
B[29] = new Books("Adobe Photoshop 6.0 Web Design ",
                  "Baumgardt, Michael", "Adobe Press", 30,
                  "Available");
count = getBookSize();
}
public int getBookSize() {
    while (B[i] != null) {
        i++;
    }
    return i;
}
public void getStatsUsingBName(String Name) {
    for(int j = 0; j <= getBookSize()-1; j++) {
        if(Name.equalsIgnoreCase( B[j].getName()))
    {
        JPanel p1 = new JPanel();
        p1.setLayout(new GridLayout(5,5));
        JLabel bnl = new JLabel("Book Name : ");
        JTextField bnt = new
                           JTextField(""+B[j].getName());
        bnt.setEditable(false);
        JLabel bal = new JLabel("Book Author : ");
        JTextField bat = new JTextField
                           ("."+B[j].getAuthor());
        bat.setEditable(false);
        JLabel bpl = new JLabel("Book Publisher : ");
        JTextField bpt = new JTextField
                           ("."+B[j].getPublisher());
        bpt.setEditable(false);
    }
}

```

```

JLabel bidl = new JLabel("Book ID : ");
JTextField bidt = new JTextField
        (""+B[j].getID());
bidt.setEditable(false);
JLabel bsl = new JLabel("Book Status : ");
JTextField bst = new JTextField
        (""+B[j].getStatus());
bst.setEditable(false);
p1.add(bnl);
p1.add(bnt);
p1.add(bal);
p1.add(bat);
p1.add(bp1);
p1.add(bpt);
p1.add(bidl);
p1.add(bidt);
p1.add(bsl);
p1.add(bst);
 JOptionPane.showMessageDialog(null,p1,"Book
Record", JOptionPane.INFORMATION_MESSAGE);
}
}

public void getStatsUsingID (int ID) {
//#####ID is invalid#####
if(ID >getBookSize()){
JOptionPane.showMessageDialog(null,"Book not Found. The
BookID you have entered is invalid","Book not
Found", JOptionPane.ERROR_MESSAGE);
}

//#####check for books in the available books#####
for(int j = 0; j <= getBookSize()-1; j++) {
    if(ID == B[j].getID()){

//#####
        JPanel plid = new JPanel();
        plid.setLayout(new GridLayout(5,5));
        JLabel idbnl = new JLabel("Book Name      : ");
        JTextField idbnt = new JTextField(""+B[j].getName());
        idbnt.setEditable(false);
        JLabel idbal = new JLabel("Book Author : ");
        JTextField idbat = new JTextField
                (""+B[j].getAuthor());
        idbat.setEditable(false);
        JLabel idbpl = new JLabel("Book Publisher : ");
        JTextField idbpt = new JTextField
                (""+B[j].getPublisher());
        idbpt.setEditable(false);
        JLabel idbidl = new JLabel("Book ID : ");

```

```

JTextField idbidt = new JTextField(""+B[j].getID());
idbidt.setEditable(false);
JLabel idbsl = new JLabel("Book Status : ");
JTextField idbst = new JTextField
        (""+B[j].getStatus());
idbst.setEditable(false);
plid.add(idbn1);
plid.add(idbnt);
plid.add(idbal);
plid.add(idbat);
plid.add(idbp1);
plid.add(idbpt);
plid.add(idbid1);
plid.add(idbidt);
plid.add(idbsl);
plid.add(idbst);
JOptionPane.showMessageDialog(null,plid,"Book Record"
        , JOptionPane.INFORMATION_MESSAGE);
////////////////////////////////////////////////////////////////////////#
}
}
}
public void addBook(String BName, String BAuthor, String
BPublisher,int BID) {
    B[getBookSize()] = new Books(BName, BAuthor, BPublisher,
    BID, "Available");
}
public void setBorrowStatus(int ID) {
    for(int j = 0; j <= getBookSize()-1; j++) {
        if(ID == B[j].getID())
        {
            B[j].setBorrowStatus();
        }
    }
}
public void setReturnStatus(int ID) {
    for(int j = 0; j <= getBookSize()-1; j++) {
        if(ID == B[j].getID())
        {
            B[j].setReturnStatus();
        }
    }
}
public String getBookStatus(int BookID) {
    for(int j = 0; j <= getBookSize()-1; j++) {
        if(BookID == B[j].getID())
        {
            z = B[j].getStatus();
        }
    }
}

```

```
    }  
}  
return z;  
}  
}
```

```
/*FILE: CONTROLLER.JAVA*/
import javax.swing.*;
import java.awt.*;

public class Controller
{
    private int Count = 0;
    FactoryMember fm;
    FactoryBook bm;
    FactoryTransaction ft;
    JPanel p1 = new JPanel();
    JPanel p2 = new JPanel();
    JTextField bsct = new JTextField();
    JTextField bsct1 = new JTextField();
    JTextField nobt = new JTextField();
    public Controller() {
        fm = new FactoryMember();
        bm = new FactoryBook();
        ft = new FactoryTransaction();
    }
    //*****
    p1.setLayout(new GridLayout(5,0));
    JLabel nob = new JLabel("No. of Books Member Borrowed : ");
    JLabel bsc = new JLabel("Book Status Changed to : ");
    JLabel bbc = new JLabel("Book Borrow Confirmed");
    bsct.setEditable(false);
    nobt.setEditable(false);
    p1.add(nob);
    p1.add(nobt);
    p1.add(bsc);
    p1.add(bsct);
    p1.add(bbc);
    //*****
    p2.setLayout(new GridLayout(3,0));
    JLabel bscl = new JLabel("Book Status Changed to : ");
    JLabel bbc1 = new JLabel("Book Return Confirmed");
    bsct1.setEditable(false);
    p2.add(bscl);
    p2.add(bsct1);
    p2.add(bbc1);
}
public void ControllerNameRetrieve(String Name)
{
    fm.getStatsUsingName(Name);
}
```

```

}

//*****
public void ControllerICRetrieve(String IC)
{
    fm.getStatsUsingIC(IC);
}
//*****
public void ControllerAddMember(String Name, int Age, String
IC, String Address, int Telephone)
{
    fm.addMember(Name,Age,IC,Address,Telephone);
}
public void ControllerBookNameRetrieve(String BName) {
    bm.getStatsUsingBName(BName);
}
public void ControllerBookIDRetrieve(int ID) {
    bm.getStatsUsingID(ID);
}
public void ControllerAddBook(String BName, String BAuthor,
String BPublisher, int BID) {
    bm.addBook(BName, BAuthor, BPublisher, BID);
}
public void BorrowBook(String MemberID, int BookID, int
DateBorrowed) {
    if(fm.getMemberBookCount(MemberID) == 4 )
    {
        JOptionPane.showMessageDialog(null,"You have
reached the maximum no of books\n that you can
borrow" , "No of book limits reached",
JOptionPane.INFORMATION_MESSAGE);
    }
    else if(bm.getBookStatus(BookID).equals("Unavailable"))
    {
        JOptionPane.showMessageDialog(null,"The Book was out
of library" , "Book not Available"
,JOptionPane.INFORMATION_MESSAGE);
    }
    else if (fm.getMemberBookCount(MemberID) < 4 && bm
        .getBookStatus(BookID).equals("Available"))
    {
        fm.UpdateBookCount(MemberID);
        bm.setBorrowStatus(BookID);
        ft.CreateBookTrans(MemberID,BookID,DateBorrowed, 0, 0.0);
        //-----
        bsct.setText(bm.getBookStatus(BookID));
        nobt.setText(""+fm.getMemberBookCount(MemberID));
        //-----
        JOptionPane.showMessageDialog(null,p1,"Member's Status",
    }
}

```

```

        JOptionPane.INFORMATION_MESSAGE);
    }
}

public void ReturnBook(int BookID, int DateReturned) {
    bm.setReturnStatus(BookID);
    fm.ReturnBookCount(ft.UpdateBookTransaction(BookID,
        DateReturned));
//-----
// bsctl.setText(""+bm.getBookStatus(BookID));
//-----
//JOptionPane.showMessageDialog(null,p2," Status of Book No
//"+BookID,JOptionPane.INFORMATION_MESSAGE);
}
public void checkTrans(String MemberID) {
    ft.getIDDetails(MemberID);
}
}

```

/*FILE: BOOKS.JAVA*/

```

public class Books
{
    String BookName,BookAuthor, BookPublisher,BookStatus;
    int BookID;

    Books(String BName, String BAuthor, String BPublisher,
    int BID,String BStatus)
    {
        BookName = BName;
        BookAuthor = BAuthor;
        BookPublisher = BPublisher;
        BookID = BID;
        BookStatus = BStatus;
    }
    public String getName()
    {
        return BookName;
    }
    public String getAuthor()
    {
        return BookAuthor;
    }
    public int getID()
    {
        return BookID;
    }
    public String getPublisher()
    {

```

```
        return BookPublisher;
    }
    public String getStatus()
    {
        return BookStatus;
    }
    public void setBorrowStatus()
    {
        BookStatus = "Unavalible";
    }
    public void setReturnStatus()
    {
        BookStatus = "Avablele";
    }
}
//*********************************************************************
//***** Subject : SEA OF JAVA *****/
//***** Project : Library System *****/
//*****Name : HARI MOHAN PANDEY *****/
//*****************************************************************/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

#####
public class GUI extends JFrame
{
    Controller c = new Controller();
    JTabbedPane tpane = new JTabbedPane();
    /*-----Components used in add member -----*/
    JPanel AddMemberPanel,AddMemberPanell;
    JLabel AddMemberNameLabel,AddAgeLabel,AddICLabel,
           AddAddressLabel,AddMemberTelLabel;
    JTextField AddMemberNameField,AddAgeField,AddICField,
           AddAddressField,AddMemberTelField;
    JButton AddMemberButton;
    /*-----Components used in add book -----*/
    JPanel AddBookPanel,AddBookPanell;
    JLabel AddBookNameLabel,AddBookPublisherLabel,
           AddBookAuthorLabel,AddBookIDLabel;
    JTextField AddBookNameField, AddBookPublisherField,
           AddBookAuthorField,AddBookIDField;
    JButton AddBookButton;
    /*-----Components used in search books -----*/
    JPanel SearchBookPanel;
    JLabel SearchBookIDLabel;
    JTextField SearchBookIDField;
    JButton SearchBookButton;
    JPanel SearchBook1Panel;
```

```

JLabel SearchBook1IDLabel;
JTextField SearchBook1IDField;
JButton SearchBook1Button;
/*-----Components used in search members -----*/
-----*/
JPanel SearchMemberPanel;
JLabel SearchMemberIDLabel;
JTextField SearchMemberIDField;
JButton SearchMemberButton;

JPanel SearchMember1Panel;
JLabel SearchMember1IDLabel;
JTextField SearchMember1IDField;
JButton SearchMember1Button;
/*-----Components used in returning books -----*/
JPanel ReturnPanel,ReturnPanel1;
JLabel ReturnDateLabel, ReturnMemberIDLabel;
JTextField ReturnDateField, ReturnBookIDField;
JButton ReturnDateButton;

/*-----Components used in borrowing -----*/
JPanel BorrowPanel,BorrowPanel1;
JLabel BorrowDateLabel,BorrowMemberLabel,
BorrowBookIDLabel;
JTextField BorrowDateField,BorrowMemberField,
BorrowBookIDField;
JButton BorrowButton;
/*-----Components used in transaction
-----*/
JPanel TransactionPanel;
JLabel TransMemberIDLabel;
JTextField TransMemberField;
JButton CheckTransButton;

public GUI()
{
super("LIBRARY SYSTEM BY HARI MOHAN PANDEY FACULTY
CE DEPARTMENT");
/*-----Panels used in our project-----*/
AddMemberPanel      = new JPanel();
AddMemberPanel1     = new JPanel();
AddBookPanel       = new JPanel();
AddBookPanel1      = new JPanel();
SearchBookPanel    = new JPanel();
SearchBook1Panel   = new JPanel();
SearchMemberPanel  = new JPanel();
SearchMember1Panel = new JPanel();
ReturnPanel        = new JPanel();
ReturnPanel1       = new JPanel();
BorrowPanel        = new JPanel();

```

```

        BorrowPanel1           = new JPanel();
        TransactionPanel       = new JPanel();
/*-----textfields initialization used in our project-----*/
        AddMemberNameField     = new JTextField(8);
        AddAgeField            = new JTextField(8);
        AddICField              = new JTextField(8);
        AddAddressField         = new JTextField(8);
        AddMemberTelField       = new JTextField(8);
        AddBookNameField        = new JTextField(8);
        AddBookPublisherField   = new JTextField(8);
        AddBookAuthorField      = new JTextField(8);
        AddBookIDField          = new JTextField(8);
        SearchBookIDField       = new JTextField(8);
        SearchBook1IDField      = new JTextField(8);
        SearchMemberIDField     = new JTextField(8);
        SearchMember1IDField    = new JTextField(8);
        ReturnDateField          = new JTextField(8);
        ReturnBookIDField        = new JTextField(8);
        BorrowDateField          = new JTextField(8);
        BorrowMemberField        = new JTextField(8);
        BorrowBookIDField        = new JTextField(8);
        TransMemberField         = new JTextField(8);

/*-----Labels initialization used in our project---*/
        AddMemberNameLabel = new JLabel("Member Name: ");
        AddAgeLabel = new JLabel("Age : ");
        AddICLabel = new JLabel("IC No : ");
        AddAddressLabel = new JLabel("Address: ");
        AddMemberTelLabel = new JLabel("Telephone : ");
        AddBookNameLabel = new JLabel("Book Name : ");
        AddBookPublisherLabel= new JLabel("Publisher Name: ");
        AddBookAuthorLabel = new JLabel("Author Name: ");
        AddBookIDLabel = new JLabel("Book ID : ");
        SearchBookIDLabel = new JLabel("Enter Book ID:");
        SearchBook1IDLabel = new JLabel("Enter Book Name: ");
        SearchMemberIDLabel= new JLabel("Enter Member ID: ");
        SearchMember1IDLabel= new JLabel("Enter Member Name: ");
        ReturnDateLabel = new JLabel("Enter Return Date:");
        ReturnMemberIDLabel = new JLabel("Enter Book ID:");
        BorrowDateLabel = new JLabel("Enter Date Borrow:");
        BorrowMemberLabel= new JLabel("Enter Member ID:");
        BorrowBookIDLabel= new JLabel("Enter Book ID: ");
        TransMemberIDLabel = new JLabel("Enter Member ID :");

/*-----Buttons initialization used in our project----*/
        AddMemberButton = new JButton(" Create New Member");
        AddBookButton = new JButton(" Create New Book ");
        SearchBookButton = new JButton(" Search For Book ");
        SearchBook1Button = new JButton(" Search For Book");
        SearchMemberButton=new JButton(" Search For Member");
        SearchMember1Button=new JButton("Search For Member");
        ReturnDateButton= new JButton(" Return Book ");

```

```
BorrowButton = new JButton(" Borrow Book ");
CheckTransButton = new JButton(" Check Transaction ");
/*-----Setting layout for the panels used in our project*/
AddBookPanel.setLayout(new GridLayout(4,1));
AddBookPanel1.setLayout(new BorderLayout());
AddMemberPanel.setLayout(new GridLayout(5,1));
AddMemberPanel1.setLayout(new BorderLayout());
SearchBookPanel.setLayout(new GridLayout(5,1));
SearchBook1Panel.setLayout(new GridLayout(5,1));
SearchMemberPanel.setLayout(new GridLayout(5,1));
SearchMember1Panel.setLayout(new GridLayout(5,1));
ReturnPanel.setLayout(new GridLayout(2,1));
ReturnPanel1.setLayout(new BorderLayout());
BorrowPanel.setLayout(new GridLayout(3,2));
BorrowPanel1.setLayout(new BorderLayout());
TransactionPanel.setLayout(new GridLayout(4,1));
/*-----Member panel-----*/
AddMemberPanel.add(AddMemberNameLabel);
AddMemberPanel.add(AddMemberNameField);
AddMemberPanel.add(AddAgeLabel);
AddMemberPanel.add(AddAgeField);
AddMemberPanel.add(AddIICLabel);
AddMemberPanel.add(AddIICField);
AddMemberPanel.add(AddAddressLabel);
AddMemberPanel.add(AddAddressField);
AddMemberPanel.add(AddMemberTelLabel);
AddMemberPanel.add(AddMemberTelField);
/*-----*/
AddBookPanel.add(AddBookNameLabel);
AddBookPanel.add(AddBookNameField);
AddBookPanel.add(AddBookAuthorLabel);
AddBookPanel.add(AddBookAuthorField);
AddBookPanel.add(AddBookPublisherLabel);
AddBookPanel.add(AddBookPublisherField);
AddBookPanel.add(AddBookIDLabel);
AddBookPanel.add(AddBookIDField);
/*-----*/
SearchBookPanel.add(SearchBookIDLabel);
SearchBookPanel.add(SearchBookIDField);
SearchBookPanel.add(SearchBookButton);
SearchBookButton.addActionListener(new
                                SearchBookButtonListener());
/*-----*/
SearchBook1Panel.add(SearchBook1IDLabel);
SearchBook1Panel.add(SearchBook1IDField);
SearchBook1Panel.add(SearchBook1Button);
SearchBook1Button.addActionListener(new
                                SearchBook1ButtonListener());
/*-----*/
```



```

/*
    ReturnPanel1.add(new JLabel("Please enter the date
properly to calculate fines correctly.See Impotant Notes
for details."),BorderLayout.NORTH);
ReturnPanel1.add(ReturnPanel,BorderLayout.CENTER);
ReturnPanel1.add(ReturnDateButton,BorderLayout.SOUTH);
ReturnDateButton.addActionListener(new
                    ReturnDateButtonListener());
*/
JPanel design = new JPanel();
design.setLayout(new BorderLayout());
String design1 =
        "<html>" +
        "<p><b><i>Designed by:</i></b></p>" +
        "<p><b>Faculty:</b></p>" +
        "<ul>" +
        "<li>Prof. Hari Mohan Pandey Computer Engineering
Department, MPSTME, SVKM's NMIMS University</li>" +
        "</ul>";
JLabel dby = new JLabel(design1);
design.add(dby, BorderLayout.CENTER);
//%%%%%%%%%%%%%
JPanel timep = new JPanel();
Date td = new Date();
String tdate1 = DateFormat.getDateInstance().format(td);
JLabel timel1 = new JLabel("Today the date is "+tdate1+". And
now the time is ");
JLabel timel = new JLabel();
new TimerTime(timel);
timep.add(timel1);
timep.add(timel);
//%%%%%%%%%%%%%
JPanel help = new JPanel();
help.setLayout(new BorderLayout());
String helpl =
        "<html>" +
        "<p><b>Important Notes:</b></p>" +
        "<p><b>Below were the some important points that you need
to</b></p>" + "<p><b>keep in mind while using this system:
</b></p>" + "<ul>" + "<li>If the book was borrowed previous
month, for entering the return date please calculate and
enter the no of days that book have been with member.
</li>" + "<li>Ex: Borrowed date : 25th February
</li>" + "<li>Return date: 15th March </li>" + "<li>No of days in
february: 28-25 = 3 </li>" + "<li>No of days in march : 15
</li>" + "<li>Total no of days = 3+15 = 18 days </li>" +
        "</ul>";
JLabel h1 = new JLabel(helpl);
help.add(h1, BorderLayout.CENTER);
//%%%%%%%%%%%%%

```

```
tpane.addTab("Add Members",AddMemberPanel1);
tpane.addTab("Add Books",AddBookPanel1);
tpane.addTab("Search",searchhp);
tpane.addTab("Returning Books",ReturnPanel1);
tpane.addTab("Borrowing Books",BorrowPanel1);
tpane.addTab("Transaction Records",TransactionPanel);
tpane.addTab("Designed By",design);
tpane.addTab("Important Notes",help);
/*-----*/
Container mainpanel =getContentPane();
//getting contentpane and creating mainpanel
mainpanel.setLayout(new BorderLayout());
mainpanel.add(tpane,BorderLayout.CENTER);
mainpanel.add(timep,BorderLayout.SOUTH);
setSize(625, 325);
setVisible(true);
/*-----*/
}
class AddMemberButtonListener implements ActionListener
{
public void actionPerformed(ActionEvent ae)
{
    c.ControllerAddMember (AddMemberNameField.getText(), Integer.parseInt(AddAgeField.getText()),
    AddICField.getText(),AddAddressField.getText(),
    Integer.parseInt(AddMemberTelField.getText()));
}
}
class AddBookButtonListener implements ActionListener
{
    private int a;
    public void actionPerformed(ActionEvent ae) {
    a = Integer.parseInt(AddBookIDField.getText());
    c.ControllerAddBook (AddBookNameField.getText(),AddBook
    AuthorField.getText(),AddBookPublisherField.
    getText(),a);
    }
}
class SearchBookButtonListener implements ActionListener {
    private int a;
    public void actionPerformed(ActionEvent ae) {
    a = Integer.parseInt(SearchBookIDField.getText());
    c.ControllerBookIDRetrieve(a);
    }
}
//-----
class SearchBook1ButtonListener implements ActionListener {
public void actionPerformed(ActionEvent ae) {
c.ControllerBookNameRetrieve(SearchBook1IDField.getText());
}
```

```

}
//-----
class SearchMemberListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        c.ControllerICRetrieve(SearchMemberIDField.getText());
    }
}
//*****
class SearchMember1Listener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        c.ControllerNameRetrieve(SearchMember1IDField.getText());
    }
}
//*****
class ReturnDateButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        c.ReturnBook(Integer.parseInt(ReturnBookIDField.getText()),
                     Integer.parseInt(ReturnDateField.getText()));
    }
}
class BorrowButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        c.BorrowBook(BorrowMemberField.getText(),
                     Integer.parseInt(BorrowBookIDField.getText()),
                     Integer.parseInt(BorrowDateField.getText()));
    }
}
class CheckTransButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        c.checkTrans(TransMemberField.getText());
    }
}
//-----
//-----
}

```

/*FILE: LIBRARY.JAVA*/

```

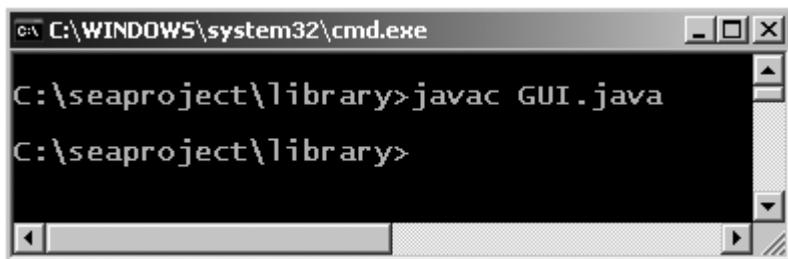
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class Library
{
    public static void main (String[] args)
    {
        GUI library = new GUI();
    }
}

```

Implementation:

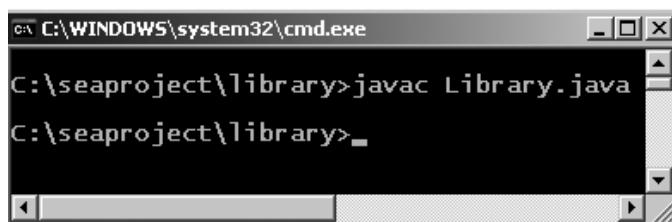
Step 1: First, compile all the files using javac.

Step 2: After compiling all the files, compile GUI.java as



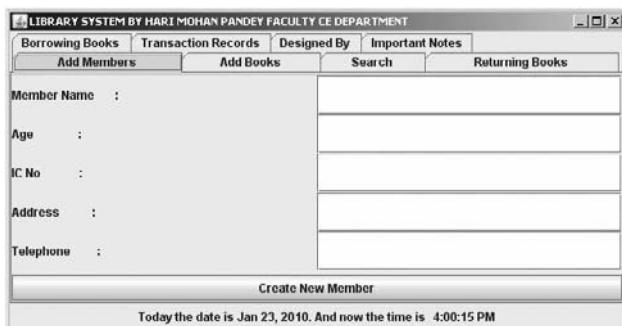
```
C:\> C:\WINDOWS\system32\cmd.exe
C:\seaproject\library>javac GUI.java
C:\seaproject\library>
```

Step 3: Then compile the file Library.java

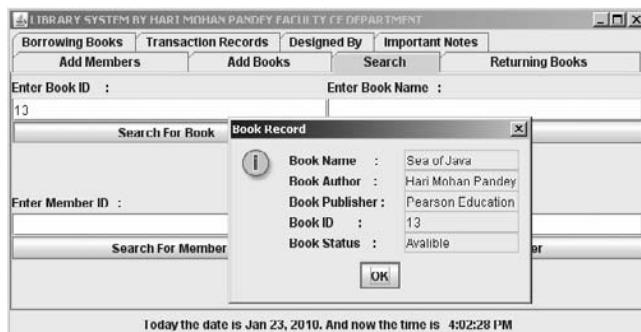


```
C:\> C:\WINDOWS\system32\cmd.exe
C:\seaproject\library>javac Library.java
C:\seaproject\library>
```

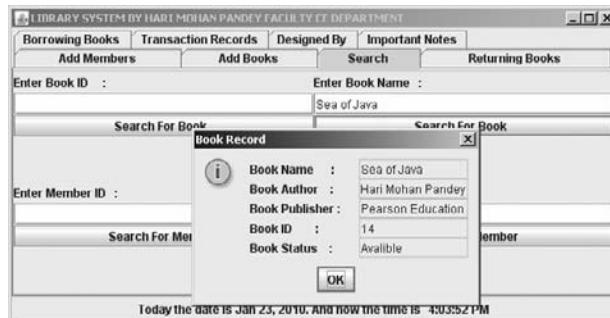
Step 4: Now interpret Library.java using Java interpreter as shown ahead:



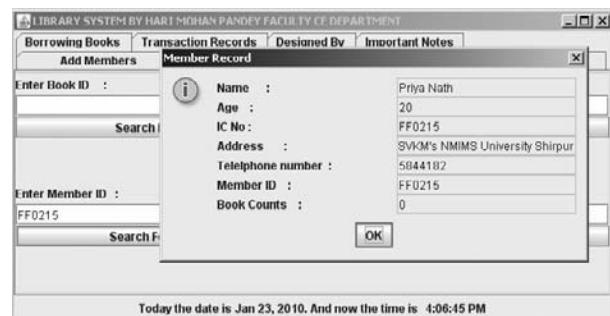
Screen shot for project at the initial stage



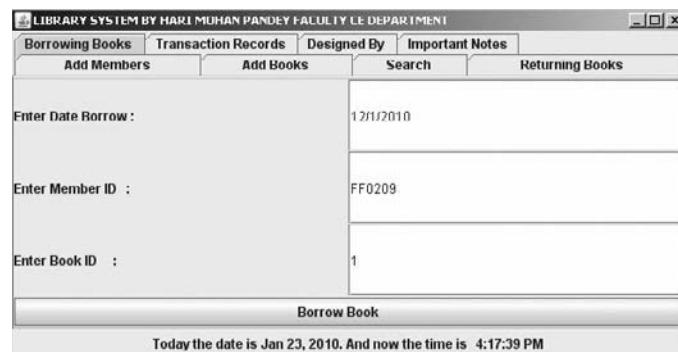
Screen shot of searching a book using book ID



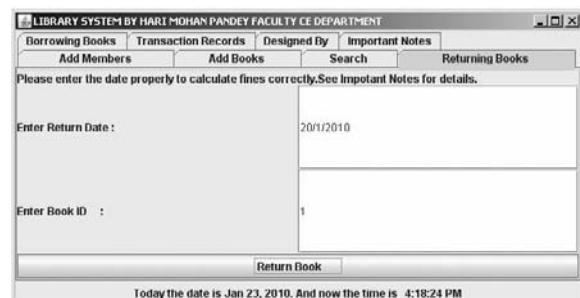
Searching book using book title



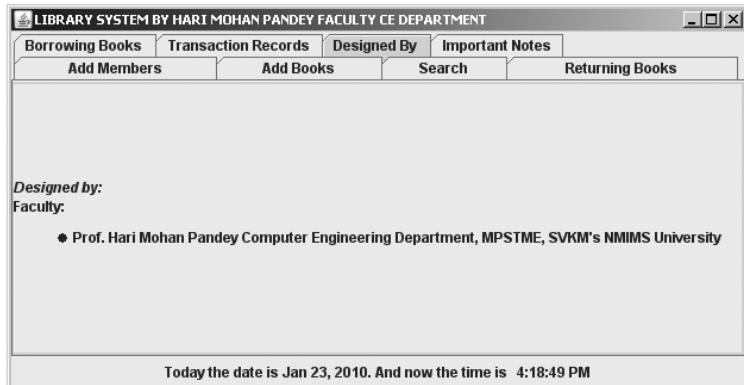
Member records



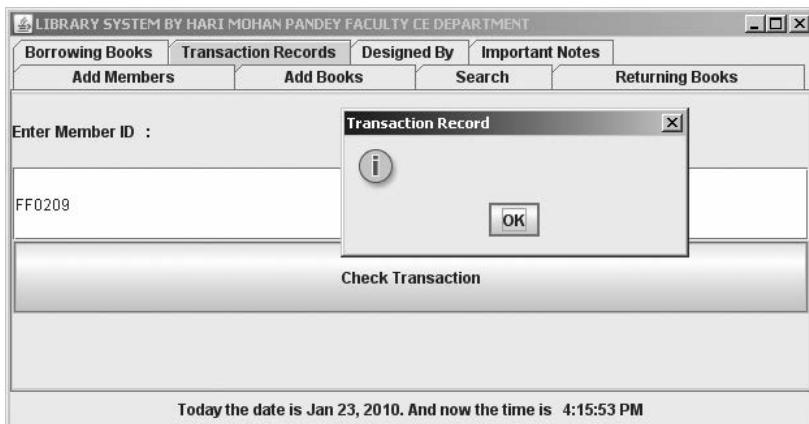
Implementation Borrowing Books option



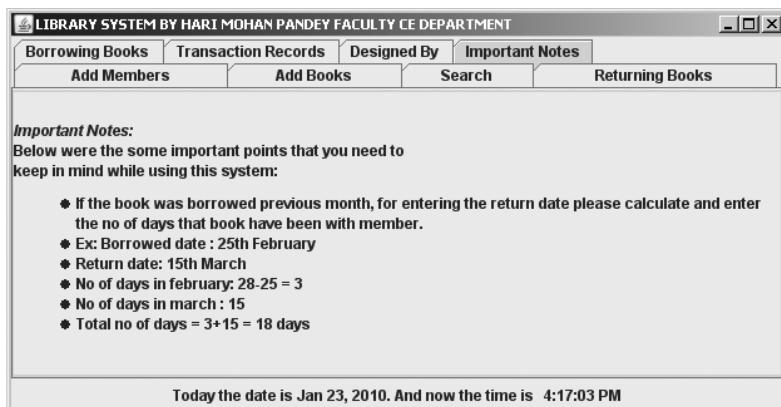
Implementation of Returning Books option



Developer details



Implementation of Transaction Records



Implementation of Important Notes option

A1

Java Keyword Reference

| Keyword | Description |
|----------|--|
| abstract | Used to modify a class or a method. |
| boolean | It is a Java primitive type. A boolean variable may take on one of the values true or false. |
| break | Used to prematurely exit a loop or to mark the end of a case block in a switch statement. |
| byte | It is a Java primitive type. A byte can store an integer value in the range [128, 127]. |
| case | Used to label each branch in a switch statement. |
| catch | Used to define exception handling blocks in try? catch? finally statements. |
| char | It is a Java primitive type. It can store unsigned single unicode character. |
| class | Used to declare a new Java class, which is a collection of related variables and/or methods. |
| continue | Used to the next iteration of a for, while, or do loop. |
| default | Used to label the default branch in a switch statement. |
| do | Specifies a loop whose condition is checked at the end of each iteration. |
| double | It is a Java primitive type. A double variable may store a double precision floating point value. Since floating point data types are approximations of real numbers, you should generally never compare floating point numbers for quality. |
| else | Used in association with the if keyword in an if-else statement. |
| extends | Used in a class or interface declaration to indicate that the class or interface being declared is a subclass of the class or interface whose name follows the 'extends' keyword. |
| false | Represents one of the two legal values for a boolean variable. |
| final | May be applied to a class, indicating the class may not be extended (subclassed). |
| finally | Used to define a block that is always executed in a try? catch? finally statement. |

| | |
|-------------------------|--|
| <code>float</code> | It is a Java primitive type. To specify a single? Perception literal value, follow the number with f or F, as in 0.01f. |
| <code>for</code> | Specifies a loop whose condition is checked before each iteration. |
| <code>if</code> | Indicates conditional execution of a block. |
| <code>implements</code> | Used in a class declaration to indicate that the class declared provides implementations for all methods declared in the interface whose name follows the implements keyword. |
| <code>import</code> | Makes one class or all classes in a package visible in the current Java source file. |
| <code>instanceof</code> | Used to determine the class of an object. |
| <code>int</code> | It is a Java primitive type. An int variable may store a 32-bit integer value. The integer class is a wrapper class for the int primitive type. |
| <code>interface</code> | Used to declare a new Java interface, which is a collection of methods. |
| <code>long</code> | It is a Java primitive type. A long variable may store a 64-bit signed integer. |
| <code>Native</code> | May be applied to a method to indicate that the method is implemented in a language other than Java. |
| <code>new</code> | Used to create a new instance of a class. |
| <code>null</code> | It is a Java reserved word representing no value. |
| <code>package</code> | Specifies the Java package in which the classes declared in a Java source file reside. |
| <code>private</code> | It is an access control modifier that may be applied to a class, a method or a field (a variable declared in a class). It specifies that the member can only be accessed in its own class. |
| <code>protected</code> | It is an access control modifier that may be applied to a class, a method or a field (a variable declared in a class). It specifies that member can only be accessed within its own package, and in addition, by a subclass of its class in another package. |
| <code>public</code> | It is an access control modifier that may be applied to a class, a method or a field (a variable declared in a class). |
| <code>return</code> | Used to cause a method to return to the method that called it, passing a value that matches the return type of the returning method. |
| <code>short</code> | It is a Java primitive type. A short variable may store a 16-bit signed integer. |
| <code>static</code> | May be applied to an inner class (a class defined within another class), method or field (a member variable of a class). |
| <code>super</code> | Refers to the superclass of the class in which the keyword is used. super as a stand-alone statement represents a call to a constructor of the superclass. <code>super.<methodName>()</code> represents a call to a method of the superclass. |
| <code>switch</code> | Used to select execution of one of multiple code blocks based on an expression. |

| | |
|--------------|--|
| synchronized | May be applied to a method or statement block and provides protection for critical sections that should only be executed by one thread at a time. |
| this | Refers to the current instance. The ‘this’ keyword is used to refer to the current instance when a reference may be ambiguous. |
| throw | Used to raise an exception. Any method that throws an exception that is not a RuntimeException must also declare the exceptions it throws using a throws modifier on the method declaration. |
| throws | May be applied to a method to indicate the method raises particular types of exceptions. |
| transient | May be applied to member variables of a class to indicate that the member variable should not be serialized when the containing class instance is serialized. |
| try | Used to enclose blocks of statements that might throw exceptions. |
| true | Represents one of the two legal values for a boolean variable. |
| void | Represents a null type. It may be used as the return type of a method to indicate the method does not return a value. |
| volatile | Used to indicate a member variable that may be modified asynchronously by more than one thread. |
| while | Specifies a loop that is repeated as long as a condition is true. |

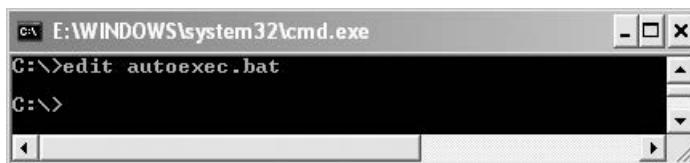
Creating Java Executables

A2

A2.1 EXECUTABLE JAVA ARCHIVES

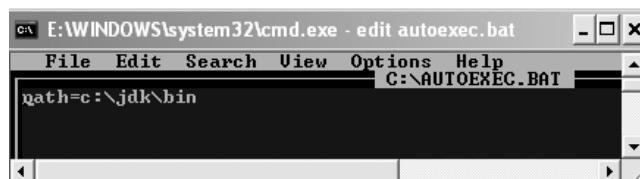
In order to compile and run the Java program, there must be a Java programming environment. You must install Java Development Tool Kit on your machine. There are various versions of Java presently available in the market. The most recent version is JDK 1.6 (also known as Java 6). The various components of the Java Development Kit have been discussed in detail throughout the book. After installing the Java programming environment, the next step is to develop an application in Java. No one can run the Java application directly. In order to run it on windows environment the program should be first compiled using “javac”.

For using the “javac” the class path is set first as shown below using c:\edit autoexec.bat command.



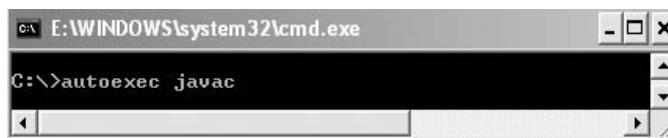
```
C:\ E:\WINDOWS\system32\cmd.exe
C:\>edit autoexec.bat
C:\>
```

Then the statement PATH=C:\jdk\bin is added.



```
File Edit Search View Options Help C:\AUTOEXEC.BAT
path=c:\jdk\bin
```

Now, after saving the above statement, switch from editor to command prompt and type c:\autoexec javac and press **Enter** key.



```
C:\ E:\WINDOWS\system32\cmd.exe
C:\>autoexec javac
C:\>
```

Upon pressing **Enter** key, path PATH=C:\jdk\bin is obtained on the command prompt as shown below.



```
C:\ E:\WINDOWS\system32\cmd.exe
C:\>autoexec javac
C:\>path=c:\jdk\bin
C:\>
```

Then write javac as: c :\javac as shown below and press Enter key to execute all the class files.

Java applications are generally run using the Java interpreter which is nothing but “java.exe”, from JDK. The Java interpreter is platform specific. Basically “java.exe” accepts the platform independent code (bytecode) available in “.class” generated by compiling the Java application using “javac”.

Now, to understand the concept in a better way consider the following example:

```
/*File: JPS1.java (PROG CLOSING WINDOW EVENT WITH FRAME) */

import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    MyFrame(String title)
    {
        super(title);
        Winadp wa = new Winadp(this);
        addWindowListener(wa);
    }
}
class Winadp extends WindowAdapter
{
    MyFrame mf;
    public Winadp(MyFrame f)
    {
        mf = f;
    }
    public void windowClosing(WindowEvent we)
    {
        mf.setVisible(false);
        System.exit(0);
    }
}
class JPS1{
    public static void main(String args[])
    {
        Frame fr = new MyFrame("Frame Demo");
        fr.setVisible(true);
        fr.setSize(300, 300);
    }
}
```

After developing the application program, the next step is to compile the program (JPS1.java) using “javac” as shown below:

```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Hari Mohan Pandey>cd\
C:\>cd jps2
C:\JPS2>javac JPS1.java
C:\JPS2>
```

Program compiled successfully

Now, the following jar tools are used:

- The Java archive tool
- Manifest file
- Main class attributes
- Creating executable Jar
- Options (c, m, and f)

To know how to use the above-mentioned tools to make Java application executable, each of them is discussed in detail.

- *The Java archive tool:* JAR (Java Archive) is a platform-independent file format that aggregates many files into one. The most common uses of JAR files are lossless data compression, archiving, decompression and archive unpacking. The general syntax of creating a jar file is:

```
jar options files
```

The three types of input files for the jar tool are:

- (a) Manifest file (optional)
- (b) Destination jar file
- (c) Files to be archived

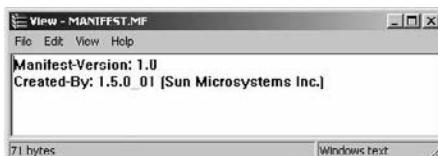
A manifest file is automatically generated by the jar utility. In order to specify a manifest file for the new jar archive, it can be specified using the -m option.

- *Creating Manifest file:* When a JAR file is created, it automatically receives a default manifest file. There can be only one manifest file in an archive, and it always has the pathname

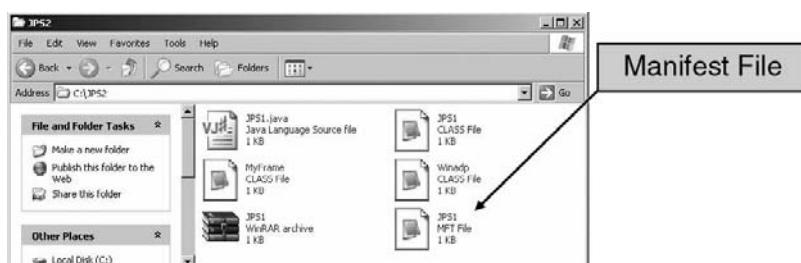
```
META-INF/MANIFEST.MF.
```

When a JAR file is created, the default manifest file simply contains the following:

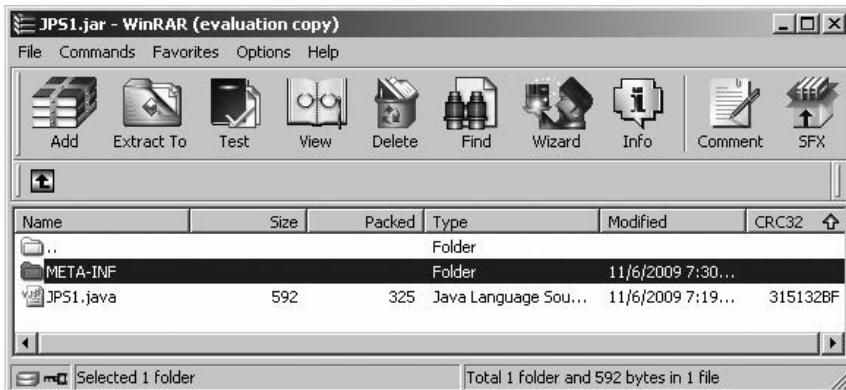
```
Manifest-Version: 1.0
Created-By: 1.5.0 (Sun Microsystems Inc.)
```



A view of manifest file



Windows for Java application contains MFT file



Windows for JPS1.jar (Evaluation copy)

For creating a manifest file, the procedure is open to any editor like notepad or wordpad and types the following:

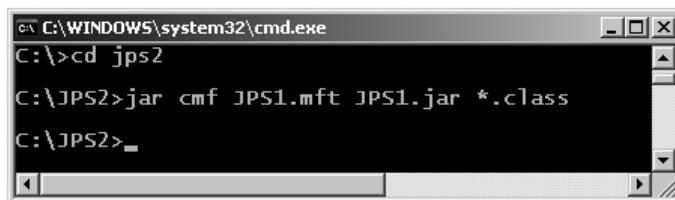
```
Main-Class: JPS1
```

A manifest file that is used must end with a new line. The last line of a manifest file will not be parsed, if it does not end with a new line character.

- *The main-class attribute:* This attribute is used by stand-alone applications that are bundled into executable jar files, which can be invoked by the Java runtime directly. The value of this attribute defines the relative path of the main application class that the launcher will load at start-up time. The value must not have the .class extension appended to the class name.
- *Creating executable JAR file:* For creating the executable jar file, the statement is used as shown below:

```
C:\JPS2>jar cmf JPS1.mft JPS1.jar *.class
```

This is also shown in the figure below:



Creation of executable jar for JPS1

➤ *Options*

- c** - Creates a new or empty archive on the standard output.
- f** - Specifies a jar file to process. In the case of creation, this refers to the name of the jar file to be created.
- m** - Includes manifest information from specified pre-existing manifest file.

This section has shown how to create executable Java archives. The next section describes how to invoke the DOS command interpreter.

A2.2 HOW TO INVOKE DOS COMMAND USING C/C++

This section deals with the methodology used to invoke the DOS command interpreter from C/C++ program. Now, consider the code given in the file “TEST.cpp”.

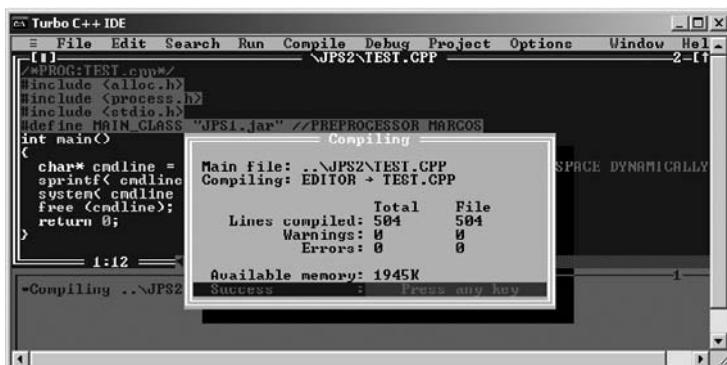
```
/*PROG:TEST.cpp*/
#include <alloc.h>
#include <process.h>
#include <stdio.h>
#define MAIN_CLASS "JPS1.jar" //PREPROCESSOR MACROS
int main()
{
    char* cmdline = (char*)malloc(1024); //ALLOCATION OF
                                         //SPACE
    DYNAMICALLY
    sprintf( cmdline, "java.exe -jar %s", MAIN_CLASS);
    system( cmdline );
    free (cmdline); //Releasing the occupied space
    return 0;
}
```

Explanation: Firstly, all the necessary header files like alloc.h, process.h and stdio.h for I/O are included. In the above code, #define is nothing but the pre-processor macros; for example:

```
#define MAIN_CLASS "JPS1.jar" //PREPROCESSOR MACROS
```

In the program TEST.cpp block of bytes are allocated dynamically from the heap using malloc(). Since malloc() function lies in alloc.h, the header file alloc.h has been included in the program. In the next line of the program, sprintf() has been used. The sprintf() method is used to send formatted output to a string. The system() method is used to invoke the DOS command interpreter file from inside and executing C program, just for the execution of DOS commands, batch file (“.bat”), or other programs named by the string “command”. At the end of the program, free() is used for releasing the occupied space.

Compile the program “Test.cpp” by using C/C++ compiler.



After Compiling “TEST.cpp” using C/C++ compiler

Now, use the make command, available in compile option at the menu bar, to create the TEST.exe.

The screenshot shows the Turbo C++ IDE interface. The main window displays the source code for TEST.CPP, which includes #include directives for stdio.h, process.h, and alloc.h, and a main() function that prints "TEST.EXE is up to date." to the console. A progress dialog box titled "Making" indicates the status of the compilation. Below the main window, a status bar shows "1:12". At the bottom, a message bar says "Linking TEST.EXE: Success : Press any key".

After selecting the make command

It is important to note that the execution of this application (JPS1.java) “JPS1.jar” and “TEST.exe” must be in the same directory. Under windows, the JRE is installed in the following directory:

C:\Program File\java\jre\bin

Primarily, one must set a path to include “bin” directory. Open the windows explorer and double-click the “TEST. exe” file. Rolling on a console window is required and remains for the life of the Java program.

The Jar Tool

A3

A3.1 INTRODUCTION

JAR (Java Archive) is a platform-independent file format that aggregates many files into one. The JAR tool combines multiple files into a single JAR file. JAR is a general-purpose archiving and compression tool, based on ZIP and ZLIB compression format. However, JAR was designed mainly to facilitate the packaging of Java applets or applications into a single archive. A JAR file allows a programmer to efficiently deploy a set of classes and their associated resources. Multiple Java applets and their requisite components (.class files, images and sounds) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time.

The most common uses of JAR files are lossless data compression, archiving, decompression and archive unpacking. The general syntax of creating a JAR file is:

```
jar options files
```

The table given below lists common JAR files operations:

| Operation | Command |
|---|---|
| To create a JAR file | jar cf jar-file input-file(s) |
| To view the contents of a JAR file | jar tf jar-file |
| To extract the contents of a JAR file | jar xf jar-file |
| To extract specifies files from a JAR file | jar xf jar-file archived file(s) |
| To run an application packaged as a JAR file
(requires the Main-class manifest header) | java – jar app.jar |
| To invoke an applet packaged as a JAR file | <applet code = AppletClassName.class archive = “JarFileName.jar” width = width height = height> </applet> |

Common JAR files operations

The next table lists various JAR command options:

| Option | Description |
|--------|------------------------------|
| c | Creates a new archive file |
| u | Updates an existing JAR file |

| | |
|---|--|
| f | The first element in the file list is the name of the archive that is to be created or accessed. |
| x | Extracts files and directories from JAR file. |
| t | Lists the table of contents from JAR file. |
| i | Generates index information for the specified JAR file. |
| v | Generates verbose output to standard output. |
| m | The second element in the file list is the name as the external manifest file. |
| M | Does not create a manifest file entry |
| C | Changes directories during command execution |
| 0 | Stores without using ZIP compression. |

Various JAR command options

The basic format of the command for creating a JAR file is:

```
jar cf jar-file input-file(s)
```

The c option indicates that programmer wants to create a JAR file. The f option indicates that the programmer wants the output to go to a file rather than to stdout. The name that can be assigned to the resulting JAR file is `jar-file`. Any file name can be used for JAR file. By convention, JAR filenames are given a `.jar` extension, though this is not required. The `input-file(s)` argument is a space separated list of one or more files that needs to be included in a JAR file. The `input-file(s)` argument can contain the wildcard `*` symbol. If none of the “input-files” are directories, the contents of those directories are added to the JAR archive recursively.

This command will generate a compressed JAR file and place it in the current directory. The command will also generate a default manifest file for the JAR archive.

A3.2 THE MANIFEST FILE

The manifest is a special file that can contain information about the files packaged in a JAR file. It is because of manifest file, JAR files support a wide range of functionality, including electronic signing, version control and package scaling.

When a JAR file is created, it automatically receives a default manifest file. There can be only one manifest file in an archive, and it always has the pathname

```
META-INF/MANIFEST.MF.
```

When a JAR file is created, the default manifest file simply contains the following:

```
Manifest-Version: 1.0
Created-By: 1.5.0 (Sun Microsystems Inc.)
```

These lines show that a manifest’s entries take the form of “header: value” pairs. The name of a header is separated from its value by a colon. The default manifest confirms to version 1.0 of the manifest specification. The manifest file is used in Java beans for storing information that which files in JAR file are Java beans.

A3.3 EXAMPLES OF JAR COMMAND OPTION

1. **jar cf abc.jar *.class *.gif**

The command creates a JAR file named abc.jar that contains all of the .class and .gif files in the current directory.

2. **jar cfm abc.jar man.mf *.class *.gif**

The command creates an abc.jar using manifest file man.mf.

3. **jar tf abc.jar**

The command lists the contents of jar file abc.jar.

4. **jar xf abc.jar**

The command extracts the contents of jar file abc.jar and places in the current directory.

5. **jar -uf abc.jar file1.class**

The command adds file1.class to the contents of abc.jar file, thus updating the JAR file.

6. **jar -uf abc.jar -C Xdir ***

The command adds all files below directory Xdir to JAR file abc.jar.

7. **jar uf abc.jar -C classes demo.class**

The command adds demo.class file from classes directory to abc.jar file.

8. **jar uf abc.jar -C classes . -C bin xyz.class**

The command adds all files in classes directory as well as file xyz.class from bin directory to the JAR file abc.jar. If classes file contains two files demo1 and demo2, the following command jar tf foo.jar will give the following output:

```
META-INF/META-INF/MANIFEST.MF bar1 bar2 xyz.class
```

9. **jar cvf bundle.jar ***

The command adds all files in the current directory to the JAR file bundle.jar. The v option displays the name of the file as it is added to the bundle.jar

10. **jar cvf bundle.jar audio classes images**

The command adds all files in the directories audio, classes and images to the file bundle.jar file.

11. To run a Java application class file that is stored within a JAR file the – jar option can be used. For that a JAR file is to be created that contains application class file. This creates a text file named Manifest.txt with the following contents:

```
Main-class: demo
```

The text file must end with a new line or carriage return and there must a space after colon. A JAR file named MyJar.jar is then created by entering the following command:

```
jar cfm MyJar.jar Manifest.txt demo.class
```

This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
```

```
Created -By: 1.5.0 (Sun Microsystems Inc.)
```

```
Main-Class: demo
```

When the JAR file is run with the following command, the main method of demo executes:

```
java -jar MyJar.jar
```

JAVA Index

Abstract class, 231–233
 insect, 235
Abstract data type (ADT), 157
Abstract windowing toolkit (AWT),
 405, 729
applet frame class, 450
button control, 459
checkbox class, 466
checkboxgroup class, 470–473
choice class, 480
component class, 372, 449
components, 406
container class, 449
controls, 458–459
events, 406
frame class, 450
label class, 464–466
list class, 473
panel class, 450
scrollbar class, 483
structure, 446–447
textfield class, 488–492
window class, 450
window hierarchy, 449
ActionEvent class, 407
Actionlistener interface, 414
Adapter classes, 428–429
AdjustmentEvent class, 408
Adjustmentlistener interface, 414
ALGOL, 2
Algorithm support, 581–586
Anonymous inner class, 441–444
Append method, 279
API, 44
 programming, 782
Applet class
 audio, 378–380

audioclip interface, 380
graphics class, 383
hierarchy of, 362
life cycle, 366–367
methods, 363
 vs. application programs,
 361–362
Applet tag and applet parameters,
 368–369
Appletcontext interface, 377–378
appletviewer applet1.
 java, 365
Arguments to methods, 203
Arithmetic operators, 50
ArithmeticeException, 293
Armstrong number, 95
ArrayList, 538–539
 class, 539
ASCII/unicode values, 86
Automatic memory management, 201
Backslash character constant, 14
Base class, 207
Binary operators, 49
Binding process
 dynamic binding, 219
 static binding, 219
Bipush statement, 796
Bitwise AND (&), 61
Bitwise operators, 61
Bitwise OR (!), 62–63
Bitwise XOR (^), 63
Boolean class, 635
Boolean variable, 56, 78, 389
Borderlayout, regions, 520
Break statement, 108–109
Built-in packages, 240–241

Bytecode, 35, 38
 implementation, 40–41
 instruction, 790
Bytecode array, values, 43
Byte stream, 351
 I/O classes, 351
Calendar class, 595–598
Call stack, 284
Capacity, 542
 method, 278
CardLayouts, 524
Cartesian coordinates, 394
Chained assignment, 59
Character class, 631–632
Character stream I/O classes, 346
CharArrayReader class, 347
CharAt method, 267
Client, 647–648
Clone method, 610–611
Collection class, 538
Comparator interface, 563–566
Classes and objects
 array, 174–177
 constructors, 181–182
 copying, 171–172
 passing and returning, 167–171
 private data assessment, 164–167
 programming examples, 160–164
 static class members, 177–178
 static member functions, 178
class.class_name, 157
Collection interface, 535
 hierarchy, 534
 iterating elements, 545
 listIterator
 interface, 545

- Command line arguments, 33–34
`command1.java`, 34
`compareTo` method, 267
`compareToIgnoreCase` method, 268
 Compilation process, 39
`ComponentEvent` class, 409
`ComponentListener` interface, 414
 Compound operator, 58
`Concat` method, 268
`Container`, 409
`ContainerAdapter` class, 414
`ContainerListener` Interface, 414
 Continue statement, 110–111
 disadvantage, 111
 Control flow statements, 77
 Constructors, 181
 inheritance, 224–229
 with parameters, 182–183
 Coordinate system in Java, 364
 Copy constructor, 195–196
 CPU cycles, 405

 Daemon threads, 331–332
 Datagram, 675–680
 Data members, 158
`DataInputStream` class, 31
`Date` class, 592–595
 Dialogs box, 506–510
`Dictionary` class, 573
 Domain Name Service (DNS), 641
 Double buffering, 709–712
 built-in, 729
 Data types
 default values, 159
 range, 16–17
 Decision-making statements, 77
 Decision structure, 1
 implementation, 2
 Default constructor, 183
 Decrement (++) operator, 59–61
 Delegation event model, 405–406
`delete` method, 278
`deleteCharAt` method, 278
 diagrammatical implementation, 269
 Dialogs, 746–753
 constants, 749
 file selection, 754–758
 do-while statement, 105–106

 Drawing lines and rectangles, 383–384
 Drawing ovals and circles, 384–385
`Drawstring` method, 399
 Dynamic method dispatch, 219–221

 else-if ladder, 83–84
`employeeName` method, 42
`EmptyStackException`, 294
`endsWith` method, 268
`ensureCapacity` method, 278
`Enumeration` interface, 566
 Environment variables, 20
 property, 608
`Equals` method, 268
`EqualsIgnoreCase` method, 268
 Event listeners, 407
 Event sources, 407
 Exception-generated block, 286
 Exception handling
 basis, 285–286
 checked vs. unchecked, 287–288
 classes, 286–287
 defined, 284
 exception handler, 285
 hierarchy, 287
 mechanism, 286
 runtime, 287
 Explicit type conversion, 70–71
 Expressions, types, 50

 Fibonacci series, 144
`File` class method, 337–341
`FileDialog` class, 511–515
`FileFilter` interface, 344
`FileInputStream` class
 methods, 355
`FileOutputStream` class
 methods, 355
`FileReader` class, 347
`FileWriter` class, 347
 Float and double class, 627–631
`FlowLayout`, 518–519
`FocusAdaptor` object, 409
`FocusEvent` class, 409
`FocusListener` interface, 415
`Font` class methods, 398
`Font` class properties, 398
`Fontmetrics` class, 400
`Fontmetrics` object, 400
 Fonts
 dialog, 397
 dialoginput, 397

 monospaced, 397
 sansserif", 397
 serif", 397
 For loop, 96–97
 different syntaxes, 97–98
 examples, 98
 nesting, 100–101
 Frame design, 41

 Garbage collected heap, 37, 792
 Garbage collection, 201
`getBytes` method, 271
`getChars` method, 268–269
 implementation, 270
`getClass` method, 616
`Getcodebase()` methods, 375–376
`Getdocumentbase()`, 375
`getName` method, 616
`getParameter` method, 371
`Getstatic`, 791, 804
`GridBagLayout`, 518, 524
`GridLayout` layout, 522

 Handling internet addresses, 642
 Handling keyboard events, 424–428
 Handling mouse events, 417
`HashMap`, 555
`HashSet` classes, 550–551
 constructor, 557
`HashTable` class, 573
 Hello World' program, 685
 Hierarchical inheritance, 209–210
 revisited, 221–223
 Historical collection, 566
 HSB color system, 386
`HTML`, 361, 729

 if-dependent statements, 79
 if-else statement, 78
 nesting, 81–83
`if_icmpge` instruction, 804
`if` statement, 77
`iinc` instruction, 805
 Illegal AccessException, 297–298
`Iload` instruction, 803
`Image`, 702–704
`ImageConsumer` interface, 723
 Image data, 716–722
 consuming, 723
`Imageicon` class, 730
`Imageobserver` interface, 705

- ImageProducer interfaces, 723
Increment (++), 59–61
InetAddress class, 642
Inheritance
 derived class, 207
 hierarchical inheritance, 209–210
 hybrid, 210
 implementation, 207
 multilevel, 208–209
 multiple, 209
 programming examples, 211–218
 single level, 208
 types, 207–208
 workflow, 6
Inner classes in event handling, 439–441
Inner classes in methods and scopes, 436–438
InputEvent class, 410
Insert method, 279
Insets, 531–532
Instance of operator, 67–68
IOException, 297
Interface definition language (IDL), 816
Interfaces
 declaration, 252–253
 extending classes, 260–262
 implementation, 253
 programming examples, 254–260
Internet Domain sockets, 645
ItemEvent class, 410
ItemListener interface, 415
Iterator interface, methods, 545
J2SE development kit, 20
Japplet class, 730
Java 3D API, 815
Java
 array, 116–118
 benefits, 10
 character set, 12
 comments, 15–16
 data type, 16
 declaration, 116
 defined, 9, 140–142
 evolution, 9–10
 features, 10–12
 first program, 18–19
 function declaration, 140–142
 function with parameters, 146–149
 output screen, 19
 overloading, 151–154
 platform, 44–45
 programming examples, 23–24
 recursion, 149
 setting-up java, 19–20
 sockets, 645–646
 structure, 17–18
 syntax, 142–145
Java animation API, 815
Java API framework, 783
Java API package, 816–818
Java applet API, 813
Java application program, runtime memory area, 792
Java base API, 813
java.applet.Applet, 361
java.awt.FontMetrics, 400
Java base platform
 java API, 782
 java virtual machine, 782
Java bytescodes, 35, 785
Javac, 45
Java collaboration API, 815
Java commerce API, 816
Java compiler, 36, 45
Java development kit (JDK), 19–20, 45
Java enterprise API, 815
Java 1.1 event model, 406
 class, 406
Java interpreter, 19, 22, 791
Java 1.0 inheritance event model, 405
Java, internet and www, 44–45
java.io package, 336
Java's checked exceptions, 289
javac applet1.java, 363
Java's GUI components, 383
java.lang, 231
Java management API, 816
 components, 816
Java media API, 814–815
 java 2D API, 815
 java media framework (JMF) API, 815
Java media capture, 815
Java media conference, 815
Java media player, 815
Java native interface (JNI)
 example, 685
 JNI-style header file, 687–688
 limitations, 688
 role, 684–685
Java.net package, 642
Java program on notepad, 22
Java remote method invocation (RMI), 695–696
Java security API, 814
Java server APIs, 186
Java speech, 815
Java stack frame, 790
Java standard extension API, 783, 813–814
Java telephony API, 815
Java tokens, 12
 constants/literal, 13–14
 identifier, 13
 keywords, 12–13
Java virtual machine (JVM), 9, 35, 37, 44, 783–785, 790, 792
 fundamental pieces, 784
 implementation, 36
 location, 35
 set of register, 790
 stack, 37, 41
Jbutton class, 731–733
Jcheckbox class, 734–736
Jcombobox class, 738–741
JDK's appletviewer, 366
Jlabel class, 730–731
Jprogressbar class, 763–767
JPS2.BC, 85–89
JPS3.BC, 794–795
JPS8.java, 39, 45
Jradiobutton class, 736–738
Jscrollpane class, 743–745
Jslider class, 767–771
Jsplitpane class, 745–746
JtabbedPane class, 741–743
Jtable class, 759–760
Jtextfield class, 733–734
Jtoolbar class, 759–760
Jtoolbar instance, 762
Jtree class, 771–774
Just-in-time compiler, 789
KeyEvent class, 410–411
KeyListener interface, 415
Labelled break, 111
Layout and layout manager, 518
Left shift operator (<<), 64–65
length() method, 267
LIFO stack, 42
Linkedlist class, 542
Linking process
 dynamic binding, 219.
 static binding, 219

- Listeners interfaces, 414
 List implementations, 539
List interface, 536–537
 Local applets, 361
 Logical AND, 56–57
 Logical Not (!), 58
 Logical operator, 56
 Logical OR, 57
 lookupswitch instruction, 809–810
 Loop/iteration structure, 1
 implementation, 2
- Magenta, 520
 MalformedURLException, 682
Maps, 554
 classes, 557
 Map.Entry Interface, 556
 Map interface, 554
 methods, 555
 Mathematical functions, 71–72
 Matrix multiplication, procedure
 for, 132
 Mediatracker class, 713–714
 Membership operator, 158
 MemoryImageSource, 724
 object, 722
 Menu and menubars
 examples, 774–779
 popup, 503–506
 shortcut key, 501
 Method area, 37
 Method overriding, 218
 mkdir method, 341
 Mnemonic instructions
 representation, 36
 Modular programming, 1
 Modifier
 private, public, 249
 protected, 249
 mousePressed method,
 391, 422
 mouseclicked method, 419
 MouseEvent Class, 411–412
 MouseListener interface,
 415–416
 MouseMotionListener
 interface, 416
 mouseReleased, 422
 Multithreading, 309
 MyMouseAdapter, 429
- Nested and inner classes, 431–432
 Newarray instruction, 800, 802
- NullPointerException, 791, 802
 Number class, 625
- Object-based languages, 7
 Object class, 610
 ObjectInputStream class, 692
 Object-oriented programming
 (OOP), 4, 9
 advantages, 6
 characteristics, 6
 class and object, 4–5
 data hiding, 5
 dynamic binding, 6
 encapsulation, 5
 inheritance, 5–6
 polymorphism, 5
 ObjectOutputStream interface, 689–690
 Object slicing process, 229–231
 Observable class, 601
 Observer interface, 601
 overriding imageUpdate, 706
 1-D array, 118–125
 examples, 119–122
 Operating system (OS), 783
 OutputStream classes, hierarchy, 352
 Overloading function, 151–154
- Packages, 44
 Packages and interfaces
 access protection table, 252
 types, 240–246
 Paint method, 372, 396
 Palindrome, 94
 PASCAL, 1
 Passing parameter, 369–370
 PixData, 724
 Polygon, defined, 385
 Popup menus, 503–506
 POW, 96
 Precedence of operator, 69–70
 PrintStream, 80
 Procedural programming language, 2
 Procedure-oriented programming
 characteristics, 2
 Process class, 621–624
 Program counter, 37
 Properties class, 576–577
 Programming methodology
 bottom-up, 2–3
 top-down approaches, 2–4
 Polymorphism, bifurcation, 5
 public final String
 getName(), 313
- public static Thread
 currentThread(), 313
 Public void paint, 790
- Quadratic equation, 85
- Random class, 599–601
 Reader class, 346
 Recursion programming, 149–151
 Relation and ternary operator, 53–56
 Remainder operator, 50
 Repaint method, 373–375
 Replace method, 269
 Return statement, 145–146
 Reverse method, 278
 RGB color system, 386
 Right shift operator (>>), 65–66
 Right Shift with Zero Fill (>>>),
 66–67
 RMI service, 696
 RTF, 729
 Running applet for applet1.java, 364
 Runtime class, 619–720
 Runtime memory area, 792
- Scanner class, methods, 32
 Scope and lifetime, 73–74
 Scrollbar class, 483
 parts, 483
 properties, 484
 Sequence structure, implementation, 1
 Serializable interface, 689
 Serialization, 689
 Server, 646–647
 Set interface, 537
 SetLength method, 279
 setXORMode, 388
 Shorthand assignment operator, 58
 ShowStatus method, 367
 Sipush statement, 796
 Sizeof () operator, 69
 Sleep method, 316
 Socket fundamentals, 645
 SortedSet interface, 538
 SortedMap interface, 556
 methods, 556
 Stack, 790
 after pop operation, 191
 after push operation, 190
 class, 571–572
 StackOverflowError, 791
 Static binding, 219
 Static class members, 177–178

- Static data members, 179–181
Static member functions, 178
Static nested class, 438–439
Static variables, 177
Stream, 336
 byte, 351
 character, 345–346
 programming Examples, 348–351
 reading and writing files, 353–354
 reading from console, 352–353
 types, 344–345
 writing to console, 353
String
 constructors for string class, 266–267
 methods of string class, 267–271
 programming examples, 272–276
 string class, 265–266
String buffer class, 276–277
 constructor of, 277
 methods, 277–280
 programming examples, 280–281
StringBuffer(int capacity), 277
StringBuffer(String str), 277
Stringtokenizer class, 590–592
Structured programming, 1
Super keyword, 223–224
Swing, components, 729
Switch-case statement, 86–89
System class, 606–607
System properties, 609
System.out.println, 80
System-dependent fonts, 397
tableswitch instruction, 807–808
Textarea class, 492–494
TextEvent class, 413
TextListener interface, 416
this reference, 196–201
Thread chat server and client, 672–675
Threads, 309–310
 communication, 324–325
 implementing thread using runnable, 318–319
 methods, 313–314
 priority, 320–321
 suspended and resuming threads, 327–328
 synchronization, 322–323
 synchronized statement, 324
Three-dimensional (3-D) array, 134–137
Three-line coding, 30
Throw keyword, 294
toString method, 271, 549
toUpperCase method, 271
Transpose matrix, 129
TreeMap Class, 559
Treeset classes, 551–554
trim method, 271
try and catch, 290–291
try block, 286
try-catch block
 nesting, 300–301
Two-dimensional (2-D) array, 125–126
 column-major implementation, 126
 diagonal element, 130–132
 row-major implementation, 126
Update method, 373
URL, 680
 constructors, 682
User-defined packages, 241–246
Unary operators, 49–50
ValueOf method, 271
Variable, 15
 length arrays, 1 34
Vector class, 566–567
 constructors, 567
 methods, 568
Virtual machine
 array, 800–801
 class instance, 796–798
 instruction set, 793
 invoking methods, 793–794
 switch statements, 805–806
 throwing and handling exceptions, 810–813
VirtualMachineError, 792
Virtual parts, 36
while loop, 89–93
while statement, 89
windowClosing method, 453
WindowEvent class, 413
WindowListener interface, 416–417
without try-catch, 289–290
Wrapper class, 624
Writer class, 347

XOR color
XOR mode, 387, 391