

# JAVA

ONE STEP AHEAD

ANITA SETH

*Assistant Professor  
Institute of Engineering and Technology  
DAVV University, Indore*

B.L. JUNEJA

*Formerly Professor  
Indian Institute of Technology, Delhi*

OXFORD  
UNIVERSITY PRESS

**OXFORD**  
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford. It furthers the University's objective of excellence in research, scholarship, and education by publishing worldwide. Oxford is a registered trade mark of Oxford University Press in the UK and in certain other countries.

Published in India by  
Oxford University Press  
YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India

© Oxford University Press 2017

The moral rights of the author/s have been asserted.

First published in 2017

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission in writing of Oxford University Press, or as expressly permitted by law, by licence, or under terms agreed with the appropriate reprographics rights organization. Enquiries concerning reproduction outside the scope of the above should be sent to the Rights Department, Oxford University Press, at the address above.

You must not circulate this work in any other form  
and you must impose this same condition on any acquirer.

ISBN-13: 978-0-19-945964-3  
ISBN-10: 0-19-945964-9

Typeset in Times New Roman  
by Cameo Corporate Services Limited, Chennai  
Printed in India by Magic International (P) Ltd, Greater Noida

Cover image: Damaratskaya Alena / Shutterstock

Third-party website addresses mentioned in this book are provided  
by Oxford University Press in good faith and for information only.  
Oxford University Press disclaims any responsibility for the material contained therein.

# Preface

The development of computer systems and programming languages now span more than half a century. During this period technology has developed tremendously both in hardware as well as software. Now even school children can afford individual laptops or desktops and use it for studies and for browsing the Internet. Computer languages have developed from unstructured languages to structured procedural languages such as C and from procedural languages to object-oriented computer languages such as C++ and Java, and from single threaded languages to the ones that support multi-threaded programming so that different parts of a large program can run concurrently on multiple processors. The language Java supports multi-thread programming and its compiled programs are independent of native operating system. Java is in tune with hardware development in which multiple processor CPUs are the order of the day.

Java is an important programming language in software development space right now. It has undergone many changes since its inception and that has led to its use in wide range of areas. From a simple web page to big scientific applications, from cell phones to supercomputers, Java is everywhere!

In view of the developments described above, it is not surprising that undergraduate courses at many universities include Java along with C and C++, and some are switching from C++ to Java. The need has long been felt for a book which makes the learning of Java easy and self-reliant.

## About the Book

The present book is primarily written for beginners and intermediate level readers, at the same time the topics are dealt with appropriate complexity often needed by professionals. The book is ideal for self-learning of Java and also useful for postgraduate students.

The language of the book is simple so that core ideas are easily understood and almost every topic is followed by illustrative programs so that a reader can easily grasp the programming technique. Besides, the topics on GUIs for which Java is so popular are dealt in depth so that the reader after learning from the book can easily take up professional assignments.

The book has following salient features:

- Covers Java SE8, which is the latest version of Java language.
- Provides line-by-line explanation of programs in every chapter to enable the students understand the logic used for writing an efficient program.
- Key notes and call outs are provided at appropriate places to highlight the important points. In addition to these, common programming errors are also provided at the end of each chapter.
- Covers cartoon figures to help the readers understand the concepts in a better way.
- Numerous programming examples and application programs are provided to help readers understand the implementation of concepts learnt.
- Covers several advanced topics such as Java Beans, Java Servlet, and JDBC, which are usually not found in other books.
- Provides enhanced coverage of Swing components such as JMenu, JScrollbar, JSlider, JScrollpane, Lambda expression for ActionListener, JCheckbox, JTooltip, etc.
- Includes a variety of end-chapter exercises for practice which have subjective as well as objective questions.
- Appendices cover interview questions, history on Java versions, and making webpages.

## Content and Organization of the Book

The book comprises 27 chapters and two appendices. Their details are presented below.

*Chapter 1* introduces the concept of classes and objects and object oriented programming (OOP). It also gives an overview of structure of Java language, and ends with description of the points of similarities and differences between Java and C++.

*Chapter 2* explains the essential structure of an application Java program with the help of simple programs. It explains in detail how to write, compile, and run the program using 'Free Java' compiler. The arithmetic operators are also introduced and illustrative computational programs are presented and explained.

*Chapter 3* describes in detail the primitive data types in Java and the range of values for each *type* of variable. It explains declaration of variables, the *type* casting and scope of variables. It also elaborates on the various methods encapsulated in Math class and illustrates use of these methods through several simple programs. It also describes in detail all the operators in Java language as well as their attributes.

*Chapter 4* deals with the different control statements which provide the choice to branch off such as `if` (expression), `if-else` expressions, and `switch` statement. It also takes up the looping expressions such as `while`, `do-while`, and `for` loops. The other jump conditions such as `break`, `break with label`, `continue`, and `continue with label` are also dealt in detail with examples.

*Chapter 5* introduces the formal definitions of class and objects and the attributes of various classes are discussed in great detail.

*Chapter 6* starts with the definition of methods in a class and their applications. The chapter also covers the topics such as Lambda functions and their applications, assigning method references, predicates, and method chains.

*Chapter 7* deals with topics on single-dimensional and multi-dimensional arrays. It discusses in detail the operations involving two-dimensional arrays and three-dimensional arrays. The solution of linear algebraic equations is explained using illustrative programs.

*Chapter 8* covers in detail the process of inheritance of classes and implementation of interfaces. The different types of inheritances are discussed. The problem of access control, use of keyword `super`, method overriding, and dynamic method dispatch are also explained. For all these topics, illustrative programs are provided.

*Chapter 9* presents the definition and characteristics of interfaces. Various types and methods of interfaces are explained with the help of illustrative programs.

*Chapter 10* discusses the packages in Java and important classes in `java.lang` package. It also includes auto-boxing and auto-unboxing, important classes of `java.util`, details of scanner class, and newly created `java.time` package for date and time.

*Chapter 11* deals with how exceptions arise and how to handle them. It explains how a programmer can create own exception class and also deals with nested `try` and `catch` blocks.

*Chapter 12* describes the string class and its methods and objects. The class `StringBuffer` is discussed along with its constructors and methods. The methods of `StringBuilder` class are also explained and illustrated by programs.

*Chapter 13* introduces the concept of threads and their benefits, and class and methods of creating threads. The deadlock situations are analyzed and different states of threads are discussed with examples.

*Chapter 14* deals with generics and types. Generic class with one type and multi-type parameters along with generic method overloading and generic interface are discussed in the chapter.

*Chapter 15* discusses the basic concepts about an image in Java and the various mechanisms of manipulations of images supported in Java. It explains the various image file formats and the process of loading and displaying an image in Java.

*Chapter 16* deals with the concept of collections and its types and properties along with numerous programming examples.

*Chapter 17* deals with input and output streams. The file class and various other classes and their subclasses related to file handling are explained using programs.

*Chapter 18* deals with applet programming including its classes and methods. The life cycle of an applet and HTML tags and how to create a web page are explained. The process of animation is explained with supporting programming examples.

*Chapter 19* explains the generation of events and how they are handled by using delegation event model. It also discusses the hierarchy of event handling classes, event types, sources of events and event listener interfaces and classes. The important event classes are covered in detail.

*Chapter 20* discusses the creation of AWT windows and adding graphical components to it. Different classes, methods, and layouts are explained with suitable examples.

*Chapter 21* deals with drawing different shapes in an AWT window. The methods of drawing of strings, straight lines, rectangles, polygons, ellipses, and arcs are explained with the help of illustrative programs.

*Chapter 22* discusses the swing packages and hierarchy of swing classes are presented. The different window panes in swing windows in applet as well as in `JFrame` objects are explained. The description of swing components, methods of adding components, and different border classes are also discussed and illustrative programs are given.

*Chapter 23* explains why a single thread is used in swing GUI and various classes and components are discussed and several application programs are given. Programming with multiple panels is illustrated for generating complex window patterns.

*Chapter 24* describes the internet address systems IPv4 and IPv6. The commonly used classes such as URL, URLConnection, HttpURLConnection, Sockets, ServerSocket, and classes in UDP are aptly discussed. A number of programs are given to demonstrate the implementation of the topics discussed in the chapter.

*Chapter 25* covers the fundamentals of *Java beans*. The basic processes involved in bean building are also explained through live projects. The process of building various types of beans has been explained using program illustrations.

*Chapter 26* explains the basic concepts related to server-side programming. The basics of servlet architecture, servlet API, and the step-by-step procedure for writing servlet programs are also covered.

*Chapter 27* introduces the concepts relating to *Java database connectivity*. It includes the basics of JDBC API and procedure for making connections of Java application with database system. JDBC transaction management and batch processing are also explained using illustrative programs.

*Appendix A* lists important additions in different versions of Java and also includes the latest enhancements in Java SE8. *Appendix B* introduces the concept and some techniques in HTML for preparing webpages. Interview questions are also provided at the end of the book.

## Online Resources

The companion website of the book offers the following online resources for instructors and students:

### *For Faculty*

- PowerPoint Slides
- Lab exercises
- A chapter on remote method invocation (RMI)

### *For Students*

- Additional projects
- Quizzes
- List of packages in Java
- Debugging exercises with answers
- Model question papers with answers
- Unicode system

## Acknowledgements

We are grateful to the Director of IIT Delhi and the Director of Institute of Engineering and Technology (IET), Indore for providing us facilities while writing this book. We would also like to thank the Head of Mechanical Engineering Department, IIT Delhi and the Head of Department of Electronics and Communications, IET, Indore for providing departmental facilities, advice, encouragement, and inspiration.

We are thankful to Prof. G. Singh of Maryland University for always being a source of inspiration and encouragement, and Prof. P.V.M. Rao, Department of Mechanical Engineering, IIT Delhi, for his time, advice, and encouragement. We are indebted to our colleagues at Institute of Engineering and Technology, Indore for providing their support and motivation. We thank Dr Vaibhav Neema for providing valuable ideas and suggestions. The books, publications, and video lectures of various professors on the subject also inspired us to write this book. The ideas given by Nikita Seth cannot be overlooked.

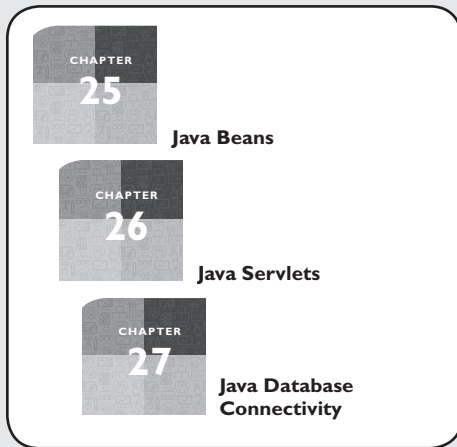
We express our gratitude to the following reviewers for giving their invaluable feedback and suggestions. Many of these suggestions have been incorporated that has greatly enhanced the content.

- Prof. Mahesh Raghunath Shirole, VJTI, Mumbai
- Prof. Sachin Gupta, Dronacharya Group, Greater Noida
- Prof. Sonjay Kumar Saha, Jadavpur University, Kolkata
- Prof. Meenu Khurana, Chitkara University, Punjab
- Prof. Charu Virmani, Manav Rachna International University, Faridabad
- Prof. C.M. Chidambaranathan, SRM University, Sonipat
- Prof. K. Gopala Reddy, Ramachandra Engineering College, Andhra Pradesh
- Dr Nekuri Naveen, Sasi Institute of Technology and Engineering, Andhra Pradesh
- Dr Nagender Kumar S., Geethanjali College of Engineering Technology, Hyderabad

We would like to thank the editorial team at Oxford University Press, India for their excellent support and suggestions, and the entire team for all the wonderful coordination and assistance they provided us.

**Anita Seth  
B. L. Juneja**

# Features of

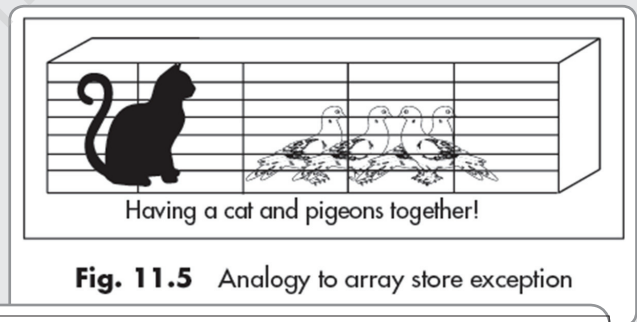


## Advanced Topics

Dedicated chapters on advanced topics such as JDBC, Servlets, and Java Beans, delve into the networking and distributed applications of Java.

## Cartoons, Notes, and Callouts

Cartoons, notes, and callouts are given along with relevant concepts and programs throughout the text. *Cartoons* help in grasping a complex concept easily through simplified figures. *Notes* and *callouts* provide additional and important information and help in avoiding mistakes.



The default order in priority queue is the ascending order, that is, the smallest value is at the head of the queue and the largest value at the tail end.

```
type Method_identifier(type parameters)throws Exceptions
```

Name of method throwing Exception

Parameters of method

throws clause

# the Book

## Programming Examples

Numerous examples along with separate section on application programs (in relevant chapters) are provided to understand the working of the concepts discussed in the book and to get an idea on the practical applications of the subject.

### Program 15.12: Illustration of image crop filter

```
1 import java.awt.Toolkit;
2 import java.awt.Graphics;
3 import java.awt.Image;
4 import java.awt.image.CropImageFilter;
5 import java.awt.image.FilteredImageSource;
6 import java.applet.Applet;
```

Program 20.15 presents the code for online voting for choosing a suitable tion in XYZ organization.

### Program 20.15: Illustration of online voting system

```
1 package onlinevoting;
2 import java.awt.*;
```

## Common Programming Errors and Tips

1. *Capitalization of key words*—it is the most common error that programmers often com Writing keywords such as `class` and `int` with a letter beginning in upper case would ca an error message that depends on which keyword was capitalized.
2. Specifying return type to constructor may give errors as it cannot specify a return type.
3. Sometimes, class methods may not be written in the required format. Class methods ha

```
ClassName.MethodName(Argument(s))
```

4. A common error is to forget the class name. In that case, an error message is generated.
5. When the class is defined, each argument should be prefixed with the type. For instance

```
public void demo(int a, int b, double c)
```

## Common Programming Errors and Tips

They are provided at the end of the relevant chapters and list important tips and common errors that readers tend to make while writing a program.

## Chapter-end Exercises

Includes wide range of questions, such as MCQs, review exercise, programming exercises (of varying levels), debugging exercises, and mini projects to help readers self-check and apply the learnt concepts in their own programs.

### Programming Exercises

1. Write a program in which four buttons are incorporated. The buttons can be named B1, B2, B3, and B4. GridLayout with  $2 \times 2$  grid is used.
2. Write a program using GridBagLayout that contains the components TextField for entering name and address details; further, include submit and cancel
7. Write a program in which a user can choose a mode of transport for daily purpose inclu scooter, bus, cycle using checkboxes in window on selecting a particular mode, it should disp have chosen.....for daily transportation purp
8. Write a program in which list of items for man

### Debugging Exercises

1. Find the errors in the following program; debug it and run the program.

```
class Operator
{public static void main (String args)
{
double x = 5.5, y = 10.5, p = 4.0;
int n = 40, m = 50.2;
```
3. Find the errors in the following program; run the program.

```
class Typecast2
{public static void main (String
{
Char B = 7, Q = '6';
int A = 4, D, E;
```

### Mini Project

There is a retail organization ABC that maintains the customer records regarding name, phone number, amount of purchase made, and the total amount of purchase made. Write a program in which the user is required to enter the details regarding customer id, first name, and last name. On pressing the submit button, it should search through the

records. If the record is found, it should about the total purchase and points ea chase is of ₹500, then 20 points are earn On every subsequent purchase of ₹500, earned. If the customer earns a total o discount of ₹100 is provided on purcha



# Brief Contents

*Preface* iii

*Features of the Book* viii

*Detailed Contents* xi

1. Object-oriented Programming and Evolution of Java	1
2. Program Structure in Java	25
3. Data Types, Variables, and Operators	58
4. Control Statements	112
5. Classes and Objects	143
6. Methods	178
7. Arrays	217
8. Inheritance	264
9. Interfaces	297
10. Packages and Java Library	336
11. Exception Handling	405
12. String Handling in Java	442
13. Multithreaded Programming	474
14. Generic Programming	519
15. Image Handling in Java	554
16. Collections	583
17. Input/Output Streams and File Operations	652
18. Applets	718
19. Event Handling	754
20. Creating GUIs in AWT Windows	801
21. Drawing in AWT Windows	839
22. Swing—Part 1	887
23. Swing—Part 2	931
24. Networking	977
25. Java Beans	1015
26. Java Servlets	1060
27. Java Database Connectivity	1102

*Appendix A Enhancements in Different Versions of Java* 1145

*Appendix B Creating Webpages* 1150

*Java Interview Questions* 1171

*Index* 1181

*About the Authors* 1191

# Detailed Contents

*Preface* iii

*Features of the Book* viii

*Brief Contents* x

## 1. Object-oriented Programming and Evolution of Java 1

- 1.1 Concept of Object orientation 1
- 1.2 Need for Object-oriented Programming 2
- 1.3 Characteristics of Object-oriented Programming 3
  - 1.3.1 Data Abstraction 3
  - 1.3.2 Encapsulation and Information Hiding 3
  - 1.3.3 Access Control to Data Variables and Methods 5
  - 1.3.4 Inheritance 5
  - 1.3.5 Polymorphism 6
- 1.4 Reuse, Coupling, and Cohesion in OOP 6
- 1.5 Object-oriented Analysis and Design 7
- 1.6 History of Development of Programming Languages 8
  - 1.6.1 Object-oriented Languages 9
- 1.7 Development of Java Programming Language 10
  - 1.7.1 Java Bytecode and valid .class File 10
- 1.8 Structure of Java Language 12
- 1.9 Versions of Java 12
- 1.10 Exploring JDK 14
  - 1.10.1 Update Path and Class Path Variables 15
  - 1.10.2 Testing Installations 15
  - 1.10.3 Basic Tools in JDK 16
- 1.11 Salient Features and Benefits of Java Language 17
  - 1.11.1 Exceptions and Exception Handling in Java 18
  - 1.11.2 Security Features of Java Language 18
- 1.12 Differences and Similarities between Java and C++ 19

## 2. Program Structure in Java 25

- 2.1 Introduction 25
  - 2.1.1 Importing Packages and Classes 26
- 2.2 Writing Simple Java Programs 26
- 2.3 Installing and Configuring Java 28
- 2.4 Compiling and Running Java Program 29
  - 2.4.1 Compilers 29
  - 2.4.2 Integrated Development Environments 29
  - 2.4.3 Procedure for Running Programs on NetBeans 8.0.1 30
- 2.5 Elements or Tokens in Java Programs 34
  - 2.5.1 Identifiers or Names 34
  - 2.5.2 Keywords in Java 35
  - 2.5.3 Literals 36
  - 2.5.4 Separators 37
  - 2.5.5 Operators 37
- 2.6 Java Statements 39
- 2.7 Command Line Arguments 40
- 2.8 Data Types 41
- 2.9 Declaration of Variables 42
  - 2.9.1 Types of Variables in Class Declaration 44
- 2.10 User Input to Programs 44
- 2.11 Escape Sequences 47
- 2.12 Comments 48
- 2.13 Programming Style 49
- 2.14 Number System Used in Java Programs 50
  - 2.14.1 Conversion of Binary Number into Hexadecimal Number 51
  - 2.14.2 Conversion of Binary Number into Octal Number 51
- 2.15 Application Programs 51
  - 2.15.1 Area of Polygons 51
  - 2.15.2 Printing Simple Patterns 52

- 2.15.3 Use of Class Math in Mathematical Calculations 53
  - 2.15.4 Making Multiplication Tables 53
  - 2.15.5 Conversion of Decimal Number into Binary Number 54
- 3. Data Types, Variables, and Operators 58**
- 3.1 Introduction 58
  - 3.2 Data Types in Java 59
  - 3.3 Declaration of Variables 60
    - 3.3.1 Non-primitive Types 60
  - 3.4 Data Types 61
    - 3.4.1 Integers 61
    - 3.4.2 Characters 63
    - 3.4.3 Floating Point Numbers 64
    - 3.4.4 boolean Type Data 66
  - 3.5 Type Casting 67
  - 3.6 Scope of Variable Identifier 70
  - 3.7 Literal Constants 72
  - 3.8 Symbolic Constants 73
  - 3.9 Formatted Output with printf() Method 74
    - 3.9.1 Formatting Output of Integer Numbers 76
    - 3.9.2 Formatting Output of Floating Point Numbers 77
    - 3.9.3 Formatting Output of Strings 78
  - 3.10 Static Variables and Methods 79
    - 3.10.1 Mathematical Functions 80
    - 3.10.2 Static Import 82
  - 3.11 Attribute Final 83
  - 3.12 Introduction to Operators 83
  - 3.13 Precedence and Associativity of Operators 84
  - 3.14 Assignment Operator (=) 85
    - 3.14.1 Chain of Assignments 86
  - 3.15 Basic Arithmetic Operators 86
    - 3.15.1 Combinational Arithmetic Operators 87
  - 3.16 Increment (++) and Decrement (--) Operators 88
  - 3.17 Ternary Operator 91
  - 3.18 Relational Operators 91
  - 3.19 Boolean Logical Operators 92
    - 3.19.1 Short-circuit Logical Operators 93
    - 3.19.2 Combinational Boolean Operators 93
    - 3.19.3 Truth Tables 94
  - 3.20 Bitwise Logical Operators 94
  - 3.21 Application Programs 98
    - 3.21.1 Factorial of Numbers 98
    - 3.21.2 Conversion of Decimal Number into Binary Number 99
    - 3.21.3 General Computation Problem 100
    - 3.21.4 Random Number Generation 101
    - 3.21.5 Calculation of Area of Triangle 102
    - 3.21.6 Game of Dice Play 103
- 4. Control Statements 112**
- 4.1 Introduction 112
  - 4.2 if Expression 113
  - 4.3 Nested if Expressions 114
  - 4.4 if-else Expressions 114
    - 4.4.1 Working of if-else Statements 115
    - 4.4.2 if-else Chains 115
  - 4.5 Ternary Operator ?: 117
  - 4.6 Switch Statement 118
    - 4.6.1 Nested Switch Statement 120
  - 4.7 Iteration Statements 121
  - 4.8 while Expression 122
    - 4.8.1 Endless while Loop 124
  - 4.9 do-while Loop 124
  - 4.10 for Loop 126
    - 4.10.1 Generating Samples of Random numbers and Average Values 127
  - 4.11 Nested for Loop 127
  - 4.12 For-Each for Loop 128
    - 4.12.1 Nested for-each for Loop 128
    - 4.12.2 Advantages and Limitations of for-each Loop 129
  - 4.13 Break Statement 130
    - 4.13.1 Break Statement with Labels 131
  - 4.14 Continue Statement 132
    - 4.14.1 Continue Statement with Labels 132
  - 4.15 Application Programs 133
    - 4.15.1 Roots of a Quadratic Equation 133
    - 4.15.2 Summation of a Series with for Loop 134
    - 4.15.3 Drawing in JFrame Window 135
    - 4.15.4 Conversion of Binary Number into Decimal Number 136
    - 4.15.5 Playing a Game based on Random Numbers 137

**5. Classes and Objects 143**

- 5.1 Introduction 143
  - 5.1.1 Types of Classes 144
- 5.2 Class Declaration and Modifiers 144
  - 5.2.1 Class Modifiers 145
- 5.3 Class Members 147
  - 5.3.1 Access Specifiers and Information Hiding 148
- 5.4 Declaration of Class Objects 148
- 5.5 Assigning One Object to Another 150
- 5.6 Access Control for Class Members 152
- 5.7 Accessing Private Members of Class 153
  - 5.7.1 Accessing Private Methods of Class 154
- 5.8 Constructor Methods for Class 155
- 5.9 Overloaded Constructor Methods 156
- 5.10 Nested Classes 157
  - 5.10.1 Member Inner Class 157
  - 5.10.2 Anonymous Class 161
  - 5.10.3 Local Class 163
  - 5.10.4 Static Nested Class 164
- 5.11 Final Class and Methods 165
  - 5.11.1 Abstract Class 166
- 5.12 Passing Arguments by Value and by Reference 167
  - 5.12.1 Passing Arguments through Reference 167
- 5.13 Keyword `this` 168
  - 5.13.1 Instance Variable Hiding 169
- 5.14 Application Programs 170
  - 5.14.1 Drawing Sale Bar Chart 170
- 5.15 Cleaning Process 171
  - 5.15.1 Garbage Collection 172
  - 5.15.2 Finalization 173

**6. Methods 178**

- 6.1 Introduction 178
  - 6.1.1 Placement of Method Definition and Execution of Methods 179
- 6.2 Defining Methods 179
  - 6.2.1 Modifiers for Method 180
  - 6.2.2 Method Type 181
  - 6.2.3 Identifier 181
  - 6.2.4 Parameter List 181
  - 6.2.5 Return Statement 182
- 6.3 Overloaded Methods 182

- 6.4 Overloaded Constructor Methods 184
- 6.5 Class Objects as Parameters in Methods 185
- 6.6 Access Control 186
  - 6.6.1 Accessing Private Data Members 186
  - 6.6.2 Accessing Private Methods 187
  - 6.6.3 Accessor Methods 188
- 6.7 Recursive Methods 189
  - 6.7.1 Fibonacci Numbers 190
- 6.8 Nesting of Methods 191
- 6.9 Overriding Methods 191
- 6.10 Attributes Final and Static 192
  - 6.10.1 Final 192
  - 6.10.2 Static Methods 193
  - 6.10.3 Methods with Variable Arguments 194
- 6.11 Lambda Expression 194
  - 6.11.1 Lambda Function Structure 195
  - 6.11.2 Lambda Expression and Variables 196
  - 6.11.3 Working of Lambda Reference 196
  - 6.11.4 Lambda Function with Local Variables 197
  - 6.11.5 Standard Functionals, their Methods, and Use in Lambda Expression 197
- 6.12 Method References 198
- 6.13 Predicates 199
  - 6.13.1 Composed Predicates 200
- 6.14 Method Chains 203
- 6.15 Keywords `transient`, `volatile`, and `assert` 204
  - 6.15.1 `transient` 205
  - 6.15.2 `volatile` Variable 206
  - 6.15.3 `assert` 207
- 6.16 Native Methods 210
  - 6.16.1 Disadvantages of Using Native Methods 211
  - 6.16.2 General Procedure of Using Native Methods 211

**7. Arrays 217**

- 7.1 Introduction 217
- 7.2 Declaration and Initialization of Arrays 218
  - 7.2.1 Declaration 218
  - 7.2.2 Initialization 219

- 7.2.3 Alternative Array Declaration  
Syntax 220
  - 7.3 Storage of Array in Computer  
Memory 220
  - 7.4 Accessing Elements of Arrays 221
    - 7.4.1 Determination of Array Size 222
    - 7.4.2 Use of *for–each* Loop 222
    - 7.4.3 Array of Strings 224
  - 7.5 Operations on Array Elements 225
    - 7.5.1 Arrays as Parameters of  
Methods 226
  - 7.6 Assigning Array to Another  
Array 227
  - 7.7 Dynamic Change of Array Size 228
  - 7.8 Sorting of Arrays 229
    - 7.8.1 Bubble Sort 229
    - 7.8.2 Selection Sort 231
    - 7.8.3 Sorting by Insertion Method 232
  - 7.9 Search for Values in Arrays 233
    - 7.9.1 Linear Search 234
    - 7.9.2 Binary Search 234
  - 7.10 Class Arrays 236
    - 7.10.1 Methods of Class Arrays 236
  - 7.11 Two-dimensional Arrays 241
    - 7.11.1 Operations on Elements of  
Two-dimensional Arrays 243
  - 7.12 Arrays of Varying Lengths 244
  - 7.13 Three-dimensional Arrays 245
  - 7.14 Arrays as Vectors 248
  - 7.15 Application Programs 251
    - 7.15.1 Computation of Mean and  
Standard Deviation 251
    - 7.15.2 Creating Bar Chart with Array  
Elements 252
    - 7.15.3 Operations on Matrices through  
Two-dimensional Arrays 254
    - 7.15.4 Programming for Solution of  
Linear Algebraic Equations 257
- 8. Inheritance 264**
- 8.1 Introduction 264
    - 8.1.1 Benefits of Inheritance 265
  - 8.2 Process of Inheritance 266
    - 8.2.1 Substitutability and  
Subtyping 267
  - 8.3 Types of Inheritances 268
    - 8.3.1 Disadvantages of Inheritance 272
  - 8.4 Universal Super Class—Object  
Class 272
  - 8.5 Inhibiting Inheritance of Class Using  
Final 273
  - 8.6 Access Control and Inheritance 274
    - 8.6.1 Private Members and  
Inheritance 276
  - 8.7 Multilevel Inheritance 277
  - 8.8 Application of Keyword Super 278
  - 8.9 Constructor Method and  
Inheritance 280
  - 8.10 Method Overriding 281
    - 8.10.1 Binding 284
  - 8.11 Dynamic Method Dispatch 285
  - 8.12 Abstract Classes 286
  - 8.13 Interfaces and Inheritance 286
    - 8.13.1 Interfaces with Default and Static  
Methods 287
  - 8.14 Application Programs 289
- 9. Interfaces 297**
- 9.1 Introduction 297
    - 9.1.1 Similarities between Interface and  
Class 298
    - 9.1.2 Dissimilarities between Class and  
Interface 299
    - 9.1.3 Rules for Classes that Implement  
Interface 299
    - 9.1.4 Types of Interfaces 299
  - 9.2 Declaration of Interface 299
    - 9.2.1 Interface Modifiers 300
    - 9.2.2 Members of Interface 300
  - 9.3 Implementation of Interface 300
    - 9.3.1 Constants in Interfaces 302
  - 9.4 Multiple Interfaces 303
    - 9.4.1 Interface References 303
    - 9.4.2 Stub Methods 305
  - 9.5 Nested Interfaces 306
  - 9.6 Inheritance of Interfaces 308
  - 9.7 Default Methods in Interfaces 309
  - 9.8 Static Methods in Interface 313
  - 9.9 Functional Interfaces 314
    - 9.9.1 Functional Consumer<T> 318
    - 9.9.2 Functional Supplier 319
  - 9.10 Annotations 319
    - 9.10.1 Benefits of Using  
Annotations 321
    - 9.10.2 Annotation Basics 321
    - 9.10.3 Retention Policy 322
    - 9.10.4 Predefined Annotations 322
    - 9.10.5 Class Method 323
    - 9.10.6 User Defined Annotation 324
    - 9.10.7 Restrictions on Annotations 328
  - 9.11 Application Programs 328

**10. Packages and Java Library 336**

- 10.1 Introduction 336
  - 10.1.1 Packages as Name Space 337
- 10.2 Defining Package 337
  - 10.2.1 Other Fully Qualified Names 338
- 10.3 Importing Packages and Classes into Programs 338
  - 10.3.1 Importing Class in Programs 339
  - 10.3.2 User-defined Packages and Classes 339
- 10.4 Path and Class Path 341
- 10.5 Access Control 347
- 10.6 Packages in Java SE 8 348
- 10.7 Java.lang Package and its Classes 348
- 10.8 Class Object 350
- 10.9 Enumeration 352
  - 10.9.1 Members of Enum 352
  - 10.9.2 Methods of Enumeration 353
  - 10.9.3 General Enum Declaration 354
- 10.10 class Math 357
- 10.11 Wrapper Classes 360
  - 10.11.1 Constructors of *Wrapper* Classes 361
  - 10.11.2 Methods of Wrapper Classes 361
- 10.12 Auto-boxing and Auto-unboxing 366
  - 10.12.1 Methods `isInfinite()` and `isNaN()` 367
- 10.13 Java util Classes and Interfaces 368
  - 10.13.1 Interfaces Defined in `java.util` 370
  - 10.13.2 Scanner Class 371
  - 10.13.3 Radix 376
- 10.14 Formatter Class 376
  - 10.14.1 Specification of Field Width 378
- 10.15 Random Class 380
  - 10.15.1 Constructors of Class 381
  - 10.15.2 Methods of Class Random 381
- 10.16 Time Package 383
  - 10.16.1 New Time Package of Java SE 8 (`Java.time`) 384
- 10.17 Class Instant (`java.time.Instant`) 389
  - 10.17.1 Constants Defined in Class 389
  - 10.17.2 Methods of Class Instant 389
- 10.18 Formatting for Date/Time in Java 8 391

- 10.18.1 Predefined Constants with Different Styles 391
- 10.19 Temporal Adjusters Class 394
  - 10.19.1 Temporal Adjuster Interface 394
  - 10.19.2 TemporalAdjusters class 395
- 10.20 Legacy Date Time Classes 396
- 10.21 Application Program 397

**11. Exception Handling 405**

- 11.1 Introduction 405
- 11.2 Hierarchy of Standard Exception Classes 406
- 11.3 Keywords `throws` and `throw` 408
- 11.4 `try`, `catch`, and `finally` Blocks 409
  - 11.4.1 `try {}` Block 409
  - 11.4.2 `catch {}` Block 410
  - 11.4.3 `finally {}` Block 410
  - 11.4.4 Uncaught Exception 412
  - 11.4.5 `finalize` in Exception Handling 413
  - 11.4.6 Constructors throwing Exception 413
- 11.5 Multiple Catch Clauses 413
- 11.6 Class `Throwable` 415
  - 11.6.1 Constructors of class `Throwable` 415
  - 11.6.2 Methods of class `Throwable` 416
- 11.7 Unchecked Exceptions 417
- 11.8 Checked Exceptions 420
- 11.9 `try-with-resources` 422
- 11.10 Catching Subclass Exception 426
- 11.11 Custom Exceptions 428
- 11.12 Nested `try` and `catch` Blocks 429
- 11.13 Rethrowing Exception 430
- 11.14 `throws` Clause 431
- 11.15 Application Program 432
  - 11.15.1 Design of Cellular Mobile System 432
  - 11.15.2 Design of Spindle Speeds of Machine Tools 434
- 11.16 Best Practices of Dealing with Exceptions 436

**12. String Handling in Java 442**

- 12.1 Introduction 442
  - 12.1.1 Overview of Three String Classes 442
  - 12.1.2 Storage of Strings 443
  - 12.1.3 Immutability 445
- 12.2 Interface `CharSequence` 445



- 12.3 Class `String` 446
    - 12.3.1 Constructors of Class `String` 446
  - 12.4 Methods for Extracting Characters from Strings 449
  - 12.5 Methods for Comparison of Strings 451
    - 12.5.1 Difference between `==` and `Method Equals()` 451
  - 12.6 Methods for Modifying Strings 454
  - 12.7 Methods for Searching Strings 455
  - 12.8 Data Conversion and Miscellaneous Methods 456
    - 12.8.1 Exceptions thrown by `String` Class Methods 458
  - 12.9 Class `StringBuffer` 459
    - 12.9.1 Constructors of Class `Stringbuffer` 459
    - 12.9.2 Methods of Class `Stringbuffer` 459
  - 12.10 Class `StringBuilder` 462
    - 12.10.1 Constructors of Class `StringBuilder` 463
    - 12.10.2 Methods of Class `StringBuilder` 463
  - 12.11 Application Program 467
- 13. Multithreaded Programming 474**
- 13.1 Introduction 474
  - 13.2 Need for Multiple Threads 474
    - 13.2.1 Multithreaded Programming for Multi-core Processor 475
  - 13.3 Thread Class 477
    - 13.3.1 Constructors of class `Thread` 477
    - 13.3.2 Thread Group 478
    - 13.3.3 Methods of Thread Class 479
  - 13.4 Main Thread 481
  - 13.5 Creation of New Threads 483
    - 13.5.1 Creation of New Thread by Extending Thread Class 483
    - 13.5.2 Creation of New Threads by Implementing `Runnable` 485
    - 13.5.3 Creation of Threads by Lambda Expression, Method Reference, and Anonymous Class 488
    - 13.5.4 Creation of Multiple Threads by `Runnable` 489
    - 13.5.5 Creation of Multiple Threads by Extending Thread Class 490
  - 13.6 Thread States 491
  - 13.7 Thread Priority 493
  - 13.8 Synchronization 496
    - 13.8.1 Synchronizing Statement for Blocks of Statements 497
  - 13.9 Deadlock and Race Situations 498
  - 13.10 Inter-thread Communication 499
  - 13.11 Suspending, Resuming, and Stopping of Threads 501
  - 13.12 Application Programs with Threads 503
    - 13.12.1 Producer–Consumer Problem 503
    - 13.12.2 Application Program Using Lambda Expression 506
    - 13.12.3 General Computation by Threads Using Method Reference 507
    - 13.12.4 Threads Computing Streams of Random Numbers—Anonymous Class Application 509
    - 13.12.5 New Thread Loading Images on to AWT Window 510
    - 13.12.6 Solution of Linear Algebraic Equations and Sorting of Arrays 511
- 14. Generic Programming 519**
- 14.1 Introduction 519
    - 14.1.1 General Benefits of Using Generics 520
    - 14.1.2 General Convention for Representation of Type Parameter 520
    - 14.1.3 Erasure, Generics, and JVM 521
  - 14.2 Generics and Primitive Types 521
  - 14.3 Declaration of Generic Class and Constructor 521
  - 14.4 Use of Object Class vs Generic Class 523
  - 14.5 Generic Class with Multiple Type Parameters 525
  - 14.6 Generic Method 526
  - 14.7 Generic Method Overloading 529
  - 14.8 Generic Interface 529
  - 14.9 Overriding Methods in Generics Class 531
  - 14.10 Upper Bound on Types 532
  - 14.11 Multiple Bounds on Types 534
  - 14.12 Wildcard 535
    - 14.12.1 Restrictions on Wildcard Type 535
  - 14.13 Bounded Wildcard 536

- 14.14 Generic Constructor in Non-generic Class 538
- 14.15 Generic Super Class and Generic Subclass 539
- 14.16 Non-generic Super Class with Generic Subclass 540
- 14.17 Generic Type Parameters and Inheritance 541
- 14.18 Generics and Reflection 542
- 14.19 Restrictions in Java Generics 545
- 14.20 Application Programs 546
  - 14.20.1 Generic Methods for Sorting and Display of Objects 547
- 15. Image Handling in Java 554**
  - 15.1 Introduction 554
  - 15.2 Image File Formats 555
  - 15.3 Image Handling and Processing 556
    - 15.3.1 Image Processing 556
    - 15.3.2 Java Classes and Interfaces Used in Image Processing 556
  - 15.4 Image Class 556
    - 15.4.1 Class Field Variables 557
    - 15.4.2 Methods of Class Image 557
  - 15.5 Loading and Displaying Images 558
    - 15.5.1 Display of Images 559
    - 15.5.2 ImageObserver 559
    - 15.5.3 Using Class ImageIO Method for Loading BufferedImage 562
  - 15.6 Pixel Color and Transparency Specifications 563
    - 15.6.1 Representation of Pixel Colour 564
  - 15.7 Class BufferedImage 565
    - 15.7.1 Fields of Class BufferedImage 565
    - 15.7.2 Constructors of BufferedImage Class 565
    - 15.7.3 Methods of Buffered Image Class 565
    - 15.7.4 Drawing Buffered Images Using Graphics2D 568
  - 15.8 Class MemoryImageSource 569
    - 15.8.1 Constructors of Class MemoryImageSource 569
  - 15.9 Toolkit Class 570
    - 15.9.1 Methods of Class Toolkit 570
  - 15.10 Creating Images from Memory Imagesource 570
  - 15.11 Manipulation of Images 574
    - 15.11.1 Greyscale Images 575
- 16. Collections 583**
  - 16.1 Introduction 583
    - 16.1.1 Purpose of Collection Framework 584
    - 16.1.2 Application of Collection Framework 584
  - 16.2 Hierarchy of Collection Interfaces/ Classes 584
  - 16.3 Methods Defined in Collection Interface 586
    - 16.3.1 Exceptions Thrown by Methods 587
  - 16.4 Interface Iterator 588
    - 16.4.1 Methods Defined in Iterator 588
    - 16.4.2 Iterator Versus ListIterator 589
  - 16.5 Collection Classes 589
  - 16.6 Set 591
    - 16.6.1 HashSet Class 591
    - 16.6.2 Constructors of Class HashSet 591
    - 16.6.3 Methods of Class Hashset 591
    - 16.6.4 Class TreeSet 593
    - 16.6.5 Constructors of Class TreeSet 594
    - 16.6.6 Methods of TreeSet class 594
    - 16.6.7 Comparator Interface 598
    - 16.6.8 Application of Iterators for TreeSet Instances 599
  - 16.7 ArrayList Class 601
    - 16.7.1 Constructors of class ArrayList 601
    - 16.7.2 Methods of Class ArrayList 601
    - 16.7.3 Other Classes that Implement List 605
  - 16.8 Class LinkedList 605
    - 16.8.1 Constructors of the Class LinkedList 605
    - 16.8.2 Methods of Class LinkedList 606
    - 16.8.3 Comparison of LinkedList and ArrayList classes 610
  - 16.9 Queues 611
  - 16.10 Class PriorityQueue 612
    - 16.10.1 Class Declaration 612
    - 16.10.2 Constructors of Class PriorityQueue 613



- 16.10.3 Methods of Class
    - PriorityQueue 613
  - 16.11 Deque 614
  - 16.12 Vector Class 616
    - 16.12.1 Constructors of Class
      - Vector 616
  - 16.13 Stack Class 619
  - 16.14 Maps 621
    - 16.14.1 Map Interfaces 622
  - 16.15 HashMap class 623
    - 16.15.1 Constructors of Hashmap
      - class 623
    - 16.15.2 Methods of class
      - HashMap 623
  - 16.16 TreeMap class 627
    - 16.16.1 Constructor Methods 627
    - 16.16.2 Methods of class TreeMap 627
  - 16.17 Class Hashtable<K,V> 631
    - 16.17.1 Constructors of class
      - Hashtable 632
  - 16.18 class Properties 634
    - 16.18.1 Constructors of Class
      - Properties 634
  - 16.19 Class Collections and Algorithms 636
    - 16.19.1 Algorithms—Collections
      - UnModifiable 637
    - 16.19.2 Algorithms for Synchronizing
      - Collections 638
    - 16.19.3 Algorithms—Collections
      - Dynamically Typesafe 638
    - 16.19.4 Search and Sort
      - Algorithms 639
  - 16.20 Application Programs 644
- 17. Input/Output Streams and File Operations 652**
- 17.1 Introduction 652
    - 17.1.1 Base I/O Classes 654
    - 17.1.2 Standard Input/Output
      - Streams 655
    - 17.1.3 Classes Files and
      - RandomAccessFile 656
  - 17.2 Class File 656
    - 17.2.1 Constructors of Class File 657
    - 17.2.2 Methods of Class File 658
  - 17.3 Reading and Writing Bytes 662
    - 17.3.1 Class InputStream 662
    - 17.3.2 Subclasses of Inputstream 663
  - 17.4 FileInputStream 664
    - 17.4.1 Constructors of
      - FileInputStream 664
    - 17.4.2 Methods of
      - FileInputStream 665
  - 17.5 ByteArrayInputStream 666
  - 17.6 OutputStream and its Subclasses 667
    - 17.6.1 Sub-classes of Outputstream 668
    - 17.6.2 Methods of OutputStream 668
  - 17.7 FileOutputStream 669
    - 17.7.1 Constructors of Class
      - FileOutputStream 669
    - 17.7.2 Methods of Class
      - FileOutputStream 669
  - 17.8 DataInputStream and
    - DataOutputStream 671
    - 17.8.1 Class DataInputSream 671
    - 17.8.2 Class DataOutputStream 672
  - 17.9 Class Writer 674
    - 17.9.1 Subclasses of Class
      - Writer 675
    - 17.9.2 Methods of Writer Class 675
  - 17.10 Class PrintWriter 675
    - 17.10.1 Constructors of Class
      - PrintWriter 676
  - 17.11 **Class Reader 677**
    - 17.11.1 Constructors 678
    - 17.11.2 Methods of Class Reader 678
    - 17.11.3 Subclasses of Reader
      - Class 679
    - 17.11.4 Subclasses of Reader
      - Class 679
  - 17.12 Class Console 682
    - 17.12.1 Methods of Class Console 682
  - 17.13 Piped Input/Output 684
    - 17.13.1 Class
      - PipedOutputStream 685
    - 17.13.2 Class PipedInputStream 685
  - 17.14 Serialization 687
    - 17.14.1 Class
      - ObjectOutputStream 688
    - 17.14.2 Class
      - ObjectInputStream 689
  - 17.15 Class RandomAccessfile 692
    - 17.15.1 Constructors 692
    - 17.15.2 Methods of Class Random
      - AccessFile 693
  - 17.16 java.nio and java.nio.2
    - Packages 694
    - 17.16.1 Differences in java.io and
      - java.nio 695
  - 17.17 Path 695

- 17.17.1 Absolute Path 696
  - 17.17.2 Relative Path 696
  - 17.17.3 *interface* Path 696
  - 17.17.4 Operations Associated with Paths 697
  - 17.18 Class Paths 697
    - 17.18.1 Methods of Class 697
  - 17.19 Class Files 699
    - 17.19.1 Methods Defined in Class Files 699
  - 17.20 Buffers 702
    - 17.20.1 Buffers Types 702
    - 17.20.2 Creation of Buffer 703
    - 17.20.3 Methods of Class *ByteBuffer* 704
  - 17.21 Channels 706
    - 17.21.1 Class *FileChannel* 706
    - 17.21.2 Creation of File Channels 707
    - 17.21.3 Methods of Class *FileChannel* 707
  - 17.22 Application Programs 710
- 18. Applets 718**
- 18.1 Introduction 718
  - 18.2 Applet Architecture 719
  - 18.3 Applet Class and Methods 720
  - 18.4 Creating Applets 721
    - 18.4.1 Life Cycle of Applets 722
  - 18.5 HTML Tags 723
    - 18.5.1 HTML `<APPLET>` Tag 723
  - 18.6 Simple Applet Display Methods 725
    - 18.6.1 Using Status Window 726
  - 18.7 Passing Parameters to Applets 727
  - 18.8 Passive Applet Programs 729
    - 18.8.1 Labels in Applets 730
  - 18.9 Adding Images to Applet Windows 730
    - 18.9.1 Loading Images in Applet Windows 731
    - 18.9.2 Display of Images 731
  - 18.10 Display of Numerical Values on Applet Windows 732
  - 18.11 Managing Colours in Applet Window 733
    - 18.11.1 Setting and Getting Colours 734
    - 18.11.2 Changing Colours of Strings 734
  - 18.12 Interactive Applets with AWT Graphical Components 736
  - 18.13 AWT Textfields in Applets 738
  - 18.14 Animation in Applet Windows 741
  - 18.15 Interactive Applets with Swing Components 742
  - 18.16 Running Applets on NetBeans IDE 744
  - 18.17 Application Program 746
    - 18.17.1 Application 2: Calculating Number of Users Supported in TDMA System 747
  - 18.18 Applet and Security 749
- 19. Event Handling 754**
- 19.1 Introduction 754
  - 19.2 Delegation Event Model 754
  - 19.3 Hierarchy of Event Classes 757
  - 19.4 Types and Sources of Events 758
  - 19.5 Event Listener Interfaces 759
  - 19.6 class *ActionEvent* 759
    - 19.6.1 Field Variable 760
    - 19.6.2 Constructors of class *ActionEvent* 760
    - 19.6.3 Methods of class *ActionEvent* 760
  - 19.7 class *AdjustmentEvent* 763
    - 19.7.1 Adjustment Types 764
    - 19.7.2 Class Constructors 764
    - 19.7.3 Class Methods 764
  - 19.8 *ChangeEvent* and *ChangeListener* 767
  - 19.9 class *ComponentEvent* 769
  - 19.10 class *ContainerEvent* 771
    - 19.10.1 Fields Defined in Class 771
    - 19.10.2 Methods of Class 772
  - 19.11 class *FocusEvent* 772
  - 19.12 class *ItemEvent* 773
    - 19.12.1 Class Fields 773
    - 19.12.2 Class Constructor 773
    - 19.12.3 Class Methods 773
    - 19.12.4 Anonymous Inner Class 774
  - 19.13 class *KeyEvent* 775
    - 19.13.1 Methods of class *KeyEvent* 776
  - 19.14 class *MouseEvent* 778
    - 19.14.1 Field Variables of the Class 779
    - 19.14.2 Methods of Class *MouseEvent* 779
    - 19.14.3 *MouseEvent* and *MouseEvent* 783
    - 19.14.4 class *MouseEvent* 785

- 19.15 class `TextEvent` 785
- 19.16 class `WindowEvent` 787
- 19.17 Details of Event Listener Interfaces 788
  - 19.17.1 interface `ActionListener` 788
  - 19.17.2 interface `AdjustmentListener` 788
  - 19.17.3 interface `ComponentListener` 788
  - 19.17.4 interface `ContainerListener` 788
  - 19.17.5 interface `FocusListener` 788
  - 19.17.6 interface `ItemListener` 789
  - 19.17.7 interface `KeyListener` 789
  - 19.17.8 interface `MouseListener` 789
  - 19.17.9 interface `MouseMotionListener` 789
  - 19.17.10 interface `MouseWheelListener` 789
  - 19.17.11 interface `TextListener` 789
  - 19.17.12 interface `WindowFocusListener` 789
  - 19.17.13 interface `WindowListener` 789
- 19.18 Adapter Classes 790
- 19.19 Application Program 792
- 20. Creating GUIs in AWT Windows 801**
  - 20.1 Introduction 801
  - 20.2 Hierarchy of Java Awt Classes 802
  - 20.3 Window Display System 802
  - 20.4 Adding Components to Window 805
  - 20.5 Labels 805
    - 20.5.1 Constructor Methods of class `Label` 805
    - 20.5.2 Methods of class `Label` 806
  - 20.6 Button 807
  - 20.7 Checkbox 810
    - 20.7.1 Constructors of `Checkbox` Class 810
    - 20.7.2 Methods of class `Checkbox` 810
  - 20.8 Textfield 812
    - 20.8.1 Constructors of class `TextField` 813
    - 20.8.2 Methods of class `TextField` 813
  - 20.9 Menus and Menubar 815
    - 20.9.1 Sub-menu 817
- 20.10 Layout Manager 818
  - 20.10.1 class `FlowLayout` 818
  - 20.10.2 Border Layout 820
  - 20.10.3 GridLayout 822
  - 20.10.4 GridBagLayout 824
- 20.11 Scroll Bar 828
  - 20.11.1 Constructors of class `Scrollbar` 828
  - 20.11.2 Methods of class `Scrollbar` 829
- 20.12 Application Programs 829
- 21. Drawing in AWT Windows 839**
  - 21.1 Introduction 839
    - 21.1.1 Class `Graphics2D` 840
    - 21.1.2 Methods of Class `Graphics` 840
    - 21.1.3 Attributes of Methods 840
  - 21.2 Display Coordinate System in a Window 841
  - 21.3 Drawing Strings in AWT Window 842
  - 21.4 Drawing Lines 843
    - 21.4.1 Drawing Lines in `java.awt.Frame` Class `Window` 843
    - 21.4.2 Drawing Lines in `AppletWindow` 844
    - 21.4.3 Drawing Polygons by Drawing Lines 844
  - 21.5 Drawing Rectangles By Method `drawRect()` 845
    - 21.5.1 Drawing of Rounded Rectangles 846
  - 21.6 Working with Colours 847
  - 21.7 Drawing Polygons 849
    - 21.7.1 Drawing Filled Polygons 850
    - 21.7.2 Drawing Polygons By `addPoint()` Method 850
  - 21.8 Drawing Ellipses and Circles 851
    - 21.8.1 Drawing of Fun Figures with Ovals 852
  - 21.9 Drawing Arcs 852
  - 21.10 Fonts 854
    - 21.10.1 Methods Defined in `Font` Class 854
    - 21.10.2 Font Metrics 855
  - 21.11 Class `Graphics2D` 857
    - 21.11.1 Methods of Class `Graphics2D` 858
  - 21.12 Class `BasicStroke` 863
    - 21.12.1 Constructors of Class `BasicStrokes` 864

- 21.13 Drawing Full and Dashed (Dotted) Lines 864
  - 21.14 Clipping 869
  - 21.15 Colour Gradient 870
  - 21.16 Drawing an Image with Graphics2D 873
  - 21.17 class PATH2D 874
    - 21.17.1 Fields 874
    - 21.17.2 Methods of Path2D 875
  - 21.18 Application Programs 878
    - 21.18.1 Drawing a Bar Chart 878
- 22. Swing—Part 1 887**
- 22.1 Introduction 887
    - 22.1.1 Benefits of Using Swing over AWT 888
  - 22.2 Hierarchy of Swing Classes 888
    - 22.2.1 Top Swing GUI Containers 889
  - 22.3 Javax Swing Packages 889
  - 22.4 Swing Window Panes 890
  - 22.5 Swing Components and Classes 892
  - 22.6 Adding Components to a Container 893
    - 22.6.1 JPanel 894
  - 22.7 JLabel 895
    - 22.7.1 Constructors of JLabel class 895
    - 22.7.2 Methods of JLabel class 895
    - 22.7.3 Layout of JLabels with setBounds() 897
    - 22.7.4 JLabel with Icon 898
  - 22.8 JButton 898
  - 22.9 JRadioButton 900
  - 22.10 Lambda Expression for ActionListener 902
  - 22.11 JTextField 906
    - 22.11.1 Constructors of Class JTextField 906
  - 22.12 JMenu 908
  - 22.13 Border Classes 911
    - 22.13.1 class BevelBorder 911
    - 22.13.2 class EmptyBorder 914
    - 22.13.3 class MatteBorder 914
    - 22.13.4 Class EtchedBorder 915
    - 22.13.5 class LineBorder 916
    - 22.13.6 MatteBorder 918
    - 22.13.7 More Illustrations of Borders and Codes 920
  - 22.14 Application Program 921
- 23. Swing—Part 2 931**
- 23.1 Introduction 931
  - 23.2 Swing Components and Threads 932
  - 23.3 Component Layout Manager Classes 932
    - 23.3.1 BorderLayout 932
    - 23.3.2 FlowLayout 935
    - 23.3.3 GridLayout 938
    - 23.3.4 CardLayout 939
    - 23.3.5 BoxLayout 941
    - 23.3.6 NullLayout 943
    - 23.3.7 OverlayLayout 944
  - 23.4 JTable 945
    - 23.4.1 Constructors 945
    - 23.4.2 Methods of class JTable 946
  - 23.5 JScrollBar 949
  - 23.6 JScrollPane 951
  - 23.7 JList 952
  - 23.8 JCheckBox 954
  - 23.9 JToolTip 956
  - 23.10 JSlider 957
  - 23.11 JTree Class 959
  - 23.12 Adding Components to Multiple Panels 961
  - 23.13 Look and Feel of Graphical Components 965
  - 23.14 Application Programs 967
- 24. Networking 977**
- 24.1 Introduction 977
  - 24.2 Networking Basics 978
  - 24.3 Protocols 979
    - 24.3.1 Internet Address (IPv4 and AND IPv6) 980
    - 24.3.2 Ports 980
    - 24.3.3 Sockets 981
    - 24.3.4 Threads in Socket Programming 981
  - 24.4 Classes and Interfaces in java.net 982
  - 24.5 class InetAddress 983
    - 24.5.1 Constructors of class in Server-side Program InetAddress 983
    - 24.5.2 Methods of class InetAddress 983
  - 24.6 class URL 985
    - 24.6.1 Constructors of URL class 985
    - 24.6.2 Methods of URL class 986
  - 24.7 class URLConnection 988

- 24.8 TCP/IP Server Socket Programming 990
  - 24.8.1 Constructors of class Socket 990
  - 24.8.2 Methods of class Socket 991
- 24.9 Class ServerSocket 992
  - 24.9.1 Constructors of class ServerSocket 992
  - 24.9.2 Methods of class ServerSocket 992
- 24.10 Communication through Sockets 993
  - 24.10.1 Client-side Socket Programming 994
- 24.11 Users/Unreliable Datagram Protocol 999
- 24.12 class DatagramPacket 999
  - 24.12.1 Constructors 1000
  - 24.12.2 Methods of class DatagramPackets 1000
- 24.13 DatagramSocket Class 1001
  - 24.13.1 Constructors of class DatagramSocket 1001
  - 24.13.2 Methods of class DatagramSocket 1001
- 24.14 Programs for Sending and Receiving Datagram 1002
  - 24.14.1 Creating a DatagramPacket for Sending and Receiving 1003
  - 24.14.2 Writing Code for Receiving Datagram 1004
- 25. Java Beans 1015**
  - 25.1 Introduction to Java Beans 1015
    - 25.1.1 Attributes of Beans 1016
    - 25.1.2 Benefits of Using Beans 1016
    - 25.1.3 Properties of a Bean 1017
  - 25.2 Java Bean API 1018
    - 25.2.1 Interfaces 1018
    - 25.2.2 Classes of java.bean package (Java SE 8) 1019
    - 25.2.3 Exceptions 1019
  - 25.3 class Beans 1019
    - 25.3.1 Class Methods 1020
    - 25.3.2 class Property ChangeSupport 1020
  - 25.4 JAR Files 1024
    - 25.4.1 Creating a JAR file 1024
    - 25.4.2 Viewing the contents of a JAR File 1025
    - 25.4.3 Extracting the Files from JAR File 1025
  - 25.5 Building Java Beans with NETBEAN IDE 1026
    - 25.5.1 Building a Composite Component Bean 1026
    - 25.5.2 Changing Properties of Components 1030
    - 25.5.3 Adding Events and Event Listeners 1033
  - 25.6 Source Code Generated by IDE 1035
  - 25.7 Java Beans Project 1—Construct a Bean Containing a Label and a Scroll bar 1038
  - 25.8 Java Beans Project 2— Construct a Bean to Illustrate Mouse Events 1040
  - 25.9 Java Beans Project 3— Construct a Bean to Illustrate Working of Radio Buttons 1042
  - 25.10 Java Beans Project 4—Construct a Bean Containg ComboBox 1045
  - 25.11 Java Beans Project 5 1048
  - 25.12 Application Program 1052
    - 25.12.1 Application Project 1 1052
    - 25.12.2 Application Project 2 1053
- 26. Java Servlets 1060**
  - 26.1 Introduction 1060
  - 26.2 HTTP Basics 1061
  - 26.3 Life Cycle of a Servlet 1062
  - 26.4 Servlet Architecture 1064
    - 26.4.1 Setting up Web Server for Servlet Deployment 1065
  - 26.5 Classes Defined in javax.servlet.http Package 1065
    - 26.5.1 Interfaces Defined in javax.servlet.http Package 1068
    - 26.5.2 Handling HTTP Request and Response 1069
  - 26.6 Writing Servlet 1070
    - 26.6.1 Reading Servlet Parameters 1073
    - 26.6.2 Classes in javax.servlet Package 1079
    - 26.6.3 Interfaces Defined in javax.servlet Package 1081
  - 26.7 Session Tracking and Session Management 1086
    - 26.7.1 Session Tracking Techniques 1087

- 26.8 Http Redirects in Servlets 1093
- 26.9 Securing Servlets 1094
- 26.10 Application Program 1095
- 27. Java Database Connectivity 1102**
  - 27.1 Introduction 1102
  - 27.2 JDBC Architecture 1103
    - 27.2.1 Two-tier Architecture for Data Access 1103
    - 27.2.2 Three-tier Architecture for Data Access 1103
  - 27.3 Installing MySQL and MySQL Connector/J 1104
    - 27.3.1 SQL Statements 1105
  - 27.4 JDBC Environment Setup 1106
    - 27.4.1 JDBC Connectivity Model and API 1107
  - 27.5 Establishing JDBC Database Connections 1108
    - 27.5.1 Load and Register the JDBC Driver 1109
    - 27.5.2 Defining the Connection URL 1110
  - 27.6 ResultSet Interface 1113
    - 27.6.1 Navigating the ResultSet 1115
    - 27.6.2 ResultSetMetaData interface 1116
  - 27.7 Creating JDBC Application 1116
  - 27.8 JDBC Batch Processing 1127
  - 27.9 JDBC Transaction Management 1129
  - 27.10 Application Programs 1133
- Appendix A Enhancements in Different Versions of Java 1145*
- Appendix B Creating Webpages 1150*
- Java Interview Questions 1171*
- Index 1181*
- About the Authors 1191*

# Interfaces

## Learning Objectives

*After reading this chapter, you will be able to*

- understand what an interface is
- declare an interface
- know the types of interfaces
- familiarize yourself with the members of an interface
- learn the implementation of an interface
- comprehend nested interfaces
- be aware of inheritance of interfaces
- study the Java SE8 additions to interface, which can now have the following methods:
  - default methods
  - static method
- Java SE8 has also been added
  - functional interfaces – `Java.util.function`
  - predicates
- write programs involving
  - interfaces and implementation of methods declared in interfaces
  - multiple interfaces and implementation of methods declared in these interfaces
  - concept of nested interfaces
  - default and static methods in interfaces
  - concept of multiple inheritance using interfaces
  - functional interfaces

## 9.1 Introduction

Similar to classes, an interface also introduces a new reference type. An interface represents an encapsulation of constants, classes, interfaces, and one or more abstract methods that are implemented by a class. An interface does not contain instance variables. An interface cannot implement itself; it has to be implemented by a class. The methods in an interface have no body (except those declared default or static); only headers are declared with the parameter list that is followed by a semicolon. The class that implements the interface has to have full definitions of all the abstract methods included in the interface. An interface can be implemented by any number of unrelated classes with their own definitions of the methods of the interface.





**Fig. 9.1** Interface provides means to induct polymorphism. The figure shows how various uses of constants or methods provided in an interface can be realized.

Different classes can have different definitions of the same methods but the parameter list must be identical to that in the interface. Thus, interfaces provide another way of dynamic polymorphic implementation of methods. Figure 9.1 illustrates how a variable declared in an interface may be used differently in implementing classes.

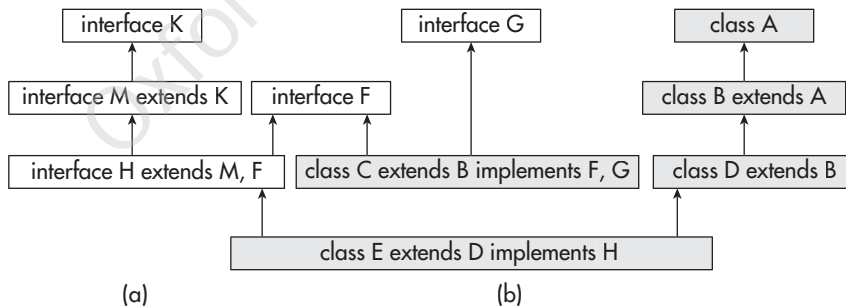
In this figure, a lady is asking her child what she wants to be stitched out of a piece of cloth—if she wants a frock, skirt, or suit. In this case, the piece of cloth is an analogy to abstract method contained in interface. As subclass provides the whole definition of this abstract method, the child would provide the dimensions and style of the outfit to be stitched.

According to modifications in Java SE8, an interface can now have default methods and static methods with full definitions and these methods are inherited by the classes that implement the interface. At the same time, the class may override the methods if necessary (see Sections 9.7 and 9.8 for the full discussion).

Any number of interfaces can be implemented by a class. This, to some extent, fulfils the need for multiple inheritance. The multiple inheritances of classes are not allowed in Java, and therefore, interfaces provide a stopgap arrangement. A class can extend (inherit) another class as well as implement a number of interfaces. For details on inheritance, see Section 8.13 in Chapter 8. With the enhancement of Java SE8, the benefits of multiple inheritances can be realized easily.

The interfaces can be extended as well as nested like classes. However, there are differences too. The similarities and dissimilarities of an interface with a class are as follows.

Figure 9.2 shows how the interfaces inherit other interfaces and the classes that implement them. A class may implement a number of interfaces besides having a super class.



**Fig. 9.2** Class can implement a number of interfaces but can extend only one class. An interface can extend any number of interfaces (a) Inheritance of interfaces (b) Inheritance of classes

There are several similarities and differences between an interface and a class, which are discussed in Sections 9.1.1 and 9.1.2.

### 9.1.1 Similarities between Interface and Class

1. Declaring an interface is similar to that of class; the keyword *class* is replaced by keyword *interface*.
2. Its accessibility can be controlled just like a class. An interface declared public is accessible to any class in any package, whereas the ones without an access specifier is accessible to classes in the same package only.



3. One can create variables as object references of interface that can use the interface.
4. It can contain inner classes (nested classes) and inner interfaces.
5. Since Java 8, an interface can have full definitions of methods with default or static modifiers.

### 9.1.2 Dissimilarities between Class and Interface

1. Interface cannot implement itself; it must be implemented by a class.
2. An interface can contain only method headers followed by a semicolon. It cannot have the full definition of a method. The full definition is given in the class that implements it. Java 8 modification allows the default and static method declarations in interfaces.
3. The methods declared in the interface are implicitly public.
4. An interface does not contain instance variables.
5. The variables declared in an interface are implicitly public, static, and final, that is, they are constants.
6. Interfaces cannot have a constructor like a class.
7. An interface cannot extend a class nor can it have a subclass. It can only extend other interfaces.
8. A class can extend (inherit) only one class but an interface can extend any number of interfaces. It is illustrated in Fig. 8.3 (see Chapter 8 for more details on inheritance).

### 9.1.3 Rules for Classes that Implement Interface

1. A non-abstract class that implements an interface must have concrete implementation of all the abstract methods of the interface; otherwise, the class will not compile.
2. The `@Override` annotation should be used on the definitions of interface methods in the class.
3. The methods declared static and default with full definitions in an interface are inherited by the implementing class since Java 8.
4. The class implementing a method of an interface must retain the exact signature of the method.
5. An abstract class need not implement all the abstract methods of the interfaces that it implements.

### 9.1.4 Types of Interfaces

From the aforementioned discussion, it is clear that an interface comprises a collection of abstract methods. It cannot contain full implementation of methods but only the signature (including name and parameters) of the method. The classes implementing the interface provide the full definition of these abstract methods. However, Java SE 8 has incorporated the concept of full implementation of default and static methods to Java interfaces. There are basically three types of interfaces that include the following:

**Top level interfaces** It is an interface that is not nested in any class or interface. It comprises a collection of abstract methods. It can contain any number of methods that are needed to be defined in the class.

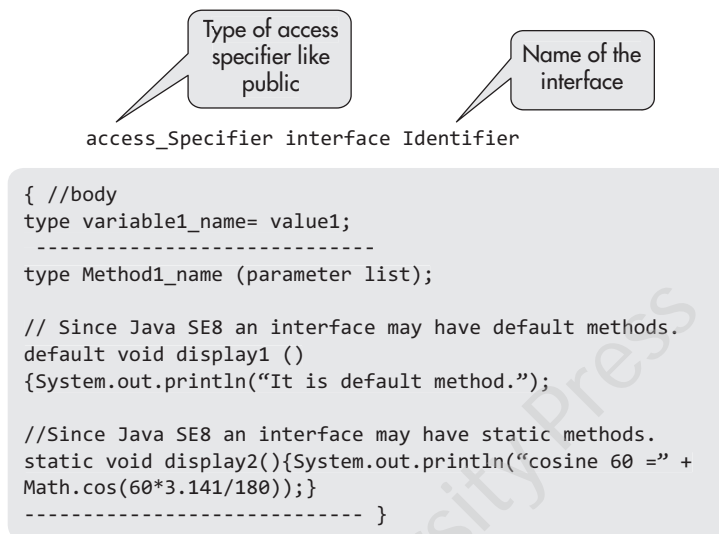
**Nested interface** It is an interface that is defined in the body of a class or interface. In nested interfaces, one or more interfaces are grouped, so that it becomes easy to maintain. It is referred to by the outer interface or class and cannot be accessed directly.

**Generic interface** Like a class, an interface is generic if it declares one or more types of variables. It comprises methods that accept or return an object. Thus, we can pass any parameter to the method that is not of the primitive type.

## 9.2 Declaration of Interface

Declaration of an interface starts with the access modifier followed by keyword `interface` which is in turn followed by its name or identifier that is followed by a block of statements; these statements contain declarations

of variables and abstract methods. The variables defined in interfaces are implicitly public, static, and final. They are initialized at the time of declaration. The methods declared in an interface are public by default. An illustration of declaration of an interface is given.



### 9.2.1 Interface Modifiers

Access modifiers for an interface are generally public or no access modifier is used. By declaring it as public, interface can be used in any package and in any class.

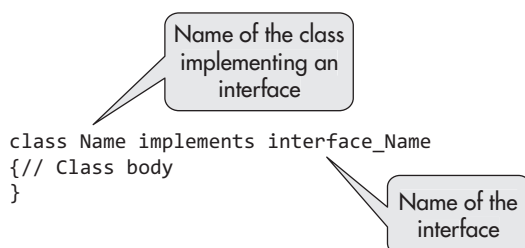
### 9.2.2 Members of Interface

The members of an interface comprise the following:

1. The members declared in the body of the interface.
2. The members inherited from any super interface that it extends.
3. The methods declared in the interface are implicitly public abstract member methods.
4. The field variables defined in interfaces are implicitly public, static, and final. However, the specification of these modifiers does not create a compile-type error.
5. The field variables declared in an interface must be initialized; otherwise, compile-type error occurs.
6. Since Java SE8, static and default methods with full definition can also be members of interface.

## 9.3 Implementation of Interface

The interfaces are implemented by classes. An illustration of declaration of class that implements an interface is as follows.





A class implementing an interface must provide implementation for all its methods unless it is an abstract class.

Class implementing multiple interfaces

```
class A implements C , D
{ // class body
}
```

Name of multiple interfaces

If a class extends another class as well as implements interfaces, it is declared as

```
class Name extends class_name implements Interface_name
```

Name of derived class

Name of super class

Name of interface

For example, a class A that extends class B as well as implements interfaces C and D is declared as

```
class A extends B implements C , D
{ // class body
}
```

For example, an interface by name SurfaceArea may be declared as

```
interface SurfaceArea {
double Compute (double x);
}
```

The implementation will determine the surface area for a figure for which definition is given in class. In Program 8.1, the aforementioned interface is implemented by two classes, one of which calculates the area of a circle, whereas another calculates the area of a square. The class Square is defined as

```
class Square implements SurfaceArea { // class body
}
```

Further, class Circle is declared as

```
class Circle implements SurfaceArea { // class body
}
```

The two classes include appropriate method definitions with header as given in the interface.

### Program 9.1: Illustration of interface to find areas of a square and a circle

```
1 interface SurfaceArea { // interface
2 double Compute (double x);
3 } // end of interface
4
5 class Square implements SurfaceArea // class Square
6 {public double Compute (double x)
7 {return (x*x);}
8 } // end of class Square
9
```

```

10 class Circle implements SurfaceArea // class Circle
11 {public double Compute(double x)
12 {return (3.141*x*x);}
13 } // End of class Circle
14
15 class Face{
16 public static void main(String arg[]){
17 Square sqr = new Square(); // object of class square
18 Circle cir1 = new Circle (); //class Circle object
19 SurfaceArea Area; // object reference of interface
20 // Assigning Square class reference to Area
21 Area = sqr;
22 System.out.println("Area of square =" + Area.Compute(10));
23 // Assigning Circle class reference to Area
24 Area = cir1;
25 System.out.println("Area of circle =" Area.Compute(10));
26 }
27 }

```

**Output**

```

Area of square = 100.0
Area of circle = 314.1

```

**Explanation**

The interface `SurfaceArea` declared in code lines 1–3 is implemented by two classes, that is, class `Square` defined in code line 5–8 and class `Circle` declared in code line 10–13. Each class has its own definition of the method. One finds the area of a circle and the other finds the area of a square. Both need only one parameter that is declared in the method header in the interface. The class `Face` is the class with main method that executes the two classes. The objects of these classes `sqr` and `cir1` are declared in code lines 17 and 18. The object references `Area` to interface `SurfaceArea` is declared in code line 19. In line 21, the reference `sqr` is assigned to `Area` and area of square with side 10 is obtained. In code line 24, reference `cir1` is assigned to `Area` and area of circle is obtained in line 25.

### 9.3.1 Constants in Interfaces

In Program 9.2, an interface defines only constant values that are implemented by classes.

**Program 9.2:** Illustration of getting constant values from interface

```

1 interface Dimensions{
2 int x = 30; // implicitly public and final
3 int y = 20; //implicitly public and final
4 }// End of interface Dimensions
5
6 class Room implements Dimensions
7 {
8 public int area(){
9 int m = x;
10 int n = y;
11 return (m*n);}
12 } // end of class Room
13
14 class Inface
15 {public static void main(String arg[]){
16 Room rm = new Room();
17 Dimensions d;
18 d = rm; // assigning Room reference to D
19 System.out.println("Area of room =" + rm.area());
20 }
21 }

```

**Output**

```
Area of room = 600
```

**Explanation**

Interface `Dimensions` declares two `int` values `x` and `y` that are initialized to 30 and 20 in code lines 2 and 3, respectively. These values are implicitly constant values. The values are used in class `Room` to calculate its area in code line 11. The class `Room` is executed in class `Interface` that has the main method and is declared in lines 14–21. The class defines an object `rm` of class `Room` and reference `d` to the interface `Dimensions`. The method `area` is invoked by the object `rm`. The output is given.



For declaring methods in an interface, method name must be chosen such that it indicates the purpose of the method in general. It may be possible that the method implemented by the class may not be related to that method name.

## 9.4 Multiple Interfaces

Like single interface, multiple interfaces can also be implemented in Java. For this, the class implements all the methods declared in all the interfaces. When the class is declared, names of all interfaces are listed after the keyword *implements* and separated by comma. As for example, if class `A` implements interfaces `C` and `D`, it is defined as.

```
class A implements C , D
{ // class body
}
```

Class implementing multiple interfaces

Name of multiple interfaces

### 9.4.1 Interface References

For interface references, variables can be declared as object references. In this case, the object reference would use interface as the type instead of class. The appropriate method is called on the basis of actual instance of the interface that is being referred to. In Program 9.3, two interfaces are declared. Method `show()` is called through interface reference using `t` variable that is declared to be of interface type `InterfaceA` and it is assigned as an instance of `MultiInterface`. Similarly, method `subtract()` is invoked through another variable `m` that is declared of the type `InterfaceB` and assigned as an instance of `MultiInterface`.



An interface reference can only access the methods declared in that interface. It cannot access other methods of the class implementing that interface.

In Program 9.3, the interface reference cannot access `display()` method of the class implementing the interfaces, as it is not declared in any of the interfaces.

#### Program 9.3: Illustration of implementing multiple interfaces

```
1 interface InterfaceA{ // interface declared
2 public void show(); // method declared
3 }
4 interface InterfaceB{ // another interface declared
5 public int method1 (int a, int b); // method declared
```

```

6     }
7     interface InterfaceC{
8     public double method2 (double x);}
9         // implements multiple interfaces
10    public class MultiInterfaceImpliment
11    implements InterfaceA, InterfaceB
12    {
13    public void show(){           // method show defined
14    System.out.println("Hello! It is java.");
15    }
16    public int method1 (int a, int b){ // method1 defined
17    return  a += b;
18    }
19    public void display()
20    {System.out.println("I cannot be called by interface references.\nOnly class object can
21    call me.");}
22    public static void main(String args[])
23    {
24    /*creating interface references and assigning class object*/
25
26    InterfaceA iA = new MultiInterfaceImpliment();
27    InterfaceB iB = new MultiInterfaceImpliment();
28
29    InterfaceC iCC = (double x) -> {return x*x*x;};
30    // invoking the methods
31    iA.show();
32    System.out.println("Value after processing a and b = " +iB.method1(25, 15));
33    System.out.println("Value after processing x = " +iCC.method2(5.0));
34    //iA.display(); error
35    //iB.display(); error
36    MultiInterfaceImpliment m=new MultiInterfaceImpliment();
37    m.display();
38    }

```

**Output**

```

Hello! It is java.
Value after processing a and b = 40
Value after processing x = 125.0
I cannot be called by interface references.
Only class object can call me.

```

**Explanation**

The program illustrates the implementation of multiple interfaces by a class. Three interfaces are declared. Each has one method. The class is declared as implementing two interfaces and the third becomes the target of Lambda expression because its method is defined inside the Lambda expression. This could have

been done in the normal way similar to the other two; however, the intention is to see the differences between the two ways of doing the same thing.

Interface `InterfaceA` is declared in lines 1–3. The second interface `InterfaceB` is declared in lines 4–6. The third interface `InterfaceC` is declared in lines 7–8. The implementing class with main method is declared in code lines 10–11. Method `show()` of the first interface is defined in class in code lines 13–15 and `method1()` of the second interface is defined in code lines 16–18. `method2()` of the third interface is defined, that is, body is placed inside the Lambda expression (line 28). It is not a regular definition of a method and compiler does not treat it as such. A method `display()` is declared in lines 19–20. It is a regular method of the class and regular member of class.

The methods of the first two interfaces are defined inside the class, and hence, we create references to the interfaces in lines 25 and 26 and these are assigned the class object reference. Further, methods are invoked with these references in lines 30 and 31 to get the results.

For method2() of the interface InterfaceC, the Lambda expression is defined in code line 28 that is implemented in line 32.

The regular method display() of the class cannot be called by interface references. They can call methods defined in their respective interfaces only. This method is called in line 36 by the class object defined in line 35. All the results of the three methods are given.

## 9.4.2 Stub Methods

As discussed earlier, if a class implements an interface, it has to define all its abstract methods; if we fail to do so, then it will make the implementing class an abstract class. In several cases, we do not need to define all the methods because we need only a few of these. In order to meet the requirement, we define *stub methods* for the methods we do not need. A stub method does not do anything; however, it fulfils the requirement. These are illustrated in Program 9.4.

In Program 9.4, an interface defines three methods, out of which, only one is needed. For the other two, stub methods are provided.

### Program 9.4: Illustration of definitions of stub methods

```

1  interface AreaVolume {
2      double surfaceAreaSphere (double radius);
3      double volumeShpere (double radius);
4      void setValue (double side);
5  }          // end of interface
6
7  public class StubMethod {
8      //Definition of stub method for surfaceAreaSphere
9      double surfaceAreaSphere (double radius)
10     { return 0; }
11     //Definition of stub method for setValue
12     void setvalue (double side) {}
13     // definition of method that is implemented
14     double volumeSphere (double radius)
15     {return 4.0*Math.PI* Math.pow(radius, 3)/3 ;}
16     // main method
17     public static void main(String[] args) {
18         StubMethod stm = new StubMethod();
19         System.out.printf("Volume of sphere of radius 10 = %.2f \n", stm.volumeSphere(10.0));
20     }
21 }

```

#### Output

Volume of sphere of radius 10 = 4188.79

#### Explanation

The program illustrates how to provide stub methods for the methods of interfaces that are not needed; however, as per the requirement of the implementing class, all the methods must be defined. The interface of this program defines three methods out which only one, that is, volumeShpere is needed. The other two, that is, surfaceAreaSphere and setValue are not needed. Therefore, we provide the following two stub methods for their definition.

- (i) double surfaceAreaSphere (double radius) { return 0; }
- (ii) void setvalue (double side) {}

These methods do not do anything but they simply fulfil the requirement. The output of the third method is given.

## 9.5 Nested Interfaces

An interface may be declared as a member of a class or in another interface. In the capacity of a class member, it can have the attributes that are applicable to other class members. For example, its access may be modified to public, protected, or private. In other cases, an interface can only be declared as public or with default (no-access modifier) access. Syntax of nested interface in another interface is given as

```
interface interface_Identifier
{
.....
.....
        interface nested_interface_Identifier
        {
.....
}
}
```

Program 9.5 illustrates an interface declared inside a class.

### Program 9.5: Illustration of nested interface in a class

```
1  class A {
2  public interface Nested {
3  int max(int x, int y);}
4  } // End of class A
5
6  class B implements A.Nested // implementing nested interface
7  {public int max (int x, int y )
8  {return x > y ? x: y ;}
9  } // End of class B
10
11 class X // Class with main method
12 {public static void main(String arg[])
13 { A.Nested NS = new B(); // creating object
14 System.out.println("Maximum of two numbers is = " + NS.max(30, 12));
15
16 }
17 }
```

#### Output

Maximum of two numbers is = 30

#### Explanation

A class A is declared in code lines 1–4 that has an interface by name Nested declared in the body of class A. The interface declares a method max with parameters int x and int y. You may have any other name of interface. The class B is declared to implement the interface declared in lines 6–9. The qualified name of interface is A.Nested. The class X has the main method that executes the two classes. In this class, reference of object of B is assigned to reference NS of A.Nested interface. NS calls the method max of class B with two integers and finds the greater of the two.

Program 9.6 illustrates an example of nested interface where an interface is declared within another interface.



**Program 9.6:** Illustration of nested interface in an interface

```
1 interface Shape{
2 double getArea(double x);
3 interface Display{
4 public void show();
5 }
6 }
7 class NestedInterface1 implements Shape, Shape.Display
8 {
9 double getArea(double x)
10 {
11 double area = 3.14*x*x;
12 return area;
13 }
14 public void Display.show()
15 {
16 System.out.println("Hello");
17 }
18
19 public static void main(String args[])
20 {
21 Shape.Display d=new NestedInterface1(); // reference to NestedInterface1
22 Shape s = new NestedInterface1(); // reference to class
23 d.show();
24 System.out.println("Area is = " +s.getArea(4.0));
25 }
26
27 }
```

**Output**

```
Hello
Area is = 50.24
```

**Explanation**

In line 1, interface named Shape is declared. Method getArea() is declared in line 2. Another interface, that is, the nested interface is declared in line 3. Within this interface, method Show() has been declared in line 4. In line 7, class is declared that implements the interface Shape and nested interface Display. For referring to this nested interface, Shape.Display is used. In line 9, the method getArea() is defined and the keyword return is used in line 12 that returns the result of area calculated followed by closing curly brackets. Method Show() is defined in lines 14–17. In line 21, the variable d is declared of the type Shape.Display that refers to the nested interface and it is assigned an instance of NestedInterface1 class. Similarly, for the s variable defined in line 22. Show() method is accessed in line 23. In line 24, the area is displayed.

Program 9.7 is another example of nested interfaces.

**Program 9.7:** Another example of nested interface

```
1 interface Demo{
2 interface NestedA{
3 public void show();
4 }
5 public interface NestedB{
6 int subtract(int a, int b);
7 }
8 }
9 public class NestedInterface2 implements Demo.NestedA, Demo.NestedB
10 {
11 {
12 public static void main(String args[])
```

```

13  {
14  Demo.NestedA a = new NestedInterface2();
15  a.show();
16  Demo.NestedB b = new NestedInterface2();
17  System.out.println("Value after subtraction is = " +b.subtract(9,2));
18
19  public void show()
20  {
21  System.out.println("Hello");
22  }
23  public void subtract()
24  {
25  return (a-b);
26  }
27  }

```

**Output**

```

Hello
Value after subtraction is = 7

```

**Explanation**

In line 1, interface is declared by the name `Demo`. Within this interface, another interface is declared by the name `NestedA` in line 2. In line 3, method is declared for this nested interface. In line 5, another nested interface is declared. In line 9, class implementing both the nested interfaces is declared. In line 14, variable `a` is declared of the type `Demo.NestedA` interface and assigned an instance of `NestedInterface2` class. In line 15, `Show()` method of `NestedA` interface is accessed. Similarly, another variable of type `Demo.NestedB` is declared in line 16. In line 17, method `subtract()` is accessed and the output is printed. Lines 19–22 define `Show()` method declared in interface `NestedA`. In lines 24–27, method `subtract()` is defined.

## 9.6 Inheritance of Interfaces

An interface can also extend an interface. The code for extension of interface is similar to the extension of classes (see Chapter 8). Here interface `C` extends interface `B` and interface `A`. Interface `C` is defined as

```

interface A {}
interface B {}
interface C extends B, A
{ // Body
}

```

Program 9.8 illustrates the inheritance of interfaces and implementation by class.

**Program 9.8:** Illustration of interface extending another interface

```

1  interface One {double Pi = 3.141;}
2
3  interface Two extends One // extending interface
4  {double radius = 10.0;}
5  interface Three extends One, Two
6  {double area ();}
7  class Circle implements Three // implementing interface
8  { public double area (){ return Pi*radius*radius;}
9  }
10
11 class Interface // class with main method

```

```

12     { public static void main(String args[])
13     {
14     Circle c = new Circle (); // creating an object
15     System.out.println( "Area of circle c =" + c.area());
16     }
17     }

```

**Output**

Area of circle C = 314.1

**Explanation**

Interface `one` defines a constant `Pi` and is inherited by interface `Two`, which defines a double value `radius`. The interfaces `one` and `two` are inherited by interface `Three`, which has an abstract method `area`. The class `Circle` declared in code lines 7–9 implements interface `Three`. The class `Interface` declared in lines 11–17 executes the class `Circle`. The output is given.

## 9.7 Default Methods in Interfaces

The *Java SE8* enhancement of interfaces allows the interfaces to have full definitions of default and static methods. The reason for this enhancement is given.

A class that implements an interface must define all the abstract methods of the interface. Now, if at a later stage, it is required to introduce another abstract method to the interface, then all the classes implementing the interface must be modified. To overcome this problem, the following two options are available in the existing (before Java SE8) arrangement.

1. Declare another interface that inherits the present interface and define the new method in it. Thus, only the classes that require the new method will have to be modified to implement the new interface.
2. The second option is to declare an abstract class with all the methods of the existing interface and the full definitions of new methods. In such a case, there is minimum change as all the subclasses inherit the abstract super class.

The enhancement in Java 8 spares the programmer of doing any change in the existing classes that implement the interface by allowing the interface to have full definition of default methods and static methods that are implicitly inherited by the class implementing the interface. The benefit of using the interface over having abstract superclass is that it allows the class to have another super class because in Java, there can be only one super class. Thus, by this enhancement, the existing framework is totally undisturbed, and at the same time, the new functionality can be added to the interface, which is inherited by classes implementing the interface. The inherited methods are also members of the class, and therefore, these may be called other methods of class.

A default method is declared with keyword *default* as

```

public interface A {
    default void display () {System.out.println("It is
interface A.");}
}

```

Keyword *default*  
for specifying  
default method

However, there are restrictions to this arrangement.

1. If a class implements two or more interfaces, the interfaces cannot have default methods with the same signature because this would cause ambiguity as to which one to execute. This is illustrated in Program 9.8.
2. In the case of methods with the same name, the compiler would choose by matching parameters.

- If the class that is extending the interfaces also defines the method with the same name, then class definition has priority over other definitions.
- If an interface A is extended by interface B and both have a default method with the same name, the definition in interface B would be chosen. However, the programmer can specify which one to choose by using `super` keyword as

```
A.super.method_name();
```

This is shown in Program 9.9

- A default method cannot be declared `final`. This is illustrated in Program 9.10.
- A default method cannot be synchronized; however, blocks of statements in the default method may be synchronized.
- The object class is inherited by all classes. Therefore, a default method should not override any non-final method of object class.

### Program 9.9: Illustration of default method in interface

```

1  interface InFace
2      {int number =10;
3          //default method
4  default void display(){System.out.println ("2 to the power 8 =" + Math.pow(2,8));}
5      }
6
7  public class DefaultMethod implements InFace{// implementing the interface
8      public static void main(String[] args) {
9  DefaultMethod df = new DefaultMethod(); // creating object
10
11     int area = number* number;
12     System.out.println("area =" +area );
13     df.display();
14     }
15 }
```

#### Output

```
area = 100
2 to the power 8 = 256.0
```

#### Explanation

The interface `InFace` is declared in code lines 1 and 2. In the interface, an `int` number is defined and a default method with full definition is also defined. This is now permissible in Java SE8—an interface may have full definitions of static methods and default methods. The default method is declared with modifier `default`. Line 9 defines a new object of the present class. Since default methods are inherited by the class, it is simply called by the object of the class in code line 13. The integer number is used in line 11.

In Program 9.10, two interfaces are implemented by a class, and the interface has default methods with the same signature.

### Program 9.10: Class implementing interfaces with default method

```

1      interface InfaceA{
2      public void showA();
3      default public void display()
4      {
5      System.out.println("Good morning to everyone");
6      }
7      }
8      interface InfaceB {
```

```
9     public void showB();
10    default public void display()
11    {
12    System.out.println("Good bye to everyone");
13    }
14
15    class DefaultMethodA implements InfaceA, InfaceB
16    {
17    public static void main(String args[])
18    {
19    public void showA()
20    {
21    System.out.println("It is interface A");
22    }
23    public void showB()
24    {
25    System.out.println("It is interface B");
26    }
27    InfaceA a = new DefaultMethodA();
28    a.display();
29    a.showA();
30
31    InfaceB b = new DefaultMethodA();
32
33    b.display();
34    b.showB();
35    }
36    }
```

**Output**

compilation error

**Explanation**

In line 1, interface `InfaceA` is declared. In line 2, method `showA()` is declared. Lines 3–6 define default method `display()`. In line 8, another interface `InfaceB` is declared. Line 9 declares another method `showB()`. In line 10, `display()` method for this interface is defined having the same signature as that of the default method defined in interface `InfaceA`. Line 13 defines the class implementing both the interfaces. In line 19, method `showA()` for interface `InfaceA` is defined. Similarly, in line 23, method `showB()` is defined. Variable `a` of the type `InfaceA` is declared in line 27 and assigned an instance of class `DefaultMethodA`. In line 28, method `display()` is accessed. Similarly, another variable of type `DefaultMethodA` is declared in line 31. Compilation error results because the default method of the same signature is used in both the interfaces.

Program 9.11 shows the case where one of the interfaces is extended by another interface, and both having default method with the same signature.

**Program 9.11:** Illustration of inheritance of interfaces by using default method

```
1     interface InfaceA{
2     public void showA();
3     default public void display()
4     {
5     System.out.println("Good morning to everyone");
6     }
7     }
8     interface InfaceB extends InfaceA{
9     public void showB();
10    default public void display()
11    {
```

```

12 System.out.println("Good bye to everyone");
13 }
14 class DefaultMethodB implements InfaceB
15 {
16 public static void main(String args[])
17 {
18 public void showA()
19 {
20 System.out.println("It is Interface A");
21 }
22 public void showB()
23 {
24 System.out.println("It is Interface B");
25 }
26 DefaultMethodB d = new DefaultMethodB();
27 d.showA();
28 d.display();
29 d.showB();
30 }
31 }

```

**Output**

Good morning to everyone

**Explanation**

In line 1, interface `InfaceA` is declared. The method of this interface is declared in line 2. Default method `display()` is defined in lines 3–6. In line 8, interface `InfaceB` extends

interface `InfaceA`. Method `ShowB()` of this interface is declared in line 9. Default method `display()` is defined in lines 10–13. Line 14 defines the class implementing both the interfaces. In lines 18–21, method `ShowA()` is defined. Similarly, in lines 22–25, method `showB()` is defined. In line 26, variable `d` of the type `DefaultMethodB` is declared. In line 27, method `ShowA()` is accessed. In line 28 `display()` method is accessed for the interface `InfaceA`. The output is shown.

Program 9.12 shows that a default method cannot be declared as `final` and gives compilation error.

**Program 9.12:** Default method declared as `final`

```

1 interface A{
2 public void show();
3 final default public void display() // error due to final
4 {
5 System.out.println("Good morning to everyone");
6 }
7 }
8 class DemoC implements A
9 {
10 public void show()
11 { System.out.println("It is Interface");
12 }
13
14 public static void main(String args[])
15 {
16 DemoC d = new DemoC();
17 d.show();
18 d.display();
19 }
20 }
21

```

**Output**

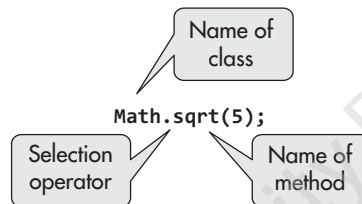
compilation error

## 9.8 Static Methods in Interface

The Java version 8 allows full definition of static methods in interfaces. A static method is a class method. For calling a static method, one does not need an object of class. It can simply be called with class name as

```
class_name.method_name()
```

For example, we have used the method `sqrt()` of `Math` class by writing the code as in the following example, because all methods of `Math` class are declared static.



A class that implements an interface also inherits all its static methods. The inherited method is also a class method, and therefore, the method can simply be called by using class name as illustrated.

### Program 9.13: Illustration of static method in interface

```

1  interface InFace1{           // defining an interface
2  int number =1000;
3  static int compute (int num) // static method defined
4  {int cube = num * num * num ;
5  System.out.println( "Cube of" + num + " = " + cube );
6  return cube ;}
7  }
8  // implementing an interface
9  public class StaticMethod implements InFace1
10
11  {
12  public static void main(String[] args) {
13  StaticMethod sm = new StaticMethod(); // creating object
14  System.out.println("Cube root of 1000 = " + Math.cbrt(number));
15  //Using number defined in interface
16  InFace1.compute(10);        // calling the static method
17  }
18  }
```

**Output**

Cube root of 1000 = 10.0  
Cube of 10 = 1000

**Explanation**

The interface `InFace1` is defined in code line 1. The interface defines an integer number `1000` that is used in code line 14 for finding its cube root. The interface also contains a static method

`compute ()`, which computes the cube of a number with full definition. This is allowed in Java SE 8. This method is called in the class with reference of name of interface `InFace1`. The output is shown.

## 9.9 Functional Interfaces

In Java SE8, a new package `java.util.function` on functional interfaces has been introduced to serve as the basis for writing Lambda functions. Functional interfaces are interfaces with one abstract method. They are also called SAM or single abstract method type. However, a functional interface can have more than one static and default methods besides the abstract method and it can override some methods of object class. The programmer may include an annotation, which is given in the following example, in order to lessen the work of compiler that anyway will recognize a functional interface.

```
@FunctionalInterface
```

By adding the aforementioned annotation, it can be helpful in detecting compile time errors. If the functional interface contains more than one abstract method, the compiler will throw an error. As we all know, Java programming language is purely an object-oriented language and an object can call only the methods encapsulated in the class for the object. There is no independent existence of methods in Java. Methods are like bonded labourers. However, there are several occasions when a method is required to be passed as argument to another method. In order to overcome this limitation of Java language, functional interfaces have been defined in package `java.util.function` with only one method to serve as target for Lambda functions or expressions. A partial list of these is given in Table 9.1. The letters T, U, R, etc., represent type parameters.

### Program 9.14: Illustration of uses of functional interfaces

```
1  import java.util.function.Function;           //importing classes
2  import java.util.function.BinaryOperator;
3  public class BiOperator {
4  public static void main(String args[]){
5  Function <Double, Double>logrithm = Math::log;
6  // method reference assigned to Function
7  System.out.println( "log of 10 to the base e = " + logrithm.apply(10.0));
8  // method reference assigned to BinaryOperator
9  BinaryOperator<Integer> minimum = Math::min;
10
11 System.out.println ("Minimum of 20 and 46 is " + minimum.apply(20, 46));
12 }
13 }
```

#### Output

```
log of 10 to the base e = 2.302585092994046
Minimum of 20 and 46 is 20
```

#### Explanation

Out of the list of functional interfaces given in Table 9.1, the program uses two of them, that is,

- (a) `function (Double, Double)`
- (b) `binaryOperator (Integer)`

The first has been assigned the reference of `Math::log` and the second is assigned the reference of method `Math::min`. The first is used in line 7 to find the logarithm of 10 to the base e. Further, the second is used to find the minimum of two specified numbers in line 11. The output is shown.

Single argument functional interfaces that return Boolean output are used for filtering with a specified test condition. These are discussed in detail in Section 6.13 in Chapter 6.



**Table 9.1** Some standard functional interfaces of package `java.util.function`

<i>Interface</i>	<i>Description</i>
<code>BiConsumer&lt;T,U&gt;</code>	Stands for methods that accept two arguments and return no value
<code>BiFunction&lt;T,U,R&gt;</code>	Stands for methods that accept two arguments and return a result
<code>BinaryOperator&lt;T&gt;</code>	Stands for methods/operation that accept two operands of the same type and produces a result of the same type
<code>BiPredicate&lt;T, U&gt;</code>	Stands for predicate method that accepts two arguments and produces a Boolean type result
<code>BooleanSupplier</code>	Stands for supplier of Boolean type values
<code>Consumer &lt;T&gt;</code>	Stands for methods that accept single argument and produce no result
<code>DoubleBinaryOperator</code>	Stands for operation on two double valued operands resulting in a double valued result
<code>DoubleConsumer</code>	Stands for operation that accepts single double-type argument and produces no result
<code>DoubleFunction&lt;R&gt;</code>	Stands for methods that accept double-type argument and produce a result
<code>DoublePredicate</code>	Stands for predicate methods that accept one <code>double</code> argument and give Boolean output
<code>DoubleSupplier</code>	Stands for methods that supply double values
<code>DoubleToIntFunction</code>	Stands for functions that accept <code>double</code> type input and result in <code>int</code> type value output
<code>DoubleToLongFunction</code>	Stands for functions that accept <code>double</code> type input and result in <code>long</code> type value output
<code>DoubleUnaryOperator</code>	Stands for operation on a single <code>double</code> type operand that yields a <code>double</code> type result
<code>Function &lt;T, R&gt;</code>	Stands for function that accepts one input object and yields appropriate object as output
<code>IntBinaryOperator</code>	Stands for operation on two <code>int</code> operands that gives <code>int</code> type result
<code>IntConsumer</code>	Stands for operations that accepts a single <code>int</code> type argument and produce no result
<code>IntFunction &lt;R&gt;</code>	Stands for function that accepts an <code>int</code> type argument and produces a result
<code>IntPredicate</code>	Stands for predicates with one <code>int</code> type argument
<code>IntSupplier</code>	Stands for method that supplies <code>int</code> type values
<code>IntToDoubleFunction</code>	Presents a function that accepts <code>int</code> input and results in <code>double</code> type value output
<code>IntToLongFunction</code>	Stands for function that accepts <code>int</code> type input and results in <code>long</code> type value output
<code>IntUnaryOperator</code>	Stands for operation on a single <code>int</code> type operand, which yields an <code>int</code> type result
<code>LongBinaryOperator</code>	Stands for operation on two <code>long</code> type operands, which give a <code>long</code> type result
<code>LongConsumer</code>	Stands for operations that accepts a single <code>long</code> type argument but results in no output
<code>LongFunction &lt;R&gt;</code>	Stands for function that accepts <code>long</code> type argument and produces a result
<code>LongPredicate</code>	Stands for predicates with one <code>long</code> type argument

(Contd)

**Table 9.1** (Contd)

Interface	Description
LongSupplier	Stands for supplier of long type values
LongToDoubleFunction	Stands for functions that accept a long type argument and produces a double type result
LongToIntFunction	Stands for functions that accept long type input and results in int type output
LongUnaryOperator	Stands for operations on a single long type operand that yields a long type result
ObjDoubleConsumer <T>	Stands for operations that accept an object value and a double type argument but return no output
ObjLongConsumer <T>	Stands for operations that accepts an object value and a long type argument and returns no result
Predicate <T>	Stands for predicate that accepts one input and determines if the input object satisfies some criteria
Supplier <T>	A supplier of objects of type T
ToDoubleBiFunction <T, U>	Targets the functions that accept two arguments and produce a double output
ToDoubleFunction <T>	Stands for function that results in double type result
ToIntBiFunction <T, R>	Stands for the functions that accept two arguments and produce an int type output
ToIntFunction <T>	Stands for functions that produce an int type result
ToLongBiFunction <T, U>	Targets the functions that accept two arguments and produce a long type output
ToLongFunction <T>	Stands for function that gives long type result
UnaryOperator <T>	Stands for operations/methods on single operand and gives results of the same type

The described functional interfaces have one abstract method and provide the target type for Lambda functions and method references that are discussed in Chapter 8. We take a few examples of use of some of these functionals in Program 9.15.

**Program 9.15:** Illustration of application of predicates in filtering arrays

```

1  import java.util.function.Predicate; // importing the class
2  public class Predicate2 {
3  public static void main(String[] args) { // main class
4
5  Predicate<String>str = Predicate.isEqual("Delhi");
6  // using Predicate to test for "Delhi"
7
8  System.out.println(str.test("Katty"));
9  System.out.println(str.test("Mahma"));
10
11
12  String [] names={"Indore", "Aligarh", "Patiala" ,"Delhi"};
13
14  for(inti =0; i<names.length; i++) // for loop
15  if (str.test (names[i]))
16  System.out.println(names[i]);

```

```

17
18     for(String s: names)                // filtering the array
19         if (!str.test(s))
20             System.out.print(s + " ");
21     System.out.println();
22     }
    }

```

**Output**

```

false
false
Delhi
Indore   Aligarh   Patiala

```

**Explanation**

The program illustrates the method `isEqual` applied with a predicate. It is used to compare strings with string “Delhi” to test if they are equal. The first code line imports the standard library interface `predicate`, which is of the following form.

```
Predicate<T>
```

It is defined with method `isEqual(“Delhi”) in line 5 and applied in code lines 8 and 9 to see if “Katty” or “Mahma” is equal to Delhi. Obviously, it is false and this result is displayed in the first two lines of the output.`

The method is also used to select “Delhi” from an array of strings through code lines 14–16. The same predicate may be used to filter the array as done in code lines 18–21. The output comprises strings of the array names without “Delhi”.

Program 9.16 illustrates the application of `IntBinaryOperator`, `DoubleBinaryOperator`, and `BiFunction` <T, U, R>. The Lambda expression is used to define the methods.

**Program 9.16:** Illustration of application of some functional interfaces

```

1  import java.util.function.Function;
2  import java.util.function.IntBinaryOperator;
3  import java.util.function.DoubleBinaryOperator;
4  import java.util.function.BiFunction;
5
6  public class Functional1 {
7      public static void main(String args[]){
8
9          IntBinaryOperator ibo = (int m, int n) -> { return (m*m + n*n)};
10
11         System.out.println(“The resultant value = ” + ibo.applyAsInt(20, 4));
12         DoubleBinaryOperator dbo = (double a, double b) -> { return Math.pow(a, b)};
13         System.out.println(“Three to the power four = ” + dbo.applyAsDouble(3.0, 4.0));
14         BiFunction<Integer, Integer, Boolean> bif = (x, y) -> {return x.equals( y)};
15         System.out.println(“Is 45 equal to 30 ? ” + bif.apply(45, 30));
16     }
17 }

```

**Output**

```

The resultant value = 416
Three to the power four = 81.0
Is 45 equal to 30 ? false

```

**Explanation**

Lambda expressions are used to define the methods for the three functionals. For the first functional, a reference is created in code line 9 as `ibo` and is assigned the method reference. `ibo` is used in code line 11 and gets the value returned by the method. The body of the method for the first one is defined as `{return (m*m + n*n)}`. It returns the sum of squares of two integers. For the arguments 20 and 4, the output is obviously 416.

For the second, it is defined in code line 12 as a raised to the power  $b$ . For the arguments 3.0 and 4.0, the result is  $3.0^{4.0}$  that is 81. For the `Bifunction(Integer, Integer, Boolean)`, the two integers 45 and 30 are compared for equality. The output is a Boolean that is obviously false. The results are given.

### 9.9.1 Functional Consumer<T>

The interface declaration is

```
@FunctionalInterface
public interface Consumer {void accept(T t);}
```

It declares one abstract method `void accept(T t)`. The method only consumes its argument. It does not give any return value. Hence, it is of `void` type. The methods that may be defined are printing methods or setting methods that set the values but do not return any value. Program 9.17 illustrates its application.

#### Program 9.17: Illustration of application of functional interface `Consumer<T>`

```
1  import java.util.function.Consumer;
2  public class Functional2 {
3
4  public static void main(String[] args) {
5  Consumer <Double> consumer=(Double d) -> { display(d);};
6  consumer.accept(3.14159);
7
8  Consumer<String> consumer1=(String s) -> {display(s);};
9  String [] sray ={"Akriti", "Kirin", "Lata", "Sunita"};
10
11 System.out.println("\nDisplay of Consumer1.");
12     for ( String str :sray )
13         consumer1.accept(str);
14
15 System.out.println("\nDisplay of Consumer2");
16 Consumer<Integer> consumer2=(Integer n) -> {display(n);};
17 Integer [] xray = {5,6,8,9,2};
18
19     for (Integer x :xray)
20         consumer2.accept(x);
21         System.out.println();
22     }
23     // Generic definition of display method
24     public static<T> void display(T t) {
25         System.out.print( t + " ");
26     }
27 }
```

#### Output

```
3.14159
Display of Consumer1.
Akriti Kirin Lata Sunita
Display of Consumer2
5 6 8 9 2
```

#### Explanation

The program defines a generic method `display()` in code lines 24–26, which is used to display any type of object. The `Consumer` interface has only one method `accept()`, which takes only a single argument. Code line 5 defines a reference of the interface with name `consumer` in which method `display()` consumes the object, which is a double value. This forms the first output.

Code line 8 defines another reference `consumer1` that displays `String` values. It can consume `String` objects. At a time, it takes only one string, and therefore, for consuming an array, a *for-each* loop is used in lines 12 and 13.

Code line 16 defines a consumer reference `consumer2` for consuming `Integer` values. It takes one integer value at a time. For display of an array of integers, a *for-each* loop is used in lines 19 and 20. All outputs are shown.

## 9.9.2 Functional Supplier

The functional `Supplier` is the opposite of consumer. It simply supplies an object through its method `get()`, which does not take an argument. The method can be used in such a situation where there is no input but there is output. The declaration of functional is as follows.

```
@FunctionalInterface
public interface Supplier {
    T get();
}
```

Program 9.18 illustrates the application of the functional.

### Program 9.18: Illustration of application of functional `Supplier`

```
1  import java.util.Random;
2  import java.util.function.Supplier;
3  class MyNumber {
4  public int getNumber(){
5  return (int)(Math.random()*100);
6  }}
7  public class Functional3 {
8  public static void main(String[] args) {
9  Supplier<MyNumber> supplier = MyNumber::new;
10 MyNumber number = supplier.get();
11 System.out.println(number.getNumber());
12 }
13 }
```

#### Output

23

#### Explanation

The program illustrates the application of functional `Supplier<T>`. A class `MyNumber` is declared in code lines 3–6. A member of the class generates random numbers. Another class with method `main` is declared in lines 7–13. The reference `supplier` of interface `Supplier` is defined in code line 9 and is assigned reference of class object. The reference is used to invoke the `get` method of `Supplier` as used in code line 10, which is displayed in line 11. The output is given.

## 9.10 Annotations

Annotation framework in Java language was first introduced in Java 5 through a provisional interface `APT`; however, it was formally introduced in `Javac` compiler in Java SE 6. It is a type of metadata that can be integrated with the source code without affecting the running of the program. Comments are also a way of adding information in a source code without affecting the implementation; however, comments are simply neglected at compile time and do not go further. The comments are introduced only for the user/programmer for understanding the program. However, annotations may be retained up to runtime and may be used to instruct the compiler and runtime system to do or not to do certain things.

Since Java SE 8, the annotations may be applied to classes, fields, interfaces, methods, and type declarations like throw clauses. The annotations are no longer simply for metadata inclusion in the program but have become a method for user's communication with compiler or runtime system.

For example, consider the case of a super class method that is overridden in a subclass. In such a case, both the method definitions should have the same signature. If, by mistake, a type or a parameter in subclass definition is changed, the method is no longer overridden, instead it becomes an overloaded method. The program will still compile without any error but the result will not be as expected. Besides, it is difficult to catch such an error. The remedy is to add the following annotation before the method definition in subclass.

```
@Override
```

This will cause the compiler to check and report if the method is really overridden or it is simply overloaded. Thus, it helps the programmer to catch and correct the error at compile time.

An annotation like `@Override` consists of two distinct words `@` and `Override`. It does not matter if there is a white space between the two. It may as well be written as `@ Override`; however, the general convention is not to have a gap. The name `Override` is the name of the interface that defines the annotation. There are a number of annotations that are predefined and are part of the package `java.lang.annotation`. However, a programmer may also define an annotation appropriate for his/her program. Annotations are defined by interfaces that are preceded by the tag character `@`. Thus, annotations can be easily recognized by symbol `@` in the code. Java language library has several predefined annotations in its library like the one illustrated. These annotations are suitable for a majority of programming requirements. However, Java language has provisions for user-defined annotations as well as plug-in third party developments.

Program 9.19 illustrates the application of `@Override` annotation.

### Program 9.19: Illustration of application of annotation `@Override`

```

1  class XX {
2  public void display(){System.out.println("This is class XX.");
3  } // end of class XX
4
5  class YY extends XX
6  {@Override public void display(){System.out.println("This is class YY.");
7  } // end of class YY
8
9  class ZZ extends XX
10
11 { @Override
12 public void display(){System.out.println("This is class ZZ.");
13 } // end of class ZZ
14 // below is class with main method
15 public class OverrideSuper {
16 public static void main (String Str[])
17 {
18     XX objX = new XX();
19     YY objY = new YY();
20     ZZ objZ = new ZZ();
21
22     objX.display();
23     objY.display();
24     objZ.display();
25 }
26 }
```

#### Output

```
This is class XX.
This is class YY.
This is class ZZ.
```

#### Explanation

The super class `XX` defines a method `Display` and the two sub classes `YY` and `ZZ` override this method. It is the information for the compiler to check whether type signature of the three methods is the same as given in `XX`. In the program,

the three definitions have the same signature, and therefore, there is no compile error by the compiler and the program runs successfully. The output is given.

### 9.10.1 Benefits of Using Annotations

1. It provides useful information to the compiler for detecting errors.
2. The information may also be used for suppressing warnings.
3. Annotations may be used for generating code in xml.
4. Annotation may be retained by another annotation for processing at runtime.
5. It can carry metadata up to runtime and the information may be obtained at runtime.

### 9.10.2 Annotation Basics

An annotation is an interface that declares only methods and is preceded by symbol @. An annotation may be defined with no methods or with one or more methods.

1. An annotation with no methods is declared as

```
@interface MyAnnotation{}
```

2. An annotation with one method may be declared as

```
@interface MyAnn{
    int value();
}
```

The method is treated as a field and may be assigned a default value as

```
@interface MyAnn{
    int value() default 0; }
```

3. An annotation with more than one member method and with default values may be declared as

```
@ interface MyAnno {
    int value1() default 10;
    double value2() default 0.0;
    String value3() default " ";
}
```

The methods included in an annotation do not have a body. They are more like fields and values are assigned either in an interface or class. An annotation with two methods is.

```
@ interface TwoAnno {
    int value1() default 0;
    double value2() default 0.0 ;}
```

The values may be assigned as

```
class AnnoValue {
    @TwoAnno { value1 = 100, value2 = 20.45}
    public static void Method1() { /* statements*/}
```

Note that the values are assigned to the names of methods, and no parentheses are used. In the aforementioned case, annotation TwoAnno is connected with method1.

### 9.10.3 Retention Policy

It specifies how long the annotation can be retained in the program. The property is imparted by another annotation called `@Retention`. Three retention options (constants) are defined in the language.

**Source** With this retention policy, the annotation is retained in the source file and is discarded at compile time. It is just like a comment.

**Class** With this policy specification, the annotation goes up to the `.class` file. It is a part of the compiled program but it does not go further. It is not available at runtime.

**Runtime** With this retention policy, the annotation is accessible throughout the running of the program. One can get the method values of the annotation if it has any.

The retention policy is declared through an argument of predefined annotation `@Retention`. The following example illustrates the specification.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnn{
    int value() default 0;
}
```

The policy applies to the following annotation.

### 9.10.4 Predefined Annotations

A number of annotations were introduced in Java 1.6. New annotations have been added in Java 8 as well. There are two types of annotations.

1. The annotations that directly apply to the code.
2. The annotations that specify some property of other annotations, that is, they apply to the annotations.

Some of the predefined annotations that are generally used are described as follows:

1. `@Deprecated`: It tells the compiler that the targeted code may be class, field, methods, or if any other feature has been superseded by a newer code.
2. `@Documented`: If an element is marked by this annotation, the Javadoc utility outputs it to the documentation file that it creates.
3. `@Encrypted`: It is annotation of security annotation framework (SAF) for encryption of write and decryption of read operation.
4. `@FunctionalInterface`: The annotation tells the compiler that it is a functional interface—an interface with one abstract method. This annotation has been added since Java 8.
5. `@NotNull`: Added since Java 8. It informs the compiler that annotated element must not be null.
6. `@Override`: It tells the compiler that the targeted method overrides a super class method.
7. `@Repeatable`: It specifies that the annotation may be applied more than once.
8. `@SuppressWarnings`: This annotation instructs the compiler not to issue a warning, if otherwise it would.
9. `@SafeVarargs`: The annotation conveys to the compiler that the code is safe, and therefore, some of the tests may be skipped for increasing the speed.
10. `@Retention`: It is the annotation that specifies the length of retention of the targeted annotation. There are three options that are described in Section 9.10.3.
11. `@Target`: When a user constructs own annotation, it is better to tell the compiler about its target elements. The targets are described in Table 9.2. Its application is illustrated which specifies the targets as `METHOD` and `FIELD`.



```
@Target(ElementType.METHOD, ElementType.FIELD)
```

**Table 9.2** Target elements

Target	Description
ElementType.ANNOTATION_TYPE	It is applicable to an annotation.
ElementType.CONSTRUCTOR	It is applicable to a constructor.
ElementType.FIELD	It is applicable to field or property
ElementType.LOCAL_VARIABLE	It is applicable to a local variable.
ElementType.METHOD	It is applicable to a method.
ElementType.PACKAGE	It applies to a declaration of package.
ElementType.PARAMETER	It applies to a parameter of a method.
ElementType.TYPE	It applies to a type declaration.
@Inherited	It specifies that the target annotation is inherited.

## 9.10.5 Class Method

The Java language has provided a facility so that its code can enquire about itself. This is provided by classes and interfaces in the package `java.lang.reflect` that contains interface `Member` and several classes. The class `Method` is one of the classes of this package.

The methods of class `Method` are useful in dealing with annotations. It has several methods that can access annotations at runtime and get information that they carry. The declaration of class `Method` is as follows.

```
public final class Method extends AccessibleObject
implements GenericDeclaration, Member
```

### 9.10.5.1 Methods of Class Method

Some methods useful for the present section are given in Table 9.3.

**Table 9.3** Methods of class Method

Method	Description
<code>boolean equals(Object obj)</code>	Compares the Method object with specified object
<code>&lt;T extends Annotation&gt; T getAnnotation(Class &lt;T&gt; annotationClass)</code>	Returns this element's annotation if present; otherwise, null
<code>Annotation[] getDeclaredAnnotations()</code>	Returns all the annotations in an array
<code>Class&lt;?&gt; getDeclaringClass()</code>	Returns the class or interface reference that declares this Method object
<code>object getDefaultValue()</code>	Returns the default value of annotation member
<code>int hashCode()</code>	Returns hash code
<code>int getModifiers()</code>	Returns modifiers used with this Method object

(Contd)

**Table 9.3** (Contd)

Method	Description
String getName()	Returns name of method represented by this Method object
Class<?> getReturnType()	Returns a class object that represents the formal return type of the method
String toString()	Returns a string that describes the Method object

The methods inherited by class Method are as follows:

1. The class inherits all the methods of Object class.
2. The class inherits the methods of java.lang.reflect.AccessibleObject, which it extends. These methods are as follows:

```
getAnnotations(),    isAccessible(),    isAnnotationPresent(),
setAccessible(),    setAccessible()
```

## 9.10.6 User Defined Annotation

We have already discussed the basics of annotations and some predefined annotations. A programmer may construct own annotation and use it. In the following discussion, we shall take a few examples of constructing annotations and using them.

### 9.10.6.1 Marker Annotation

A marker annotation does not contain any member method. It simply marks a declaration. Its presence may be obtained by the method isAnnotationPresent().

Program 9.20 illustrates the declaration of Marker annotation and also the application of some of the methods of class Method.

#### Program 9.20: Illustration of working of Marker annotation and methods of class Method

```
1  import java.lang.reflect.Method;
2  import java.lang.annotation.*;
3
4  @Retention(RetentionPolicy.RUNTIME)
5  @Target(ElementType.METHOD)
6  @interface MarkerAnnotation1 {} // Marker annotation
7
8  class MAnno {
9  @MarkerAnnotation1 // Application of annotation
10 public void display(){System.out.println("The Marker Annotation has no methods.");}
11 }
12 public class MarkerAnno {
13 public static void main(String args[])throws Exception{
14     MAnno manno = new MAnno();
15     MarkerAnno markeranno = new MarkerAnno();
16     Method m = manno.getClass().getMethod("display");
17     System.out.println("The Name of method is " + m.getName());
18     System.out.println(m.toString());
19     if(m.isAnnotationPresent(MarkerAnnotation1.class))
20         System.out.println("The MarkerAnnotation is present.");
21     else
```

```

22     System.out.println("The MarkerAnnotation is not present.");
23
24     manno.display();
25     }}

```

**Output**

The Name of method is display  
public void javaapplication1.MAnno.display()  
The MarkerAnnotation is present.  
The MarkerAnnotation has no methods.

**Explanation**

The program illustrates Marker Annotation as well as the use of some methods of class Method in enquiry of the annotation. The Marker annotation is defined in code line 6. Its retention is up to runtime. It is used in line 9 in class Manno. The class with main method

is declared in line 12. Code lines 14–16 define the objects of the classes. The method getName() is used in line 17 and its output is the name of method display (see the first line of output). The second line of output is the result of code line 18, that is, output of method toString(). Code lines 19 and 20 test the presence of annotation and give the output displayed in the third line of output. The last line of output is the result of invoking method display() by the object manno of class Manno.

**9.10.6.2 Single Method Annotation**

The annotations declare only methods; however, these do not have bodies and the methods behave like fields. Program 9.21 illustrates the case when an annotation has a single method.

**Program 9.21:** Illustration of single method (value) user's annotation

```

1   import java.lang.reflect.Method;
2   import java.lang.annotation.*;
3
4   @Retention(RetentionPolicy.RUNTIME) //defines retention
5   @Target(ElementType.METHOD) // defines target
6   @interface Single { // declares annotation
7       double value () ;
8   }
9   class SingleData{
10      @Single(3.14159 ) // defines annotation value
11      public void display(){System.out.println("It is a user
12      annotation with single method.");}
13  }
14  public class SingleValue { // class with main method
15      public static void main(String args[])throws Exception{
16
17          SingleData data =new SingleData(); // declares object
18          Method m = data.getClass().getMethod("display");
19          Single single = m.getAnnotation(Single.class);
20          System.out.println("The value = "+ single.value());
21          data.display(); //the object data invokes method display
22      }
23  }

```

**Output**

The value = 3.14159  
It is a user annotation with single method.

**Explanation**

The program declares an annotation with single method, that is, double value (); . The name of the annotation is Single. The value assigned to method

is 3.34159 in code line 10. Note that there is no semicolon at the end of the annotation (see line 10). The class SingleData also declares a method display linked to the annotation and this is used for declaring object m of

class Method. For enquiring into the annotation, we have to have objects of classes `SingleData` and `Method` and reference of annotation `Single`. Code line 20 returns the value of the method of the annotation, which is displayed.

### 9.10.6.3 Metadata in Annotation

Program 9.22 illustrates the annotation that simply carries metadata that is retrieved at runtime. Three methods are declared in the annotation. All the three are of type `String`. The values carried by the annotation is the title of a book and names of its two authors.

#### Program 9.22: Illustration of metadata in annotation

```

1      import java.lang.reflect.Method;
2      import java.lang.annotation.*;
3
4      @Retention(RetentionPolicy.RUNTIME)
5      @Target(ElementType.METHOD)
6      @interface MetaData {    // declaration of annotation
7      String title() ;
8      String author1() default "";
9      String author2() default "";
10     }
11     class AnnoData{
12     @MetaData(title = "Programming with Java", author1 = "B. L. Juneja", author2 =
13     "Anita Seth")
14     public void display(){System.out.println("It is a user annotation."); }
15     }
16     public class AUserAnnotation {
17     public static void main(String args[])throws Exception{
18
19     AnnoData data =new AnnoData();
20     Method m = data.getClass().getMethod("display");
21     MetaData md = m.getAnnotation(MetaData.class);
22
23     System.out.println("The Title is:" + md.title());
24     System.out.println("The author1 is:" + md.author1());
25     System.out.println("The author2 is:" + md.author2());
26     data.display();
27     }
28     }

```

#### Output

```

The Title is: Programming with Java
The author1 is: B. L. Juneja
The author2 is: Anita Seth
It is a user annotation.

```

#### Explanation

This program is similar to Program 9.21. Both have multi-method annotations. In the present program, the data supplied is of type `String` and concerns a book and its authors. It is like metadata. Code line 6 declares the annotation that has three methods, that is, `title`, `author1`, and `author2`. The default values of the last two methods are given as empty

strings (“”). The class `AnnoData` defines the annotation with book and its authors. The class `AnnoData` defines the annotation with actual data. The class also has a method. Another class with name `AUserAnnotation` is declared with `main` method. This class defines three objects and references, one each of classes `AnnoData`, `Method`, and annotation `MetaData` in lines 18, 19, and 20, respectively. The reference `md` of `MetaData` is used to inquire into the annotation in code lines 22–24. The output of this enquiry is shown.

Program 9.23 provides an example where multi-method annotation is being used.

**Program 9.23:** Illustration of a multi-method annotation

```

1  import java.lang.reflect.Method;
2  import java.lang.annotation.*;
3
4      @Retention(RetentionPolicy.RUNTIME)
5      @Target(ElementType.METHOD)
6          // annotation is declared below.
7      @interface MultiAnnotation {
8          int value1() default 0;
9          double value2() default 0.0;
10         char value3() default 'A';
11         String value4() default " ";
12     }
13
14     class Data{
15 @MultiAnnotation(value1 = 5, value2 = 10.45 , value3 = 'D', value4= "Delhi")
16 // note there is no semi-column.
17     public void display(){System.out.println("It is a user annotation."); }
18     }
19
20     public class MultiMethods {
21     public static void main(String args[])throws Exception{
22         Data data = new Data();
23         Method m = data.getClass().getMethod("display");
24         System.out.println(m.getDeclaringClass());
25         // below we use for-each loop
26         for(Annotation an : m.getDeclaredAnnotations())
27             System.out.println(an);
28         System.out.println(m.getDeclaredAnnotations());
29         System.out.println(m.getDeclaredAnnotations().length);
30         MultiAnnotation mA = m.getAnnotation (MultiAnnotation.class);
31         // reference of MultiAnnotation
32
33         System.out.println("The value1 = "+ mA.value1());
34         System.out.println("The value2 = " + mA.value2());
35         System.out.println("The value3 = " + mA.value3());
36         System.out.println("The value4 = " + mA.value4());
37         data.display();
38     }
39     }

```

**Output**

```

class Data
@MultiAnnotation(value2=10.45, value1=5,
value4=Delhi, value3=D)
[Ljava.lang.annotation.Annotation;@1540e19d
1
The value1 = 5
The value2 = 10.45
The value3 = D
The value4 = Delhi
It is a user annotation.

```

**Explanation**

The program illustrates four-member annotation and the code for accessing the data of the annotation. Besides, it also determines other information about the annotation. Code lines 7–12 declare a four-method annotation. The return types of methods are `int`, `double`, `char`, and `String`. Each default value is also declared with each method. The values of methods are set in code line 15 in class `Data`. Line 16 declares a linking method. Note that values are assigned with names of the methods.

Code line 19 declares a class with main method. The objects of `class Data` and `class Method` are declared in code lines 21 and 22. The reference to the annotation is defined in line 29.

Code line 23 gets information about the declaring class. The first line of output is the result—`class Data`. Code lines 25 and 26 make use of *for-each* loop to find the annotations. The second line of output is the result. It gives the name of annotation and its values.

### 9.10.7 Restrictions on Annotations

1. The annotations can have only methods as their members.
2. The methods have no parameters and no body. They have type, name, and a pair of parentheses followed by semicolon. (e.g., `String value ();`)
3. The methods can return values that are of primitive type, `enum` type, `Class`, `String`, another annotation, or an array of these.
4. The methods cannot have any throw classes.
5. An annotation cannot extend another annotation.
6. Default value to the methods may be declared.

## 9.11 Application Programs

As we know in an interface, normally abstract methods are declared. Several different concrete methods may be derived from the same abstract method. This is also a kind of polymorphism. Figure 9.3 and Program 9.24 illustrate the same. A `cutAndStitch()` method is declared in an interface `Dress`. The method may be used for stitching a shirt, a frock, or a suit. This is illustrated in Program 9.24. Lambda expressions are used for executing concrete method declaration in the expression.



**Fig. 9.3** Abstract method of interface leading to several concrete methods (see Program 9.24, which is written based on these figures)

### Program 9.24: Illustration of use of data and method declared in an interface

```

1  interface Dress {
2  int frockTime =6; //Labour time (hours) for stitching frock
3  int suitTime =100; //time (hours) needed for suit

```

```

4   int shirtTime =8 ;    //time(hours) needed for shirt
5
6   public double cutNStitch(double clothLength, double CC, double LC, int time);
7   //cc= cost of cloth, LC = labour cos/hour,
8   // Time = time of stitching
9   }                      // end of interface
10  // class that implements interface
11  public class Tailor {
12  public static void main(String[] args) {
13      // Declares costs of respective dresses
14      double frockCost, shirtCost, SuitCost;
15      // Method definitions in Lambda expression
16  Dress frock = (clothLength,CC, LC, time) -> {return(clothLength *CC +LC*
(1 + .5)*Dress.frockTime)};};
17  Dress shirt = (clothLength,CC, LC, time) -> {return(clothLength *CC +LC*
(1 + .5)*Dress.shirtTime)};};
18  Dress suit = (clothLength,CC, LC, time) -> {return(clothLength *CC +LC*
(1 + .5)*Dress.suitTime)};};
19      // Implementation of methods
20
21  double costFrock = frock.cutNStitch(1.5, 120.0, 25.0, 1);
22  double costShirt = shirt.cutNStitch(2.2, 150.0, 30.0, 8);
23  double costSuit = suit.cutNStitch(3.0, 750.0, 40.0, 100);
24
25  // output of costs for bill
26  System.out.println("Cost of stitching frock = " + costFrock);
27  System.out.println("Cost of Stitching shirt = " + costShirt);
28  System.out.println("Cost of Stitching suit = " + costSuit);
29      // Making total bill
30  System.out.println("Total Bill = " + (costFrock + costShirt + costSuit));
31  }
}

```

**Output**

```

Cost of stitching frock = 405.0
Cost of Stitching shirt = 690.0
Cost of Stitching suit = 8250.0
Total Bill = 9345.0

```

**Explanation**

The program computes the cost of stitching a dress. The different data given in the exercise is arbitrary. The interface declares time durations in hours required for stitching a frock, a suit, and a shirt. It also declares an abstract method for computing the cost of dresses. The variables of

method are `double clothLength`—length of cloth needed for the job, `double CC`—cost of cloth per metre for dress, `double LC`—cost of labour per hour, and `time` in hours needed by labour to complete the job. Fifty percent of the labour cost is taken as overhead cost for all cases and is included in labour cost.

Code line 14 declares variables for cost of three dresses. Lines 16–18 define the cost method for each dress. Although the method is same, separate definitions are given so that different formulations may be possible in the future. Lines 21–23 supply the data and calculate costs for the three dresses. The costs are displayed by lines 26–28. Line 30 prepares and displays the total bill for the three dresses. The outputs are given.

Program 9.25 illustrates an application program in which the area and volume of a given cylindrical shape is calculated.

**Program 9.25:** Illustration of another application program

```

1   interface Shape{
2   public double area(double r, double h);

```

```

3  public double volume(double r, double h);
4  interface Nested{
5  public void display();
6  }
7  }
8
9  public class ShapeCylinder implements Shape
10 {
11 double r, h;
12 public ShapeCylinder(){
13 double radius = r;
14 double height = h;
15 }
16 public double area()
17 {
18     double radius = r;
19 double height = h;
20
21 return 2*3.14*radius*(radius + height);
22 }
23 public double volume()
24 {
25 double radius = r;
26 double height = h;
27 return (3.14*radius*radius * height);
28 }
29 public void display(){
30 System.out.println("Volume and Area of cylinder is computed here");
31 }
32 }
33 }
34 public class ShapeImplement {
35 public static void main(String args[])
36 {
37     Shape.Nested a;
38     ShapeCylinder sc =new ShapeCylinder();
39     a = sc;
40     a.display();
41     Shape b;
42     ShapeCylinder st =new ShapeCylinder();
43     b = st;
44     System.out.println("Area of Cylinder is = " +b.area(7,15));
45
46     System.out.println("Volume of Cylinder is = " +b.volume(7,15));
47
48 }
49 }

```

**Output**

```

Volume and Area of cylinder is computed here
Area of Cylinder is = 967.12
Volume of Cylinder is = 2307.9

```

**Explanation**

The program declares two interfaces, one with name Shape and the other with name Nested. Two methods are declared in Shape and one in Nested. A class with name ShapeCylinder implements the interface

Shape. This class does not have main method. Therefore, another class ShapeImplement with main method executes the class ShapeCylinder. The important point is how to access nested interface. See line 37 in which reference to Nested is declared. The a calls method display. A reference to Shape is created in line 41 which is assigned object reference of class ShapeCylinder. This is used to call methods of Shape. The output is given.



## Common Programming Errors and Tips

Some common programming errors are described here.

1. The field variables declared in the interface must be initialized; otherwise, it will not compile.
2. A class implementing an interface must implement all its abstract methods.
3. The type signature of the implementing method should be exactly same as the type signature specified in the interface definition; otherwise, it may result in error.
4. Interfaces cannot implement themselves and they have no constructor methods, and hence, they cannot have objects. Use of operator new for their reference is wrong.

```
interface Book{}
Book book = new Book(); // wrong
```

The reference to Book may be created as in the following code, which is similar to declaration of primitive types of declaration.

```
Book book;
```

5. An interface with single abstract method can be the target of Lambda expression, which is explained in Chapter 6.
6. The field variables defined in top-level interfaces are implicitly public, static, and final. However, the specification of these modifiers with variables does not create a compile-type error.
7. All the annotations declare only methods; however, they have no bodies and parameters. Their values are like field variables.
8. Only the name of member is used while assigning a value to an annotation member. The parentheses are not used. Thus, if `String name();` is one member, the value is assigned as `name = "Geeta"` in the case of multi-method annotation. For single method, an annotation uses value as name.
9. Do not create instances of interface directly. For this, create an instance of some class that implements the interface and reference that instance as an instance of interface.
10. All the variables in interface are public, and therefore, public keyword in the variable declaration can be avoided.
11. All default, abstract, and static methods in interface are implicitly public, and therefore, public keyword in the method declaration can be omitted.
12. Nested interfaces cannot be accessed directly; it must be referred by the outer interface or class.
13. Nested interface can have any access modifier if it is declared within the class. However, it must be public if it is declared inside the interface.

## SUMMARY

- Declaration of an interface starts with the access modifier followed by the keyword interface. This is followed by its name or identifier, and then, by the block of statements containing declarations of variables and abstract methods.
- The variables defined in interfaces can be implicitly public, static, and final. They are initialized at the time of declaration.
- The modifiers that can be used with declaration of interfaces include public, private, and protected. However, for top-level interfaces, it is only public.
- The interfaces are implemented by classes. An interface can be declared as a member of a class or in another interface. In the capacity of a class member, it can have the attributes, which are applicable to other class members. Its access may be modified to public, protected, or private. In other cases, an interface can only be declared as public or with default (no-access modifier) access.
- An interface can also extend an interface.
- The Java version 8 allows full definition of default and static methods in interfaces. A static method is a class method. For calling a static method, one does not need an object of class. It can simply be called with class name.
- The functional interfaces are added in Java SE8. These are interfaces with one abstract method only and may be used as target for Lambda expressions and method references.

- A functional interface can have more than one static and default methods besides the abstract methods and it can override some methods of object class.
- In the program, an annotation, which is given, can be included in order to lessen the work of compiler that will recognize a functional interface.

@FunctionalInterface

Adding the aforementioned annotation would be helpful in detecting compile time errors.

- A new package `java.util.function` has been introduced in Java 8 that contains the predefined functional interfaces and that may be used as target for Lambda expressions for most of the applications.

## GLOSSARY

**@Functional Interface** A connotation for compiler to indicate that it is functional interface.

**Abstract method** It is a method that is only declared in a class but not defined. An abstract class is any class that includes an abstract method.

**Default methods in interfaces** Java 8 has allowed full definition of default methods in interfaces. The method definition has to be qualified by keyword default.

**Functional interfaces** An interface that has one abstract method; also called SAM (single abstract method type).

**Generic interface** Like a class, an interface is generic if it declares one or more type of variables.

**Java.util.function** A new package is created in Java 8 that contains predefined functional interfaces having one abstract method to be used as target for Lambda expressions.

**Nested interface** It is an interface that is defined in the body of a class or interface.

**Static method** A static method is a class method and we do not need an object of class for calling a static method.

**Static methods in interfaces** Java 8 has allowed full definition of static methods in interfaces. The method definition has to be qualified by keyword static.

**Top-level interfaces** It is an interface that is not nested in any class or interface.

## EXERCISES

### Multiple-choice Questions

- Which of the following statements are correct?
  - An interface can extend only one class.
  - An interface can extend any number of classes.
  - An interface can extend only one interface.
  - An interface can extend any number of interfaces.
- Which of the following access specifiers can be used for an interface?
  - Public
  - Private
  - Protected
  - All of these
- Which of the following statements are correct?
  - Class can extend an interface.
  - Interface can extend a class.
  - Class implements interfaces.
  - None of these.
- Which of the following statements are correct?
  - A concrete class implementing an interface must implement all the abstract methods of the interface.
  - A class may partially implement an interface.
  - A class that implements an interface must implement at least one abstract method of interface.
  - A class implementing fully a number of interfaces must be declared an abstract class.
- Which of the following is the correct way of implementing an interface `Acceleration` by class `Vehicle`?
  - class `Vehicle` implements `Acceleration` {}
  - class `Vehicle` extends `Acceleration` {}
  - class `Vehicle` imports `Acceleration` {}
  - None of these
- Which of the following statements are correct for classes implementing one interface?
  - All the classes that implement an interface must have the same definitions of methods declared in an interface.
  - Different classes can have different definitions of the methods of the interface.
  - Classes can assign different values to variables defined in an interface.
  - Variables defined in an interface are static and final.
- Which of the following represents the correct definition of interface?
  - interface `Shape` {void draw() {}}
  - interface `Shape` {void draw()}
  - interface `shape` {void draw};
  - interface `Shape` {void draw();}
- Which of the following statements are correct regarding interfaces?
  - class `Vehicle` implements `Acceleration` {}
  - class `Vehicle` extends `Acceleration` {}
  - class `Vehicle` imports `Acceleration` {}
  - None of these

- (a) An interface cannot implement itself.  
 (b) An interface can have object references.  
 (c) A class can extend an interface.  
 (d) An interface can implement another interface.
9. How many abstract methods should a functional interface have?  
 (a) One (c) Three  
 (b) Two (d) Any number
10. In Java, default methods can be declared in which of the following cases?  
 (a) In classes only  
 (b) In interfaces only  
 (c) In classes as well as in interfaces  
 (d) Outside classes or interfaces
11. Which of the following is correct for calling a static method?  
 (a) `Object_name.class_name.method_name`  
 (b) `Class_name.object_name`  
 (c) `class_name.method_name`  
 (d) None of these
12. Which of the following statements is correct?  
 (a) A functional interface may have default and static methods besides abstract method and/or other variables.  
 (b) A functional interface can have only one default method besides one abstract method.  
 (c) A functional interface can have only one static method besides one abstract method.  
 (d) A functional interface cannot have default or static methods.
13. Which of the following lines would give compilation error?  

```
interface Score {
    int p = 15;
// line 1
    public static int x = 23;
// line 2
```
- ```
public int y = 34;
// line 3
public static final int z = 56;
// line 4
}
```
- (a) 1 (c) 3  
 (b) 2 (d) 4  
 (e) None of these
14. Which of the following statements are correct regarding default methods in interfaces?  
 (a) A default method may be declared final.  
 (b) A default method can be synchronized.  
 (c) A default method cannot override the non-final methods of Object class.  
 (d) None of these.
15. What would be the output of the following codes?  

```
interface Show{
    public void method();
}
class X {
    public void method(){
        System.out.println (" Class X method");
    }
}
class Y extends X implements Show{
    public void method() {
        System.out.println (" Class Y method");
    }
}
public class Demo extends Y {
    public static void main (String[] args)
    {
        Show s = new Y();
        s.method();
    }
}
```

 (a) Class Y method  
 (b) Class X method  
 (c) Compiles but does not print anything  
 (d) Gives compilation error

## Review Exercises

- Differentiate between a class and an interface.
- What is the purpose of having an interface in a program?
- What are functional interfaces?
- What is the difference between an interface and an abstract class?
- What are the attributes of a variable declared in an interface?
- How do you declare an interface?
- How do you implement an interface?
- How are methods declared in an interface?
- Does an interface extend another interface?
- Does an interface implement another interface?
- Can you declare an interface inside a class?
- Can you declare a class in an interface?
- How do you access an interface declared in a class?
- Can an interface extend more than one interface?
- What is a predicate?

## Programming Exercises

1. Declare an interface with a method to calculate the volume with one double parameter. Implement the same for finding the volume of a sphere and cube by two different classes.
2. Write a program that illustrates accessing a nested interface in another interface that is nested in a class.
3. Write a program to illustrate the implementation of nested interfaces.
4. Write a program to illustrate use of interface with a nested class.
5. Write a program in which a class implements more than two interfaces.
6. Write a program to illustrate an interface extending two interfaces and implemented by a class.
7. Write a program that uses a predicate to test if an array of integers contains a particular number.
8. Design an interface named `InfaceX` with a method `reverse()` that reverses the digits of the number (i.e., if 45,678 is the number, then it should return 87,654).
9. Design an interface with the following details:

Interface named `InfaceA` having method `getVelocity(r)` that calculates and returns velocity of satellite as given

$$\text{Velocity} = \text{sqrt. of } (\mu/r)$$

where  $r$  is the altitude of satellite measured from the centre of the earth,  $\mu$  is the Kepler's constant with value  $3.986004418 \times 10^5$ .

There is nested interface, `InfaceB` with method, and `getAcceleration(r)` that calculates and returns the acceleration of the satellite as

$$\text{Acceleration} = (\mu/r^2)$$

10. Write a program in which interface is given by name `MeanInterface`. Method `mean()` is defined in this interface that calculates the mean of the given numbers arranged in an array. This interface is then extended and method is defined in this interface that calculates deviation from the mean value evaluated for each of the numbers.
11. Write a program that illustrates the functional Binary Operator.
12. Write a program that illustrates the functional Function.

## Debugging Exercises

1. Debug the following program and run it to find the area of a circle and a square.

```
interface Volume;
{double Compute (double x);
}

class Cube implements Volume
{public double Compute (double x)
{return (x*x*x);}
}
class Sphere implements Volume
{public double Compute(double x)
{return ( 4*3.141*x*x*x/3);}
}
classCubSpher;
{public static void main(String arg[])
{ Cube Cub = new Cube()
Sphere Sphr = new Sphere();
Volume Vol;
Vol = Cub;
System.out.println("Volume of cube = " +
Vol.Compute(10));
Vol =Sphr;
System.out.println("Volume of sphere = " +
Vol.Compute());
}}
```

2. Debug the following program and run it to find the area of a circle and a square.

```
public interface A{
int area(int x, int y);
}
{
int i = x;
int j = y;
return (i*j);
}
}
class SingleRoom implements A
{
public int area(int x, int y);
}
class Room
{
public static void main(String args[])
{
Room r=new Room();
System.out.println("Area of room is = " +
r.area(4.0, 7.0));
}
}
```

3. Debug the following program and run it to find the area of a circle and a square.

```
class demo {
public interface Inface {
```

```
int sum( int x, int y);
}

class B implements Inface
{
public int sum (int x, int y )
    {return x + y ; }
}

class ABC
{public static void main(String arg[])
{
inface s = new B();
System.out.println("Addition of two numbers
is = " + s.sum (30, 12));
}
}
```

4. Debug the following program and run it to find the area of a circle and a square.

```
interface InfaceD
{
double getVolume(double x);
interface Display{
public void show();
}
}

1. class Calculate implements InfaceD, Display
{
double getVolume(double x)
{
double volume = 4/3*PI*x*x*x;
return volume;
}
public void Display.show()
{
System.out.println("Volume of given
shapes");
}

public static void main(String args[])
{
2. Display d=new Calculate();
InfaceD s = new Calculate();
d.show();
System.out.println("Volume is = " +d.get-
Volume(4.0)");
}
}
```

5. Debug the following program and run it to find the area of a circle and a square.

```
import java.lang.reflect.Method;
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
// annotation is declared below.
@interface MultiAnnotation {
int value1() default 0;
double value2() default 0.0;
char value3() default 'A';
String value4() default " ";
}

class Data{
@MultiAnnotation(value1 = 5, value2 = 10.45
, value3 = 'D', value4= "Delhi")
public void display(){System.out.println("It
is a user annotation."); }
}

public class MultiMethods { // class
with main method
public static void main(String args[])
throws Exception{

Data data = new Data();
Method m = data.getClass().get-
Method("display");
System.out.println(m.getDeclared
Class());
System.out.println(m.getDeclaringAnnota-
tions());

System.out.println(m.getDeclaredAnnota-
tions().length);
MultiAnnotation mA=m.getAnnotation(MultiAn-
notation.class);

System.out.println("The value1 = " +
mA.value1());
System.out.println("The value2 = " +
mA.value2());

System.out.println("The value3 = " +
mA.value3());
System.out.println("The value4 = " + mA.val-
ue4());
data.display();
}
}
```

**Answers to Multiple-choice Questions**

---

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

---