

# **Programming with JAVA**

**A PRIMER**

**E BALAGURUSAMY**

Director  
PSG Institute of Management  
Coimbatore



**Tata McGraw-Hill Publishing Company Limited**

**NEW DELHI**

*McGraw-Hill Offices*

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas Lisbon London  
Madrid Mexico City Milan Montreal San Juan Singapore Sydney Tokyo Toronto

**Tata McGraw-Hill**

*A Division of The McGraw-Hill Companies*



Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

© 1998, Tata McGraw-Hill Publishing Company Limited

Second reprint 1998

RYDXCDDKRXLRY

No part of this publication can be reproduced in any form or by any means without the prior written permission of the publishers

This edition can be exported from India only by the publishers,  
Tata McGraw-Hill Publishing Company Limited

ISBN 0-07-463049-0

Published by Tata McGraw-Hill Publishing Company Limited,  
7 West Patel Nagar, New Delhi 110 008, typeset at  
LeoCap Expressions, B 302, Rishi Apartments, Alaknanda, New Delhi-110 049  
and printed at Replika Press Pvt. Ltd., Plot No. A-229,  
DSIDC Narela Industrial Park, Narela, New Delhi-110 040

*Dedicated to*

Shri N Chandrababu Naidu  
Honourable Chief Minister of Andhra Pradesh

*The Author of*  
Cyberspace Government in Andhra Pradesh

# Preface

Java is yet another computer language but with a difference. It is the only language that is purely object-oriented. Java's designers have borrowed the best features of many existing languages such as C and C++ and added a few new features to form a simple, easy-to-learn and object-oriented language. It is the most complete programming language available today. It is also a secure language, making it well-suited for Internet programming. One of the important reasons for Java's success, apart from its object-orientation, is the amazing functionality it adds to the World Wide Web.

Java has two lives--one as a stand-alone computer language for general-purpose programming and the other as a supporting language for Internet programming. The general-purpose programs are known as *applications* and programs written for Internet are known as *applets*. Till recently, C++ has been considered as an industry standard language for object-oriented programming. Now the battle between Java and C++ has begun. We must get ready for an industry starving for Java programmers.

This book is for novice as well as experienced programmers. While the book assumes that the reader's ultimate goal is to develop Java programs, both *applications* and *applets*, it does not assume any significant knowledge of programming on the part of the reader. If the reader is a C or C++ programmer, he or she may probably be able to read through some of the initial chapters quickly. However, a novice reader will need to go through the whole book carefully.

This book comprehensively covers all aspects of Java language. Beginning with an introduction to the language and its relationship with the Internet and World Wide Web, it explores Java's object-oriented features, and then moves on to discuss advanced topics that are unique to Java.

The concept of 'learning by example' has been stressed throughout the book. Each major feature of the language is treated in depth followed by a complete program example to illustrate its use. Wherever necessary, concepts are explained pictorially to facilitate better understanding.

The book contains a large number of example programs. All programs have been tested and, wherever possible, the nature of output has been discussed. These programs also demonstrate

the general principles of a good programming style. This book has all that a reader needs to start programming in Java right away.

Finally, this book is for everyone who is either excited about Internet or interested in Java Programming.

E BALAGURUSAMY

## Acknowledgements

No book is created entirely by an individual. Many people have helped to create this book and each of their contribution has been valuable. The timely completion of this book is mainly due to the interest and persuasion of late Prof. N K Venkatasubramanian who was not only my teacher and colleague but also a good friend and guide. His contribution will be remembered forever.

I would like to thank many other individuals at PSG Institute of Management who have contributed greatly to the success of this project. Thanks are due to G P Raja, S Lalitha, K Bl;Iakrishnan, S Saravanan, J R Pratibha, and G Nithya for their valuable assistance in preparing the manuscript.

The idea of this book was planted by my wife, Sushila, while reading an article on Java in *The Hindu* newspaper. My special thanks are due to her not only for the idea but also for the encouragement and unstinted support throughout the Writing of this book.

Thanks are due to R Radhakrishnan and Saroja Radhakrishnan for their active involvement throughout the production stages of the book.

Finally, I wish to thank Dr N Subrahmanyam, Vibha Mahajan and other publishing professionals at Tata McGraw-Hill for bringing out the book in its present form in record time.

**t** BALAGURUSAMY

# Contents

Preface	vii
Acknowledgements	ix
<b>FUNDAMENTALS OF OBJECT-ORIENTED PROGRAMMING</b>	
	1
1.1 Introduction	1
1.2 Object-Oriented Paradigm	2
1.3 Basic Concepts of Object-Oriented Programming	3
<i>Objects and classes</i>	3
<i>Data abstraction and encapsulation</i>	
<i>Inheritance</i>	5
<i>Polymorphism</i>	6
<i>Dynamic binding</i>	6
<i>Message communication</i>	
1.4 Benefits of OOP	8
1.5 Applications of OOP	9
1.6 Summary	10
Review Questions	10
 <b>JAVA EVOLUTION</b>	12
	12
2.1 Java History	12
2.2 Java Features	13
<i>Compiled and interpreted</i>	13
<i>Platform-independent and portable</i>	14

<i>Object-oriented</i>	14
<i>Robust and secure</i>	14
<i>Distributed</i>	14
<i>Simple, small and familiar</i>	15
<i>Multithreaded and interactive</i>	15
<i>High performance</i>	15
<i>Dynamic and extensible</i>	15
<b>2.3 How Java Differs from C and C++</b>	<b>15</b>
<i>Java and C</i>	16
<i>Java and C++,</i>	16
<b>2.4 Java and Internet</b>	<b>17</b>
<b>2.5 Java and World Wide Web</b>	<b>17</b>
<b>2.6 Web Browsers</b>	<b>19</b>
<i>HotJava</i>	20
<i>Netscape Navigator</i>	21
<i>Internet Explorer</i>	21
<b>2.7 Hardware and Software Requirements</b>	<b>21</b>
<b>2.8 Java Support Systems</b>	<b>21</b>
<b>2.9 Java Environment</b>	<b>22</b>
<i>Java development kit</i>	22
<i>Java standard library</i>	23
<b>2.10 Summary</b>	<b>24</b>
<b>Review Questions</b>	<b>24</b>



## OVERVIEW OF JAVA LANGUAGE 26

<b>3.1 Introduction</b>	<b>26</b>
<b>3.2 Simple Java Program</b>	<b>27</b>
<i>Class declaration</i>	28
<i>Opening brace</i>	28
<i>The main line</i>	28
<i>The output line</i>	29
<b>3.3 More of Java</b>	<b>29</b>
<i>Use of math functions</i>	30
<i>Comments</i>	30
<b>3.4 An Application with Two Classes</b>	<b>30</b>
<b>3.5 Java Program Structure</b>	<b>31</b>
<i>Documentation section</i>	32
<i>Package statement</i>	32
<i>Import statements</i>	33

<i>Interface statements</i>	33
<i>Class definitions</i>	33
<i>Main method class</i>	33
<b>3.6 Java Tokens</b>	<b>33</b>
<i>Java character set</i>	35
<i>Keywords</i>	35
<i>Identifiers</i>	36
<i>Literals</i>	37
<i>Operators</i>	37
<i>Separators</i>	37
<b>3.7 Java Statements</b>	<b>38</b>
<b>3.8 Implementing a Java Program</b>	<b>40</b>
<i>Creating the program</i>	40
<i>Compiling the program</i>	40
<i>Running the program</i>	41
<i>Machine neutral</i>	42
<b>3.9 Java Virtual Machine</b>	<b>42</b>
<b>3.10 Command Line Arguments</b>	<b>43</b>
<b>3.11 Programming Style</b>	<b>45</b>
<b>3.12 Summary</b>	<b>46</b>
<i>Review Questions</i>	46



<b>CONSTANTS, VARIABLES, AND DATA TYPES</b>	<b>47</b>
<b>4.1 Introduction</b>	<b>47</b>
<b>4.2 Constants</b>	<b>47</b>
<i>Integer constants</i>	47
<i>Real constants</i>	48
<i>Single character constants</i>	49
<i>String constants</i>	49
<i>Backslash character constants</i>	49
<b>4.3 Variables</b>	<b>50</b>
<b>4.4 Data Types</b>	<b>50</b>
<i>Integer types</i>	51
<i>Floating point types</i>	52
<i>Character type</i>	53
<i>Boolean type</i>	53
<b>4.5 Declaration of Variables</b>	<b>53</b>
<b>4.6 Giving Values to Variables</b>	<b>54</b>
<i>Assignment statement</i>	54

<i>Read statement</i>	55
4.7 Scope of Variables	56
4.8 Symbolic Constants	57
<i>Modifiability</i>	58
<i>Understandability</i>	58
4.9 Type Casting	58
<i>Automatic conversion</i>	59
4.10 Getting Values of Variables	61
4.11 Standard Default Values	63
4.12 Summary	63
Review Questions	64

**5****OPERATORS AND EXPRESSIONS**

66

5.1 Introduction	66
5.2 Arithmetic Operators	66
<i>Integer arithmetic</i>	67
<i>Real arithmetic</i>	67
<i>Mixed-mode arithmetic</i>	68
5.3 Relational Operators	69
5.4 Logical Operators	71
5.5 Assignment Operators	72
5.6 Increment and Decrement Operators	73
5.7 Conditional Operator	74
5.8 Bitwise Operators	74
5.9 Special Operators	75
<i>instanceof operator</i>	75
<i>Dot operator</i>	75
5.10 Arithmetic Expressions	75
5.11 Evaluation of Expressions	76
5.12 Precedence of Arithmetic Operators	76
5.13 Type Conversions in Expressions	78
<i>Automatic type conversion</i>	78
<i>Casting a value</i>	79
5.14 Operator Precedence and Associativity	81
5.15 Mathematical Functions	82
5.16 Summary	83
Review Questions	85



## DECISION MAKING AND BRANCHING

88

6.1	Introduction	88
6.2	Decision Making with if Statement	88
6.3	Simple if Statement	90
6.4	The If...else Statement	92
6.5	Nesting of if...else Statements	94
6.6	The else If Ladder	98
6.7	The switch Statement	102
6.8	The ?: Operator	106
6.9	Summary	107
	Review Questions	108



## DECISION MAKING AND LOOPING

111

7.1	Introduction	111
7.2	The while Statement	113
7.3	The do Statement	114
7.4	The for Statement	116
	Additional features of for loop	119
	Nesting of for loops	121
7.5	Jumps in Loops	122
	Jumping out of a loop	123
	Skipping a part of a loop	123
7.6	Labelled Loops	123
7.7	Summary	126
	Review Questions	127



## CLASSES, OBJECTS AND METHODS

129

8.1	Introduction	129
8.2	Defining a Class	129
8.3	Adding Variables	130
8.4	Adding Methods	130
8.5	Creating Objects	133
8.6	Accessing Class Members	134

<b>8.7 Constructors</b>	<b>137</b>
<b>8.8 Methods Overloading</b>	<b>138</b>
<b>8.9 Static Members</b>	<b>139</b>
<b>8.10 Nesting of Methods</b>	<b>141</b>
<b>8.11 Inheritance: Extending a Class</b>	<b>142</b>
<i>Defining a subclass</i> 142	
<i>Subclass constructor</i> 145	
<i>Multilevel inheritance</i> 145	
<i>Hierarchical inheritance</i> 146	
<b>8.12 Overriding Methods</b>	<b>147</b>
<b>8.13 Final Variables and Methods</b>	<b>148</b>
<b>8.14 Final Classes</b>	<b>149</b>
<b>8.15 Finalizer Methods</b>	<b>149</b>
<b>8.16 Abstract Methods and Classes</b>	<b>149</b>
<b>8.17 Visibility Control</b>	<b>150</b>
<i>public access</i> 150	
<i>friendly access</i> 150	
<i>protected access</i> 151	
<i>private access</i> 151	
<i>private protected access</i> 151	
<i>Rules of Thumb</i> 152	
<b>8.18 Summary</b>	<b>152</b>
<b>Review Questions</b>	<b>152</b>

**ARRAYS, STRINGS AND VECTORS****155**

<b>9.1 Arrays</b>	<b>155</b>
<b>9.2 One-Dimensional Arrays</b>	<b>155</b>
<b>9.3 Creating an Array</b>	<b>157</b>
<i>Declaration of arrays</i> 157	
<i>Creation of arrays</i> 157	
<i>Initialization of arrays</i> 158	
<i>Array length</i> 160	
<b>9.4 Two-Dimensional Arrays</b>	<b>161</b>
<i>Variable size arrays</i> 164	
<b>9.5 Strings</b>	<b>164</b>
<i>String arrays</i> 166	
<i>String methods</i> 166	
<i>StringBuffer class</i> 167	

9.6	Vectors	169
9.7	Wrapper Classes	171
9.8	Summary	175
	Review Questions	175
<b>INTERFACES: MULTIPLE INHERITANCE</b>		<b>179</b>
10.1	Introduction	179
10.2	Defining Interfaces	179
10.3	Extending Interfaces	181
10.4	Implementing Interfaces	182
10.5	Accessing Interface Variables	184
10.6	Summary	187
	Review Questions	187
<b>PACKAGES: PUTTING CLASSES TOGETHER</b>		<b>188</b>
11.1	Introduction	188
11.2	System Packages	189
11.3	Using System Packages	190
11.4	Naming Conventions	191
11.5	Creating Packages	192
11.6	Accessing a Package	193
11.7	Using a Package	193
11.8	Adding a Class to a Package	198
11.9	Hiding Classes	199
11.10	Summary	200
	Review Questions	200
<b>MULTITHREADED PROGRAMMING</b>		<b>201</b>
12.1	Introduction	201
12.2	Creating Threads	203
12.3	Extending the Thread Class	204
	<i>Declaring the class</i>	<i>204</i>
	<i>Implementing the run() method</i>	<i>205</i>

<i>Starting new thread</i>	205
<i>An example of using the thread class</i>	205
<b>12.4 Stopping and Blocking a Thread</b>	<b>208</b>
<i>Stopping a thread</i>	208
<i>Blocking a thread</i>	208
<b>12.5 Life Cycle of a Thread</b>	<b>208</b>
<i>Newborn state</i>	209
<i>Runnable state</i>	210
<i>Running state</i>	210
<i>Blocked state</i>	212
<i>Dead state</i>	212
<b>12.6 Using Thread Methods</b>	<b>212</b>
<b>12.7 Thread Exceptions</b>	<b>214</b>
<b>12.8 Thread Priority</b>	<b>215</b>
<b>12.9 Synchronization</b>	<b>218</b>
<b>12.10 Implementing the 'Runnable' Interface</b>	<b>220</b>
<b>12.11 Summary</b>	<b>221</b>
<b>Review Questions</b>	<b>222</b>
	<small>ANSWERS</small>

## **MANAGING ERRORS AND EXCEPTIONS**

223

<b>13.1 Introduction</b>	<b>223</b>
<b>13.2 Types of Errors</b>	<b>223</b>
<i>Compile-time errors</i>	223
<i>Run-time errors</i>	225
<b>13.3 Exceptions</b>	<b>226</b>
<b>13.4 Syntax of Exception Handling Code</b>	<b>227</b>
<b>13.5 Multiple Catch Statements</b>	<b>230</b>
<b>13.6 Using finally Statement</b>	<b>232</b>
<b>13.7 Throwing Our Own Exceptions</b>	<b>233</b>
<b>13.8 Using Exceptions for Debugging</b>	<b>235</b>
<b>13.9 Summary</b>	<b>235</b>
<b>Review Questions</b>	<b>236</b>

## **APPLET PROGRAMMING**

237

<b>14.1 Introduction</b>	<b>237</b>
<i>Local and remote applets</i>	237

<b>14.2 How Applets Differ from Applications</b>	<b>239</b>
<b>14.3 Preparing to Write Applets</b>	<b>239</b>
<b>14.4 Building Applet Code</b>	<b>240</b>
<b>14.5 Applet Life Cycle</b>	
<i>Initialization state</i> 244	
<i>Running state</i> 244	
<i>Idle or stopped state</i> 244	
<i>Dead state</i> 245	
<i>Display state</i> 245	
<b>14.6 Creating an Executable Applet</b>	
<b>14.7 Designing a Web Page</b>	
<i>Comment section</i> 246	
<i>Head section</i> 246	
<i>Body section</i> 247	
<b>14.8 Applet Tag</b>	<b>248</b>
<b>14.9 Adding Applet to HTML File</b>	<b>249</b>
<b>14.10 Running the Applet</b>	<b>250</b>
<b>14.11 More About Applet Tag</b>	<b>251</b>
<b>14.12 Passing Parameters to Applets</b>	<b>253</b>
<b>14.13 Aligning the Display</b>	<b>254</b>
<b>14.14 More About HTML Tags</b>	<b>257</b>
<b>14.15 Displaying Numerical Values</b>	<b>258</b>
<b>14.16 Getting Input from the User</b>	<b>259</b>
<i>Program analysis</i> 261	
<b>14.17 Summary</b>	<b>262</b>
<b>Review Questions</b>	<b>262</b>
<b>GRAPHICS PROGRAMMING</b>	<b>263</b>
<b>15.1 Introduction</b>	<b>263</b>
<b>15.2 The Graphics Class</b>	<b>263</b>
<b>15.3 Lines and Rectangles</b>	<b>265</b>
<b>15.4 Circles and Ellipses</b>	<b>267</b>
<b>15.5 Drawing Arcs</b>	<b>269</b>
<b>15.6 Drawing Polygons</b>	<b>271</b>
<b>15.7 Line Graphs</b>	<b>274</b>
<b>15.8 Using Control Loops in Applets</b>	<b>275</b>
<b>15.9 Drawing Bar Charts</b>	<b>277</b>

<b>15.10</b>	<b>Summary</b>	<b>279</b>
	<b>Review Questions</b>	<b>279</b>

## **APPENDICES**

<b>Appendix A</b>	<b>Java Language Reference</b>	<b>281</b>
<b>Appendix B</b>	<b>Java Keywords</b>	<b>289</b>
<b>Appendix C</b>	<b>Differences Between Java and C/C++</b>	<b>293</b>
<b>Appendix D</b>	<b>Bit-Level Programming</b>	<b>297</b>
<b>Appendix E</b>	<b>Java Class Library</b>	<b>304</b>
<b>Appendix F</b>	<b>Java Classes and Their Packages</b>	<b>312</b>
<b>Appendix G</b>	<b>Points to Remember</b>	<b>322</b>
<b>Appendix H</b>	<b>Common Coding Errors</b>	<b>325</b>
<b>Appendix I</b>	<b>Glossary of Java Terms</b>	<b>327</b>
<b>Bibliography</b>		<b>336</b>
<b>Index</b>		<b>338</b>

# Chapter 1

## INTRODUCTION

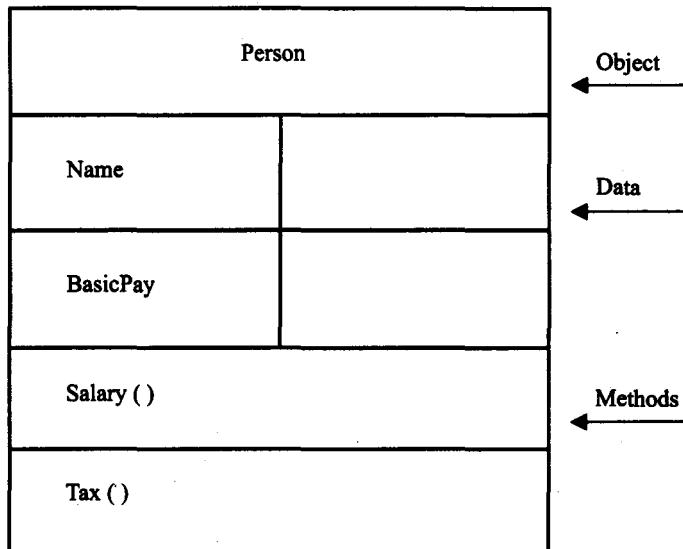
One characteristic that is constant in the software industry today is the "change". Change is one of the most critical aspects of software development and management. New tools and new approaches are announced almost every day. The impact of these developments is often very extensive and raises a number of issues that must be addressed by the software engineers. Most important among them are maintainability, reusability, portability, security, integrity, and user friendliness of software products.

To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules. We need to use sound construction techniques and program structures that are easy to comprehend, implement and modify in a wide variety of situations.

Since the invention of the computer, many programming approaches have been tried. These include techniques such as *modular programming*, *top-down programming*, *bottom-up programming* and *structured programming*. The primary motivation in each case has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques became popular among programmers over the last two decades.

With the advent of languages such as C, structured programming became very popular and was the paradigm of the 1980s. Structured programming proved to be a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development, which attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily. Languages that support OOP features include Smalltalk, Objective C, C++, Ada and

**Fig. 1.2****Representation of an object**

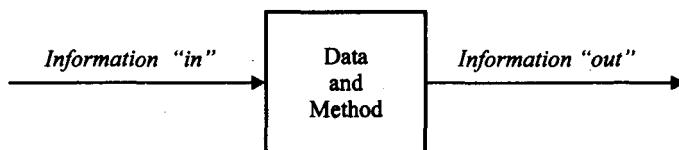
Classes are user-defined data types and behave like the built-in types of a programming language. For example, the syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
Fruit mango;
```

will create an object mango belonging to the class fruit.

### Data Abstraction and Encapsulation

The wrapping up of data and methods into a single unit (called class) is known as *encapsulation*. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those methods, which are wrapped in the class, can access it. These methods provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding*. Encapsulation makes it possible for objects to be treated like 'black boxes', each performing a specific task without any concern for internal implementation (see Fig. 1.3).

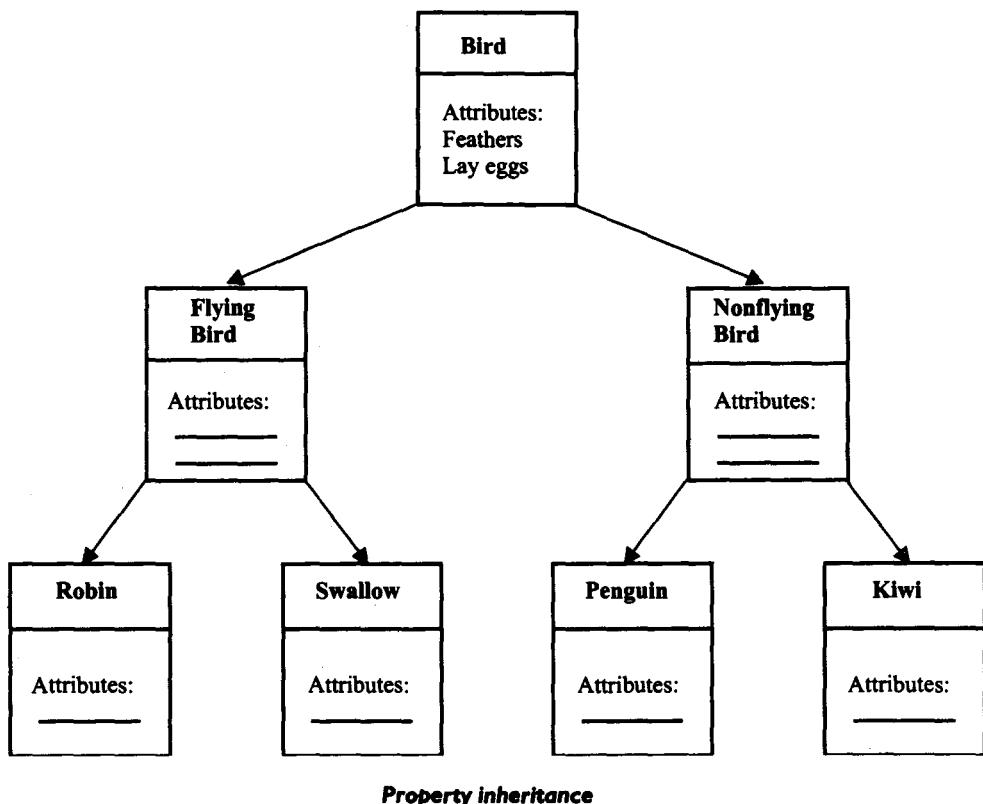
**Fig. 1.3****Encapsulation—Objects as "black boxes"**

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and methods that operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

## Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying bird, which is again a part of the class bird. As illustrated in Fig. 1.4, the principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

**Fig. 1.4**



In OOP, the concept of inheritance provides the idea of reusability: This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. Thus the real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class

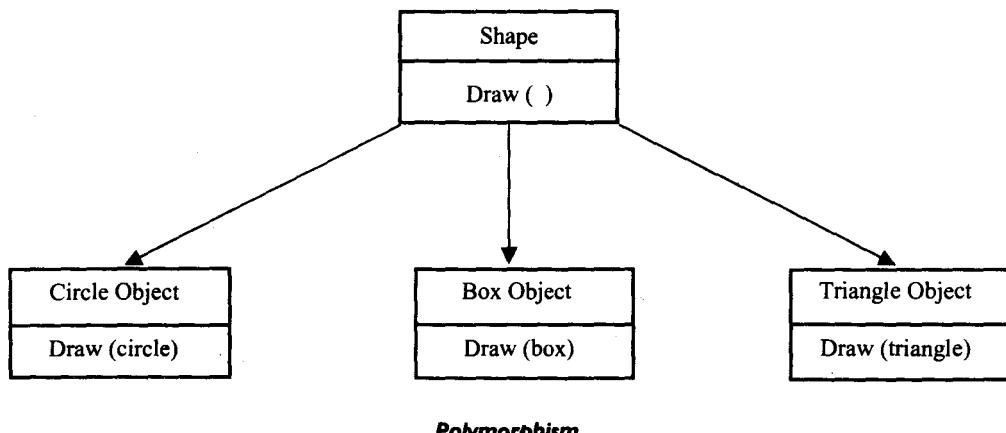
in such a way that it does not introduce any undesirable side effects into the rest of the classes. In Java, the derived class is known as 'subclass'.

Note that each subclass defines only those features that are unique to it. Without the use of inheritance, each class would have to explicitly include all of its features.

## Polymorphism

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. Figure 1.5 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context.

**Fig. 1.5**



Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

## Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at runtime. It is associated with polymorphism and inheritance. A procedure call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in Fig. 1.5. By inheritance, every object will ~ this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

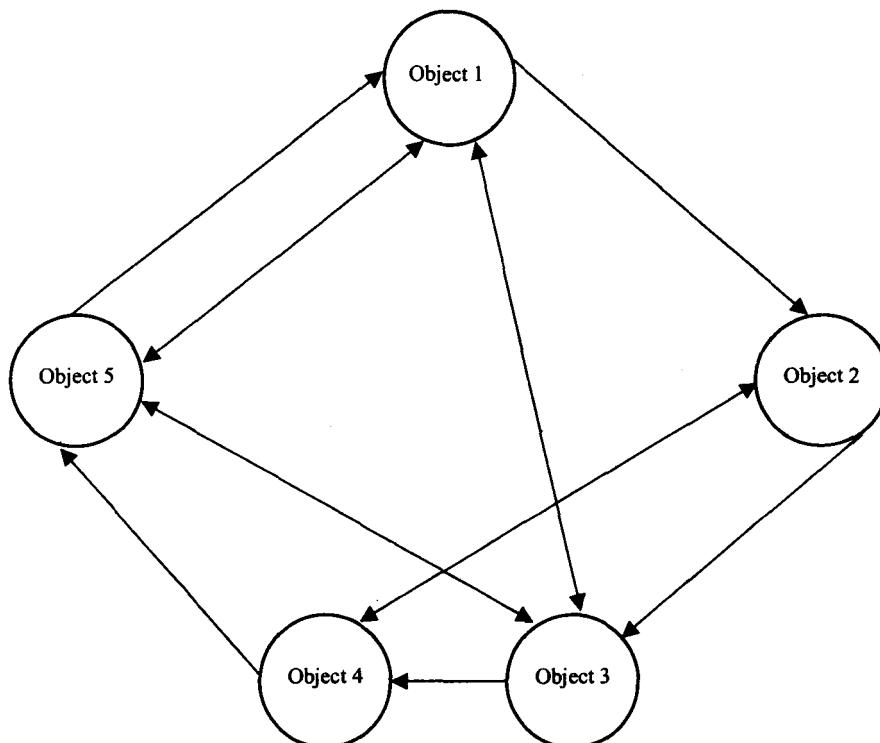
### Message Communication

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour.
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another as shown in Fig. 1.6. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

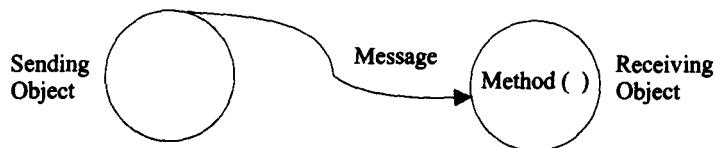
**Fig. 1.6**



**Network of objects communicating between them**

A message for an object is a request for execution of a procedure, and therefore will invoke a method (procedure) in the receiving object that generates the desired result, as shown in Fig. 1.7.

**Fig. 1.7**



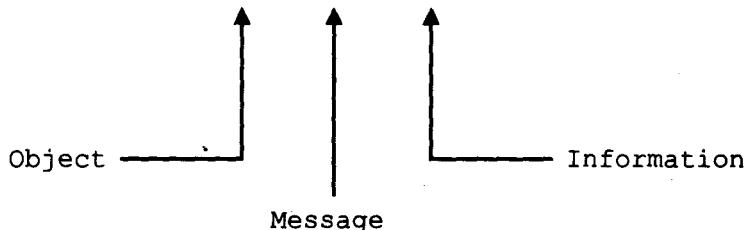
### **Message triggers a method**

Message passing involves specifying the name of the object, the name of the method (message) and the information to be sent. For example, consider the statement

```
Employee.salary(name);
```

Here, **Employee** is the object, **salary** is the message and **name** is the parameter that contains information.

```
Employee.salary(name);
```



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

## **1.4      BENEFITS OF OOP**

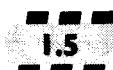
OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes .
  - We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple objects to coexist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in an implementable form.
- Object-oriented system can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects make the interface descriptions with external systems much simpler ..
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, class libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

A software that is easy to use is *hard to build*. It is hoped that the object-oriented programming languages like C++ and Java would help manage this problem.



## APPLICATIONS OF OOP

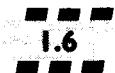
OOP is one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. There are hundreds of windowing systems developed using OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in this type of applications because it can simplify a complex problem. The promising areas for application of OOP includes:

- Real-time systems
- Simulation and modelling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAD/CAD system

It is believed that the richness of OOP environment will enable the software industry to improve not only the quality of software systems but also its productivity. Object-oriented

technology is certainly going to change the way ~~soU'''-e ---rs~~ will think, analyze, design and implement systems in the future.



## SUMMARY

Java is an object-oriented language. It enables us not only to ~~ORj1UIULE~~ our program code into logical units called objects but also to take advantage of ~~cl Eyon~~ inheritance, and polymorphism. In this chapter, we have introduced the basic ~~cocepts~~ of object-oriented programming which include

- Encapsulation,
- Inheritance, and
- Polymorphism

We also discussed briefly the benefits and applications of object-oriented programming approach.

### KEY TERMS

**Structured Programming, Object-Oriented Paradigm, Class, Object, Method, Abstraction, Encapsulation, Data Hiding, Inheritance, Reusability, Polymorphism, Dynamic Binding.**

### REVIEW QUESTIONS

- 1.1 What do you think are the major issues facing the software industry today?
- 1.2 Briefly discuss the software evolution during the period from 1950 to 1995.
- 1.3 What is object-oriented programming? How is it different from the procedure-oriented programming?
- 1.4 How are data and methods organized in an object-oriented program?
- 1.5 What are unique advantages of an object-oriented programming paradigm?
- 1.6 Distinguish between the following terms:
  - (a) Objects and classes
  - (b) Data abstraction and data encapsulation
  - (c) Inheritance and polymorphism
  - (d) Dynamic binding and message passing
- 1.7 What kinds of things can become objects in OOP?
- 1.8 Describe inheritance as applied to OOP.
- 1.9 List a few areas of application of OOP technology.

1.10 State whether the following statements are TRUE or FALSE.

- (a) In conventional, procedure-oriented programming, all data are shared by all functions.
- (b) The main emphasis of procedure-oriented programming is on algorithms **rather** than on data.
- (c) One of the striking features of object-oriented programming is the division of programs into objects that represent real-world entities.
- (d) Wrapping up of data of different types into a single unit is known as encapsulation.
- (e) One problem with OOP is that once a class is created, it can never be changed.
- (f) Inheritance means the ability to reuse the date values of one object by other objects.
- (g) Polymorphism is extensively used in implementing inheritance.
- (h) Object-oriented programs are executed much faster than conventional programs.
- (i) Object-oriented systems can scale up better from small to large.
- (G) Object-oriented approach cannot be used to create databases.

## **Chapter 2**



### **JAVA HISTORY**

Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991. Originally called *Oak* by James Gosling, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices like TVs, VCRs, toasters and such other electronic machines. This goal had a strong impact on the development team to make the language simple, portable and highly reliable. The Java team which included Patrick Naughton discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. However, they modelled their new language Java on C and C++ but removed a number of features of C and C++ that were considered as sources of problems and thus made Java a really simple, reliable, portable, and powerful language. Table 2.1 lists some important milestones in the development of Java.

**Table 2.' Java Milestones**

Year	Development
1990	Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
1991	After exploring the possibility of using the most popular object-oriented language C++, the team announced a new language named "Oak".
1992	The team, known as Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
1993	The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team came up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.

*(Continued)*

Table 2.1 (Continued)

Year	Development
1994	The team developed a Web browser called "HotJava" to locate and run applet programs on Internet. HotJava demonstrated the power of the new language, thus making it instantly popular among the Internet users.
1995	Oak was renamed "Java", due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
1996	Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Java found its home.

The most striking feature of the language is that it is a platform-neutral language. Java is the first programming language that is not tied to any particular hardware or operating system. Programs developed in Java can be executed anywhere on any system. We can call Java as a revolutionary technology because it has brought in a fundamental shift in how we develop and use programs. Nothing like this has happened to the software industry before.



## JAVA FEATURES

The inventors of Java wanted to design a language which could offer solutions to some of the problems encountered in modern programming. They wanted the language to be not only reliable, portable and distributed but also simple, compact and interactive. Sun Microsystems officially describes Java with the following attributes:

- Compiled and Interpreted
- Platform-Independent and Portable
- Object-Oriented
- Robust and Secure
- Distributed
- Familiar, Simple, and Small
- Multithreaded and Interactive
- High Performance
- Dynamic and Extensible

Although the above appears to be a list of buzzwords, they aptly describe the full potential of the language. These features have made Java the first application language of the World Wide Web. Java will also become the premier language for general purpose stand-alone applications.

### Compiled and Interpreted

Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system. First, Java compiler translates source code into what is known as *bytecode* instructions. Bytecodes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly

executed by the machine that is running the Java program. We can thus say that Java is both a compiled and an interpreted language.

### **Platform-Independent and Portable**

The most significant contribution of Java over other languages is its portability. Java programs can be easily moved from one computer system to another, anywhere and anytime. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs. This is the reason why Java has become a popular language for programming on Internet which interconnects different kinds of systems worldwide. We can download a Java applet from a remote computer onto our local system via Internet and execute it locally. This makes the Internet an extension of the user's basic system providing practically unlimited number of accessible applets and applications.

Java ensures portability in two ways. First, Java compiler generates bytecode instructions that can be implemented on any machine. Secondly, the size of the primitive data types are machine-independent.

### **Object-Oriented**

Java is a true object-oriented language. Almost everything in Java is an *object*. All program code and data reside within objects and classes. Java comes with an extensive set of *classes*, arranged in *packages*, that we can use in our programs by inheritance. The object model in Java is simple and easy to extend.

### **Robust and Secure**

Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures series errors and eliminates any risk of crashing the system.

Security becomes an important issue for a language that is used for programming on Internet. Threat of viruses and abuse of resources is everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

### **Distributed**

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

### **Simple, Small and Familiar**

Java is a small and simple language. Many features of C and C++ that are either redundant or sources of unreliable code are not part of Java. For example, Java does not use pointers, preprocessor header files, goto statement and many others. It also eliminates operator overloading and multiple inheritance. For more detailed comparison of Java with C and C++, refer to Section 2.3.

Familiarity is another striking feature of Java. To make the language look familiar to the existing programmers, it was modelled on C and C++ languages. Java uses many constructs of C and C++ and therefore, Java code "looks like a C++" code. In fact, Java is a simplified version of C++.

### **Multithreaded and Interactive**

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another. For example, we can listen to an audio clip while scrolling a page and at the same time download an applet from a distant computer. This feature greatly improves the interactive performance of graphical applications.

The Java runtime comes with tools that support multiprocess synchronization and construct smoothly running interactive systems.

### **High Performance**

Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode. According to Sun, Java speed is comparable to the native C/C++. Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of Java programs.

### **Dynamic and Extensible**

Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods, and objects. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

Java programs support functions written in other languages such as C and C++. These functions are known as native *methods*. This facility enables the programmers to use the efficient functions available in these languages. Native methods are linked dynamically at runtime.



## **HOW JAVA DIFFERS FROM C AND C++**

**Although Java was modelled after C and C++ languages, it differs from C and C++ in many ways. Java does not incorporate a number of features available in C and C++.** For the benefit of

C and c++ programmers, we point out here a few major differences between C/C++ and Java languages.

## **Java and C**

Java is a lot like C but the major difference between Java and C is that Java is an object-oriented language and has mechanism to define classes and objects. In an-effort to build a simple and safe language, the Java team did not include some of the C features in Java.

- Java does riot include the Cunique statement keywords goto, sizeof, and typedef.
- Java does not contain the data types struct, union and enum .
- Java does not define the type modifiers keywords auto, extern, register, signed, and unsigned.
- Java does not support an explicit pointer type.
- Java does not have a preprocessor and therefore we cannot use # define, # include, and # ifdef statements.
- Java does not support any mechanism for defining variable arguments to functions.
- Java: requires that the functions with no arguments must be declared with empty parenthesis and not with the void keyword as done in C.
- Java adds new operators such as instanceof and >>.
- Java adds labelled break and continue statements.
- Java adds many features required for object-oriented programming.

## **Java and C++**

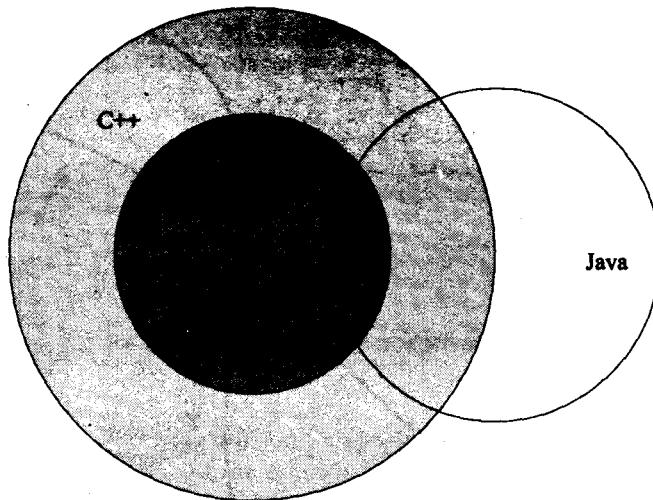
Java is a true object-oriented language while c++ is basicallyC with object-oriented extension. That is what exactly the increment operator ++ indicates. C++ has maintained backward compatibility with C. It is therefore possible to write an old style C program and run it successfully under C++. Java appears to be similar to C++ when we consider only the "extension" part of C++. However, some object-oriented features of C++ make the C++ code extremely difficult to follow and maintain.

Listed below are some major C<sub>+</sub> + features that were intentionally omitted from Java or significantly modified .

- Java does not support operator overloading.
- Java does not have template classes as in C++ .
- Java does not support multiple inheritance of classes. This is accomplished using a new feature called "interface".
- Java does not support global variables. Every variable and method is declared within a class and forms part' of that class.
- Java does not use pointers.
- Java has replaced the destructor function with a finalize() function.
- There are no header files in Java.

Java also adds some new features. While C++ is a superset of C, Java is neither a superset nor a subset of C or C++. Java may be considered as a first cousin of C++ and a second cousin of C as illustrated in Fig. 2.1. A more detailed discussion on the differences between C++ and Java is available in Appendix: C.

**Fig. 2.1**



*Overlapping of C, C++, and Java*



## JAVA AND INTERNET

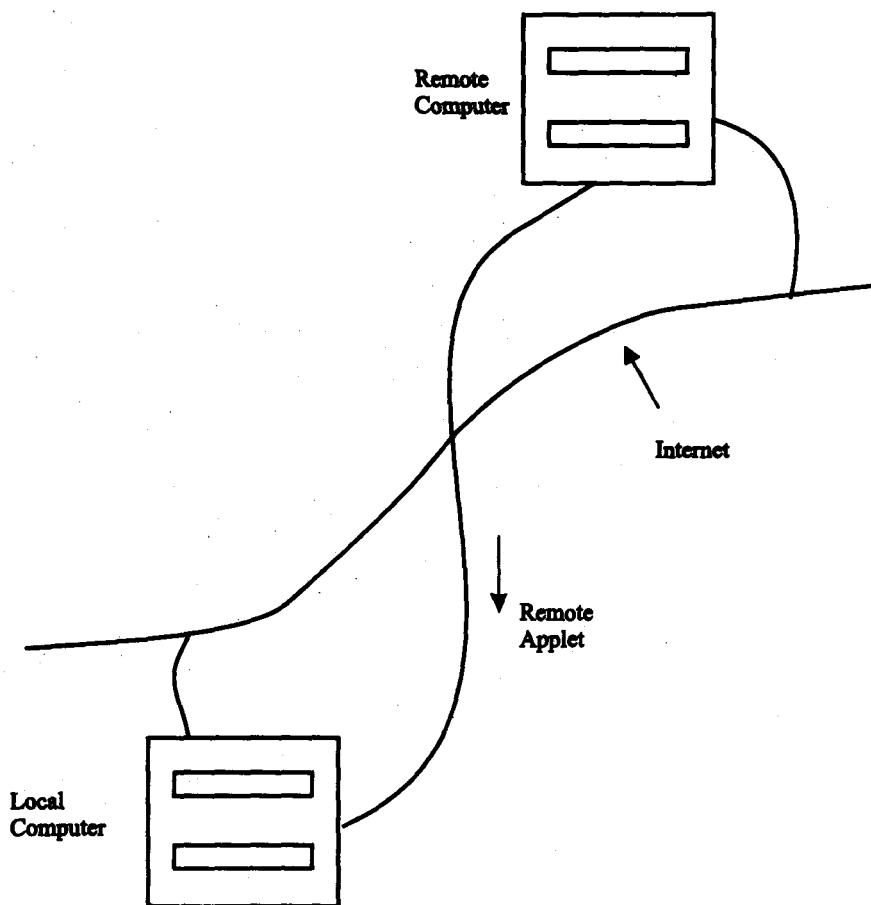
Java is strongly associated with the Internet because of the fact that the first application program written in Java was HotJava, a Web browser to run applets on Internet. Internet users can use Java to create applet programs and run them locally using a "Java-enabled browser" such as HotJava. They can also use a Java-enabled browser to download an applet located on a computer anywhere in the Internet and run it on his local computer (see Fig. 2.2). In fact, Java applets have made the Internet a true extension of the storage system of the local computer.

Internet users can also set up their Web sites containing Java applets that could be used by other remote users of Internet. The ability of Java applets to hitch a ride on the Information Superhighway has made Java a unique programming language for the Internet. In fact, due to this, Java is popularly known as Internet language.



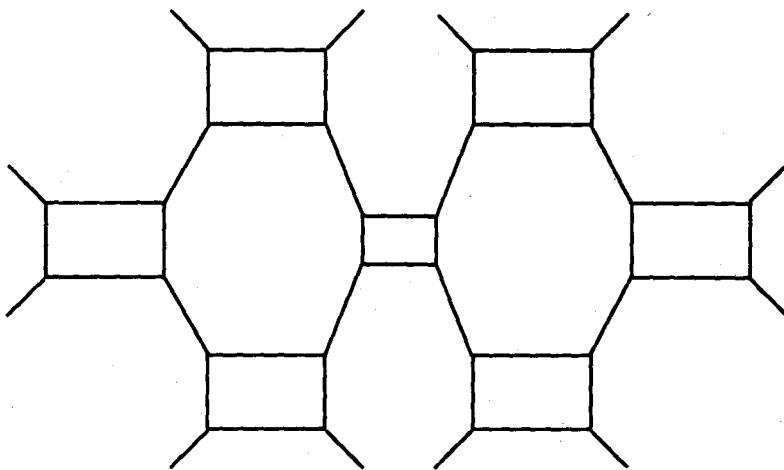
## JAVA AND WORLD WIDE WEB

World Wide Web (WWW) is an open-ended information retrieval system designed to be used in the Internet's distributed environment. This system contains what are known as Web pages that provide both information and controls. Unlike a menu-driven system where we are guided

**Fig. 2.2****Downloading of applets via Internet**

through a particular direction using a decision tree structure, the Web system is open-ended and we can navigate to a new document in any direction as shown in Fig. 2.3. This is made possible with the help of a language called *Hypertext Markup Language* (HTML). Web pages contain HTML tags that enable us to find, retrieve, manipulate and display documents worldwide.

Java was meant to be used in distributed environments such as Internet. Since, both the Web and Java share the same philosophy, Java could be easily incorporated into the Web system. Before Java, the World Wide Web was limited to the display of still images and texts. However, the incorporation of Java into Web pages has made it capable of supporting animation, graphics, games, and a wide range of special effects. With the support of Java, the Web has become more interactive and dynamic. On the other hand, with the support of Web, we can run a Java program on someone else's computer across the Internet.

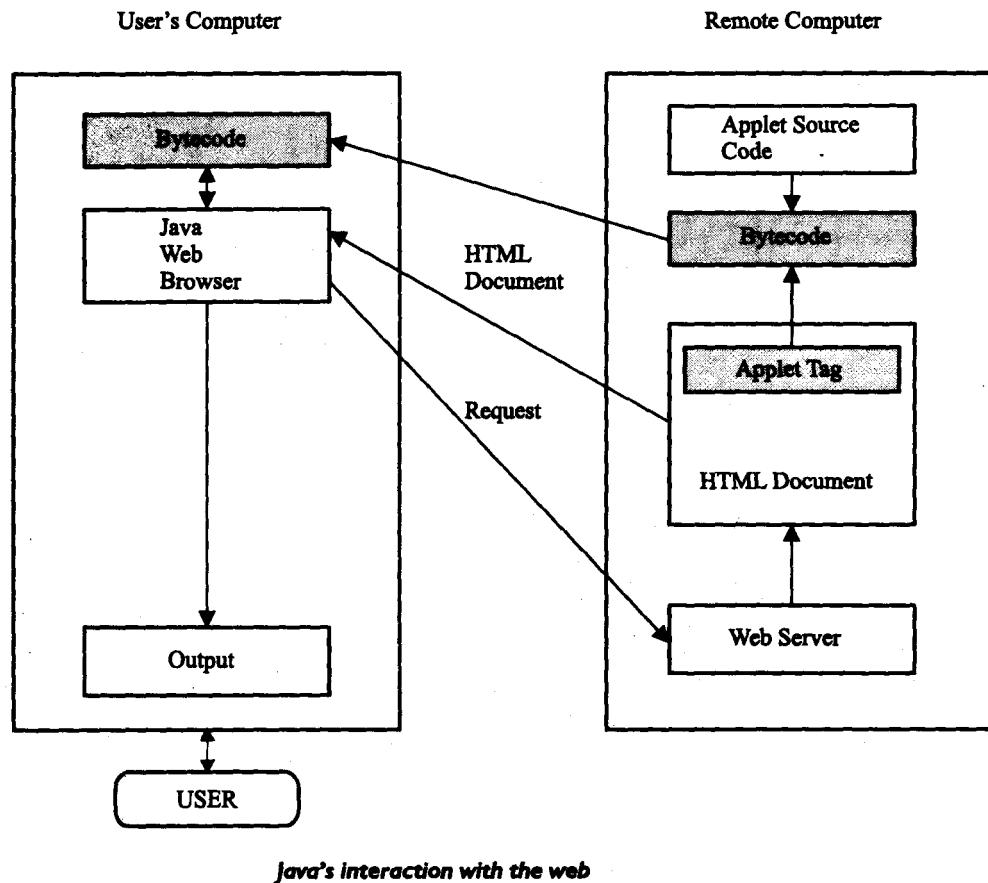
**Fig. 2.3****Web structure of information search**

Java communicates with a Web page through a special tag called <APPLET>. Figure 2.4 illustrates this process. The figure shows the following communication steps:

1. The user sends a request for an HTML document to the remote computer's Web server. The Web server is a program that accepts a request, processes the request, and sends the required document.
2. The HTML document is returned to the user's browser. The document contains the APPLET tag, which identifies the applet.
3. The corresponding applet bytecode is transferred to the user's computer. This bytecode had been previously created by the Java compiler using the Java source code file for that applet.
4. The Java-enabled browser on the user's computer interprets the bytecodes and provides output.
5. The user may have further interaction with the applet but with no further downloading from the provider's Web server. This is because the bytecode contains all the information necessary to interpret the applet.

## **2.4 WEB BROWSERS**

As pointed out earlier, the Internet is a vast sea of information represented in many formats and stored on many computers. A large portion of the Internet is organized as the World Wide Web which uses hypertext. Web browsers are used to navigate through the information found on the net. They allow us to retrieve the information spread across the Internet and display it using the hypertext markup language (HTML). Examples of Web browsers, among others, include:

**Fig. 2.4**

- HotJava
- Netscape Navigator
- Internet Explorer

HTML documents and <APPLET> tags are discussed in detail in Chapter 14.

## HotJava

HotJava is the Web browser from Sun Microsystems that enables the display of interactive content on the Web, using the Java language. HotJava is written entirely in Java and demonstrates the capabilities of the Java programming language.

When the Java language was first developed and ported to the Internet, no browsers were available that could run Java applets. Although we can view a Web page that includes Java applets with a regular browser, we will not gain any of Java's benefits. HotJava is currently available for the SPARC/Solaris platform as well as Windows 95 and Windows NT. So far as being a Web browser goes, it is nothing special and does not offer anything special that most other

Web browsers don't offer. Its biggest draw is that it was the first Web browser to provide support for the Java language, thus making the Web more dynamic and interactive.

### Netscape Navigator

Netscape Navigator, from Netscape Communications Corporation, is a general-purpose browser that can run Java applets. With versions available for Windows 95, NT, Solaris and Apple Macintosh, Netscape Navigator is one of the most widely used browsers today.

Netscape Navigator has many useful features such as visual display about downloading process and indication of the number bytes downloaded. It also supports JavaScript, a scripting language used in HTML documents.

### Internet Explorer

Internet Explorer is another popular browser developed by Microsoft for Windows 95 and NT Workstations. Both the Navigator and Explorer use tool bars, icons, menus and dialog boxes for easy navigation. Explorer uses a just-in-time (JIT) compiler which greatly increases the speed of execution.



## HARDWARE AND SOFTWARE REQUIREMENTS

Java is currently supported on Windows 95, Windows NT, Sun Solaris, Macintosh, and UNIX machines. Though, the programs and examples in this book were tested under Windows 95, the most popular operating system today, they can be implemented on any of the above systems.

The minimum hardware and software requirements for Windows 95 version of Java are as follows:

- ◆ IBM-compatible 486 system
- ◆ Minimum of 8 MB memory
- ◆ Windows 95 software
- ◆ A Windows-compatible sound card, if necessary
- ◆ A hard drive
- ◆ A CD-ROM drive
- ◆ A Microsoft-compatible mouse



## JAVA SUPPORT SYSTEMS

It is clear from the discussion we had up to now that the operation of Java and Java-enabled browsers on the Internet requires a variety of support systems. Table 2.2 lists the systems necessary to support Java for delivering information on the Internet.

**Table 2.2 Java Support Systems**

Support System	Description
Internet Connection	Local computer should be connected to the Internet.
Web Server	A program that accepts requests for information and sends the required documents.
Web Browser	A program that provides access to WWW and runs Java applets.

(Continued)

**Table 2.2 (Continued)**

Support System	Description
HTML	A language for creating hypertext for the Web.
APPLET Tag	For placing Java applets in HTML document.
Java Code	Java code is used for defining Java applets.
Bytecode	Compiled Java code that is referred to in the APPLET tag and transferred to the user computer.



## JAVA ENVIRONMENT

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of the system known as *Java Development Kit* (JDK) and the classes and methods are part of the *Java Standard Library* (JSL).

### Java Development Kit

The Java Development Kit comes with a collection of tools that are used for developing and running Java programs. They include:

- ◆ appletviewer ( for viewing Java applets )
- ◆ javac ( Java compiler )
- ◆ java ( Java interpreter )
- ◆ javap ( Java disassembler )
- ◆ javah ( for C header files )
- ◆ javadoc ( for creating HTML documents )
- ◆ jdb ( Java debugger )

Table 2.3 lists these tools and their descriptions.

**Table 2.3 Java Development Tools**

Tool	Description
appletviewer	Enables us to run Java applets (without actually using a Java-compatible browser).
java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javac	The Java compiler, which translates Java sourcecode to bytecode files that the interpreter can understand.
javadoc	Creates HTML-format documentation from Java source code files.
javah	Produces header files for use with native methods.
javap	Java disassembler, which enables us to convert bytecode files into a program description.
jdb	Java debugger, which helps us to find errors in our programs.

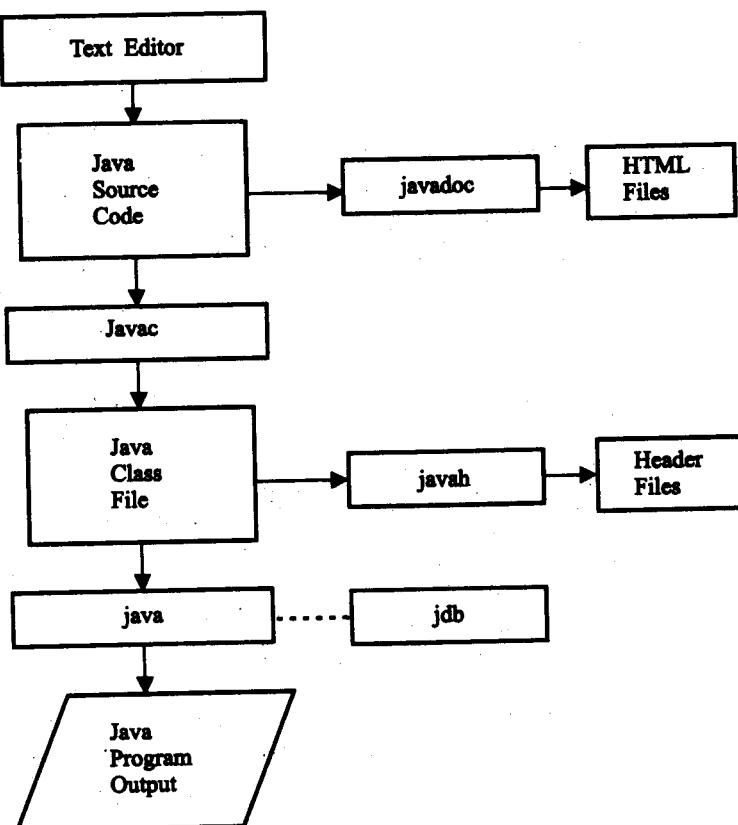
The way these tools are applied to build and run application programs is illustrated in Fig. 2.5. To create a Java program, we need to create a source code file using a text editor. The source code is then compiled using the Java compiler javac and executed using the `java` interpreter `java`. The Java debugger `jdb` is used to find errors, if any, in the source code. A compiled Java program can be converted into a source code with the help of Java disassembler `javap`. We learn more about these tools as we work through the book.

## Java Standard Library

The Java Standard Library includes hundreds of classes and methods grouped into six functional packages .

- Language Support Package: A collection of classes and methods required for implementing basic features of Java.
- Utilities Package: A collection of classes to provide utility functions such as date and time functions.

**Fig. 2.5**



**Process of building and running Java application programs**

- Input/Output Package: A collection of classes required for input/output manipulation.
- Networking Package: A collection of classes for communicating with other computers via Internet.
- AWT Package: The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.
- Applet Package: This includes a set of classes that allows us to create Java applets.

The use of these library classes will become evident when we start developing Java programs.

## SUMMARY

In this chapter, we have introduced a brief history of Java and its salient features. Java is a pure object-oriented language introduced by Sun Microsystems of USA and has a number of characteristics that make it suitable for Internet programming. We have discussed briefly how Java can be incorporated into the World Wide Web system with the help of Web browsers.

We have also brought out some of the fundamental differences between Java and C/C++ languages. Finally, we discussed the environment required and various tools available for implementing Java programs.

### KEY TERMS

**Object, Internet, World Wide Web, Applets, Package, Platform-neutral, Multithread, Bytecode, Dynamic linking, Native methods, HTML, Web browser, Applet tag, Web server, HotJava, Netscape Navigator, appletviewer, java, javac, javap, javah, javadoc, jdb.**

## REVIEW QUESTIONS

---

- 2.1 Why is Java known as platform-neutral language?
- 2.2 How is Java more secured than other languages?
- 2.3 What is multithreading? How does it improve the performance of Java?
- 2.4 List at least ten major differences between C and Java.
- 2.5 List at least five major C++ features that were intentionally removed from Java.
- 2.6 How is Java strongly associated with the Internet?
- 2.7 What is World Wide Web? What is the contribution of Java to the World Wide Web?
- 2.8 What is Hypertext Markup Language? Describe its role in the implementation of Java applets.
- 2.9 Describe the various systems required for Internet programming?

2.10 Describe with a flowchart, how various Java tools are used in the application development.

---

## **Chapter 3**

# **Overview of Java Language**



## **INTRODUCTION**

Java is a general-purpose, object-oriented programming language. We can develop two types of Java programs:

- Stand-alone applications
- Web applets

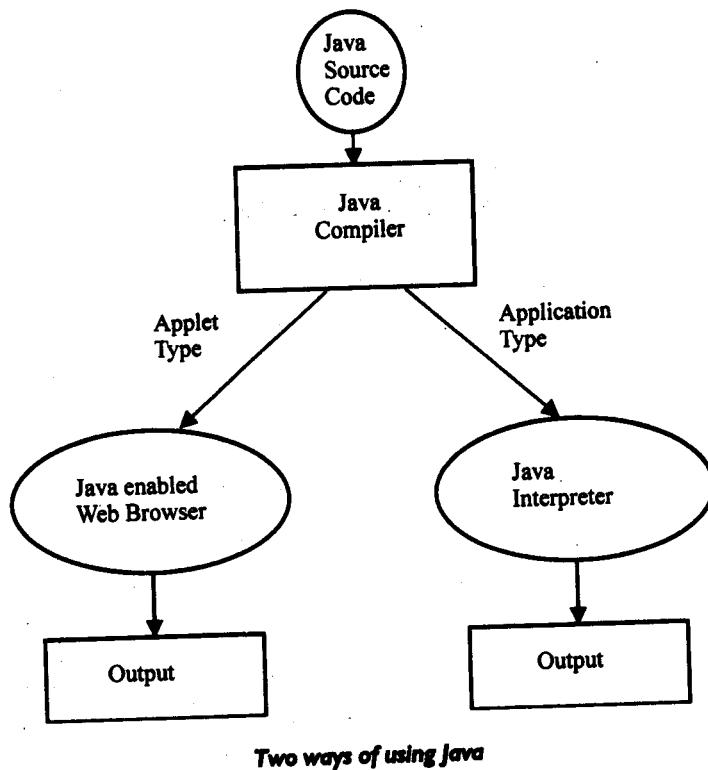
They are implemented as shown in Fig. 3.1. Stand-alone applications are programs written in Java to carry out certain tasks on a stand-alone local computer. In fact, Java can be used to develop programs for all kinds of applications, which earlier, were developed using languages like C and C++. As pointed out earlier, HotJava itself is a Java application program. Executing a stand-alone Java program involves two steps:

1. Compiling source code into bytecode using Javac compiler
2. Executing the byte code program using Java interpreter.

Applets are small Java programs developed for Internet applications. An applet located on a distant computer (Server) can be downloaded via Internet and executed on a local computer (Client) using a Java-capable browser. We can develop applets for doing everything from simple animated graphics to complex games and utilities. Since applets are embedded in an HTML (Hypertext Markup Language) document and run inside a Web page, creating and running applets are more complex than creating an application.

Stand-alone programs can read and write files and perform certain operations that applets cannot do. An applet can only run within a Web browser.

In this chapter, we shall consider some simple application programs, which would demonstrate the general structure of Java application programs. We shall also discuss here the basic elements of Java language and steps involved in executing a Java application program. Creation of applets will be discussed later in Chapter 14.

Fig. 3.1

### **SIMPLE JAVA PROGRAM**

The best way to learn a new language is to write a few simple example programs and execute them. We begin with a very simple program that prints a line of text as output.

#### **Program 3.1 A simple Java program**

---

```

class SampleOne
{
    public static void main(String args[])
    {
        System.out.println("Java is better than C++.");
    }
}

```

---

Program 3.1 is perhaps the simplest of all Java programs. Nevertheless, it brings out some salient features of the language. Let us therefore discuss the program line by line and understand the unique features that constitute a Java program.

## Class Declaration

The first line

```
class SampleOne
```

declares a class, which is an object-oriented construct. As stated earlier, Java is a true object-oriented language and therefore, every keyword must be placed inside a class. `class` is a keyword and declares that a new class definition follows. `SampleOne` is a Java *identifier* that specifies the name of the class to be defined.

## OpenIna ~

Every class definition in Java begins with an opening brace "{" and ends with a matching closing brace "}", appearing in the last line in the example. This is similar to C++ class construct. (*Note that a class ~ in C++ ends With a semicolon.*)

The main Line

The third line

```
public static void main(String args[ ])
```

defines a method named `main`. Conceptually, this is similar to the `main()` function in C/C++. Every Java application program must include the `main()` method. This is the starting point for the interpreter to begin the execution of the program. A Java application can have any number of classes but only one of them must include a `main` method to initiate the execution. (Note that Java applets will not use the `main` method at all.)

This line contains a number of keywords, `public`, `static` and `void`.

`public:` The keyword `public` is an access specifier that declares the `main` method as unprotected and therefore making it accessible to all other classes. This is similar to the C++ `public` modifier.

`static:` Next appears the keyword `static`, which declares this method as one that belongs to the entire class and not a part of any objects of the class. The `static` must always be declared as `static` since the interpreter uses this method before any objects are created. More about static methods and variables will be discussed later in Chapter 8.

`Void:` The type modifier `void` states that the `main` method does not return any value (but simply prints some text to the screen.)

All parameters to a method are declared inside a pair of parentheses. Here, `String args()` declares a parameter named `args`, which contains an array of objects of the class type `String`.

## The Output Un.

The only executable statement in the program is

```
System.out.println("Java is better than C++.");
```

This is similar to the printf( ) statement of C or cout << construct of C++. Since Java is a true object oriented language, every method must be part of an object. The println method is a member of the out object, which is a static data member of System class. This line prints the string

```
'Java is better than C++.
```

to the screen. The method println always appends a newline character to the end of the string. This means that any subsequent output will start on a new line. Note the semicolon at the end of the statement. *Every Java statement must end with a semicolon.* (Saving, compiling, and executing a Java program are discussed in Section 3.8)



## MORE OF JAVA

Assume that we would like to compute and print the square root of a number. A Java program to accomplish this is shown in Program 3.2. This is a slightly complex program. This program when compiled and run produces the output

```
y = 2.23607
```

---

### **Program 3.2 A Java program with multiple statements**

---

```
/*
 * More Java statements
 * This code computes square root
 */
import java.lang.Math;
class SquareRoot
{
    public static void main (String args[])
    {
        double x = 5 ; // Declaration and initialization
        double y; // Simple declaration
        y = Math.sqrt(x) ;
        System.out.println(y = ~ + y);
    }
}
```

---

The structure of the program is similar to the previous one except that it has more number of statements. The statement

```
double x = 5 ;
```

declares a variable x and initializes it to the value 5 and the statement

```
double yr
```

merely declares a variable y. Note that both of them have been declared as double type variables.. (double is a data type used to represent a floating point number. Data types are discussed in the next chapter).

The statement

```
y = Math.sqrt(x);
```

invokes the method sqrt of the Math class, computes square root of x and then assigns the result to the variable y. The output statement

```
System.out.println("y = " + y);
```

displays the result on the Screen as

```
y = 2.23607
```

Note the use of + symbol. Here, the + acts as the concatenation operator of two strings. The value of y is converted into a string representation before concatenation.

## Use of Math Functions

Note that the first statement in the program is

```
import java.lang.Math;
```

The purpose of this statement is to instruct the interpreter to load the Math class from the package lang. (This statement is similar to #include statement in C.) Remember, Math class contains the sqrt method required in the program.

## Comments

Java permits both the single-line comments and multi-line comments available in C++. The single-line comments begin with // and end at the end of the line as shown on the lines declaring x and y. For longer comments, we can create long multi-line comments by starting with /\* and ending with \*/ as shown at the beginning of the program.

## 3.4 AN APPLICATION WITH TWO CLASSES

Both the examples discussed above use only one class that contains the main method. A real-life application will generally require multiple classes. Program 3.3 illustrates a Java application with two classes.

### Pr0gram 3.3 A pr.gram with multiple classes

---

```

class Room
{
    float length;
    float breadth;

    void getdata(float a, float b)
    {
        length = a;
        breadth = b
    }
}

class RoomArea
{
    public static void main(String args[])
    {
        float area;
        Room room1 = new Room(); // Creates an object room1
        room1.getdata(14, 10);
        area = room1.length * room1.breadth;
        System.out.println("Area = " + area);
    }
}

```

---

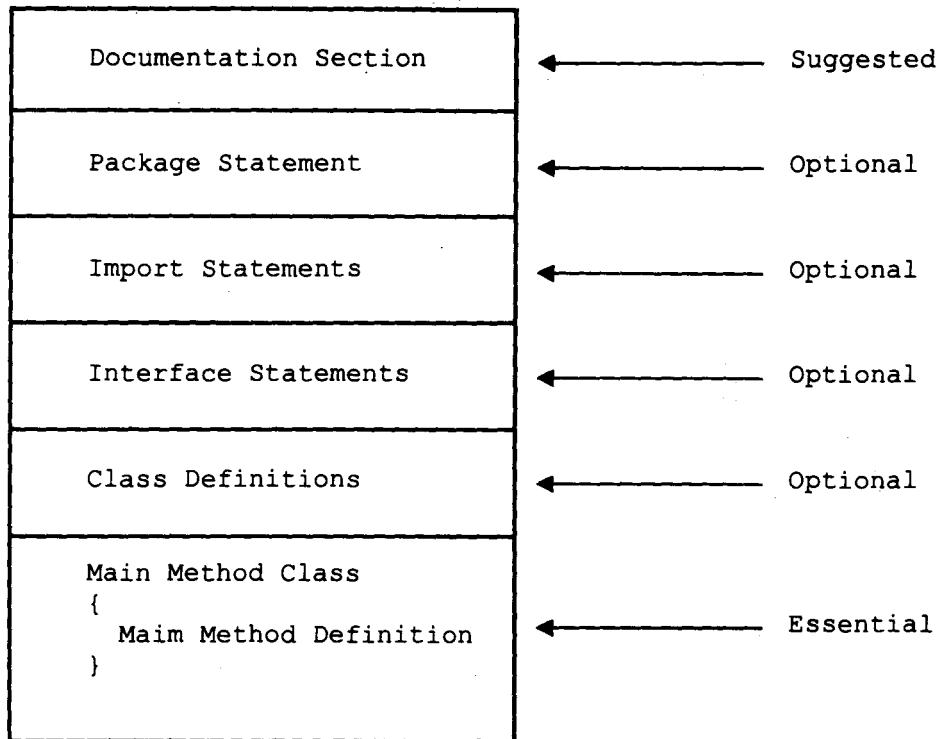
Pr0gram 3.3 defines two classes Room and RoomArea. The Room class defines two variables and one method to assign values to these variables. The class RoomArea contains the main method that initiates the execution.

The main method declares a local variable area and a Room type object room! and then assigns values to the data members of Room class by using the getdata method. Finally, it calculates the area and prints the results. Note the use of dot operator to access the variables and methods of Room class. Classes and methods are discussed in Chapter 8. The use of the keyword new is explained later in this Chapter.



## JAVA PROGRAM STRUCTURE

AB we have seen in the previous examples, a Java program may contain many classes of which only one class defines a main method. Classes contain data members and methods that operate on the data members of the class. Methods may contain data type declaratioRs and executable statements. To write a Java program, we first define classes and then put them together. A Java program may contain one or more 'sections' as shown in Fig. 3.2

**Fig. 3.2****General structure of a Java program**

### Documentadon Section

The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmer would like to refer to at a later stage. Comments must explain *why* and *what* of classes and *how* of algorithms. This would greatly help in maintaining the program. In addition to the two styles of comments discussed earlier, Java also uses a third style of comment ;•.... \*; known as documentation *comment*. This form of comment is used for generating documentation automatically.

### Package Statement

The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to this package. Example:

```
package student;
```

The package statement is optional. That is, our classes do not have to be part of a package. More about packages will be discussed in Chapter 11.

## Import Statements

The next thing after a package statement (but before any class definitions) may be a number of import statements. This is similar to the #include statement in C. Example:

```
import student. test;
```

This statement instructs the interpreter to load the test -class contained in the package student. Using import statements, we can have access to classes that are part of other named packages. More on import statements in Chapter 11.

## Interface Statements

An interface is like a class but includes a group of method declarations. This is also an optional section and is used only when we wish to implement the multiple inheritance feature in the program. Interface is a new concept in Java and is discussed in detail in Chapter 10.

## Class Definitions

A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program. These classes are used to map the objects of real-world problems. The number of classes used depends on the complexity of the problem.

## Main Method Class

Since every Java stand-alone program requires a main method as its starting point, this class is the essential part of a Java program. A simple Java program may contain only this part. The main method creates objects of various classes and establishes communications between them. On reaching the end of main, the program terminates and the control passes back to the operating system.

## JAVA TOKENS

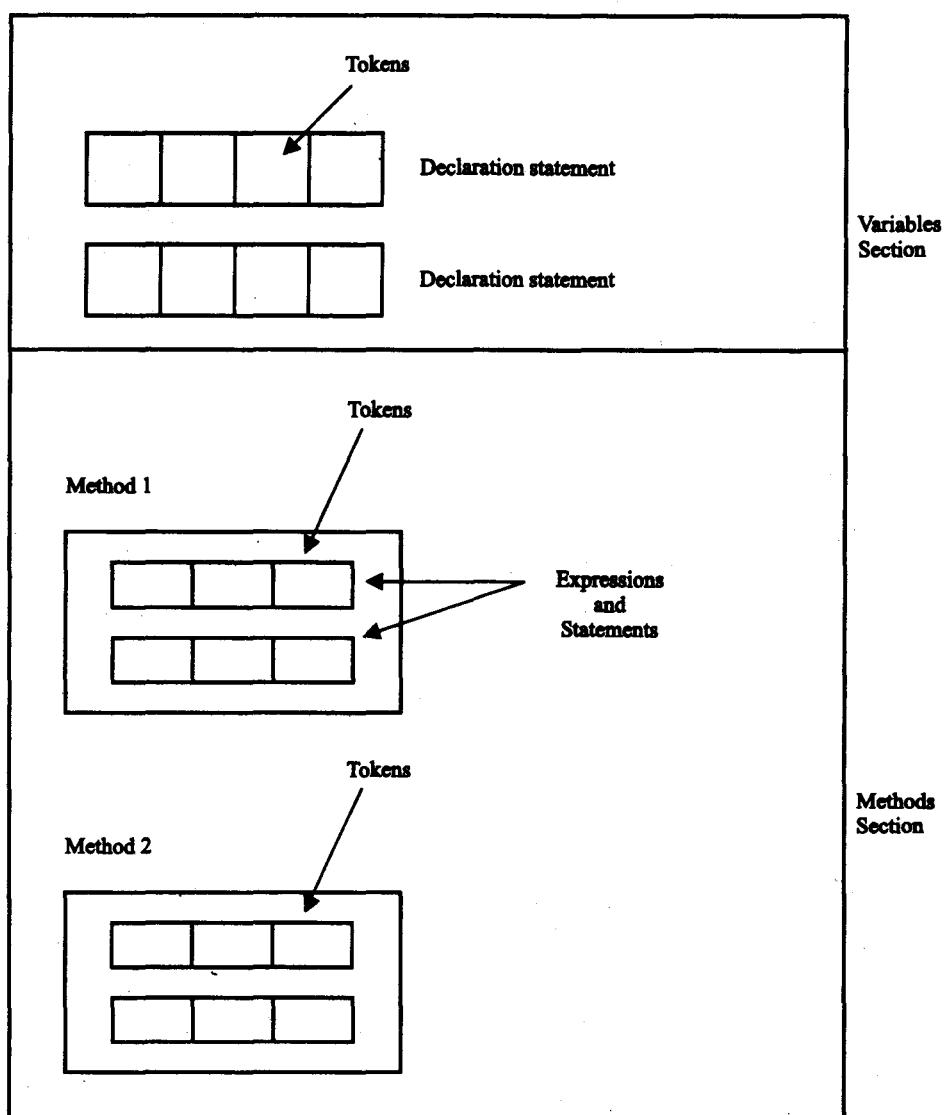
A Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements (see Fig. 3.3). Most statements contain expressions, which describe the actions carried out on data. Smallest individual units in a program are known as tokens. The compiler recognizes them for building up expressions and statements.

In simple term!! a Java program is a collection of tokens, comments and white spaces. Java language includes five types of tokens. They are:

- Reserved Keywords
- Identifiers
- Literals
- Operators
- Separators

Fig. 3.3

Java Class



*Elements of a Java class*

## Java Character Set

The smallest units of Java language are the characters used to write Java tokens. These characters are defined by the *Unicode* character set, an emerging standard that tries to **create** characters for a large number of scripts worldwide.

The Unicode is a 16-bit character coding system and currently supports more than 34,000 defined characters derived from 24 languages from America, Europe, Middle East, Africa and Asia (including India). However, most of us use only the basic ASCII characters, which include letters, digits and punctuation marks, used in normal English. We, therefore, have used only ASCII character set (a subset of UNICODE character set) in developing the programs in this book.

## Keywords

Keywords are an essential part of a language definition. They implement specific features of the language. Java language has reserved 60 words as keywords. Table 3.1 lists these keywords. These keywords, combined with operators and separators according to a syntax, form definition of the Java language. Understanding the meanings of all these words is important for Java programmers.

Since keywords have specific meaning in Java, we cannot use them as names for variables, classes, methods and so on. All keywords are to be written in lower-case letters. Since Java is case-sensitive, one can use these words as identifiers by changing one or more letters to upper case. However, it is a bad practice and should be avoided.

Although Java has been modelled on C and C++ languages, Java does not use many of C/C++ keywords and, on the other hand, had added as many as 27 new keywords to implement the new features of the language. The keywords that are unique to Java are shown in boldface.

**Table 3.1 Java Keywords**

<b>abstract</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>	<b>byvalue*</b>
<b>case</b>	<b>cast*</b>	<b>catch</b>	<b>char</b>	<b>class</b>
<b>const*</b>	<b>continue</b>	<b>default</b>	<b>do</b>	<b>double</b>
<b>else</b>	<b>extends</b>	<b>false**</b>	<b>final</b>	<b>finally</b>
<b>float</b>	<b>for</b>	<b>future*</b>	<b>generic*</b>	<b>goto*</b>
<b>if</b>	<b>implements</b>	<b>import</b>	<b>inner*</b>	<b>instanceof</b>
<b>int</b>	<b>interface</b>	<b>long</b>	<b>native</b>	<b>new</b>
<b>null**</b>	<b>operator*</b>	<b>outer*</b>	<b>package</b>	<b>private</b>
<b>protected</b>	<b>public</b>	<b>rest*</b>	<b>return</b>	<b>short</b>
<b>static</b>	<b>super</b>	<b>switch</b>	<b>synchronized</b>	<b>this</b>
<b>threadsafe*</b>	<b>throw</b>	<b>throws</b>	<b>transient</b>	<b>true**</b>
<b>try</b>	<b>var*</b>	<b>void</b>	<b>volatile</b>	<b>while</b>

\* Reserved for future use

\*\* These are values defined by Java

## Identifiers

Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules:

1. They can have alphabets, digits, and the underscore and dollar sign characters.
2. They must not begin with a digit.
3. Uppercase and lowercase letters are distinct.
4. They can be of any length.

Identifier must be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily read. Java developers have followed some naming conventions.

- Names of all public methods and instance variables start with a leading lowercase letter.

Examples:

```
average
sum
```

- When more than one word are used in a name, the second and subsequent words are marked with a leading uppercase letters. Examples:

```
dayTemperature
firstDayOfMonth
totalMarks
```

- All private and local variables use only lowercase letters combined with underscores.

Examples:

```
length
batch_strength
```

- All classes and interfaces start with a leading uppercase letter(and each subsequent word with a leading uppercase letter). Examples:

```
Student
HelloJava
Vehicle
MotorCycle
```

- Variables that represent constant values use all uppercase letters and underscores between words. Examples:

```
TOTAL
F MAX
PRINCIPAL AMOUNT
```

They are like symbolic constants in C.

It should be remembered that all these are conventions and not rules. We may follow our own conventions as long as we do not break the basic rules of naming identifiers.

## Literals

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Java language specifies five major types of literals. They are:

- Integer literals
- Floating-point literals
- Character literals
- ~ String literals
- Boolean literals

Each of them has a type associated with it. The type describes how the values behave and how they are stored. We will discuss these in detail when we deal with data types and constants in the next chapter.

## Operators

An operator is a symbol that takes one or more arguments and *operates* on them to produce a result. Operators are of many types and are considered in detail in Chapter 5.

## Separators

Separators are symbols used to indicate where groups of code are divided and arranged. They basically define the shape and function of our code. Table 3.2 lists separators and their functions.

Table 3.2 Java Separation

Name	What it is used for
parentheses( )	Used to enclose parameters in method definition and invocation, also used for defining precedence in expressions, containing expressions for now control, and surrounding cast types
braces{ }	Used to contain the values of automatically initialized arrays and to define a block of code for classes, methods and local scopes
brackets ( )	Used to declare array types and for dereferencing array values
=colon ;	Used to separate statements
comma ,	Used to separate consecutive identifiers in a variable declaration, also used to chain statements together inside a 'for' statement
period .	.Used to separate package names from sub-packages and classes; also used to separate a variable or method from a reference variable.

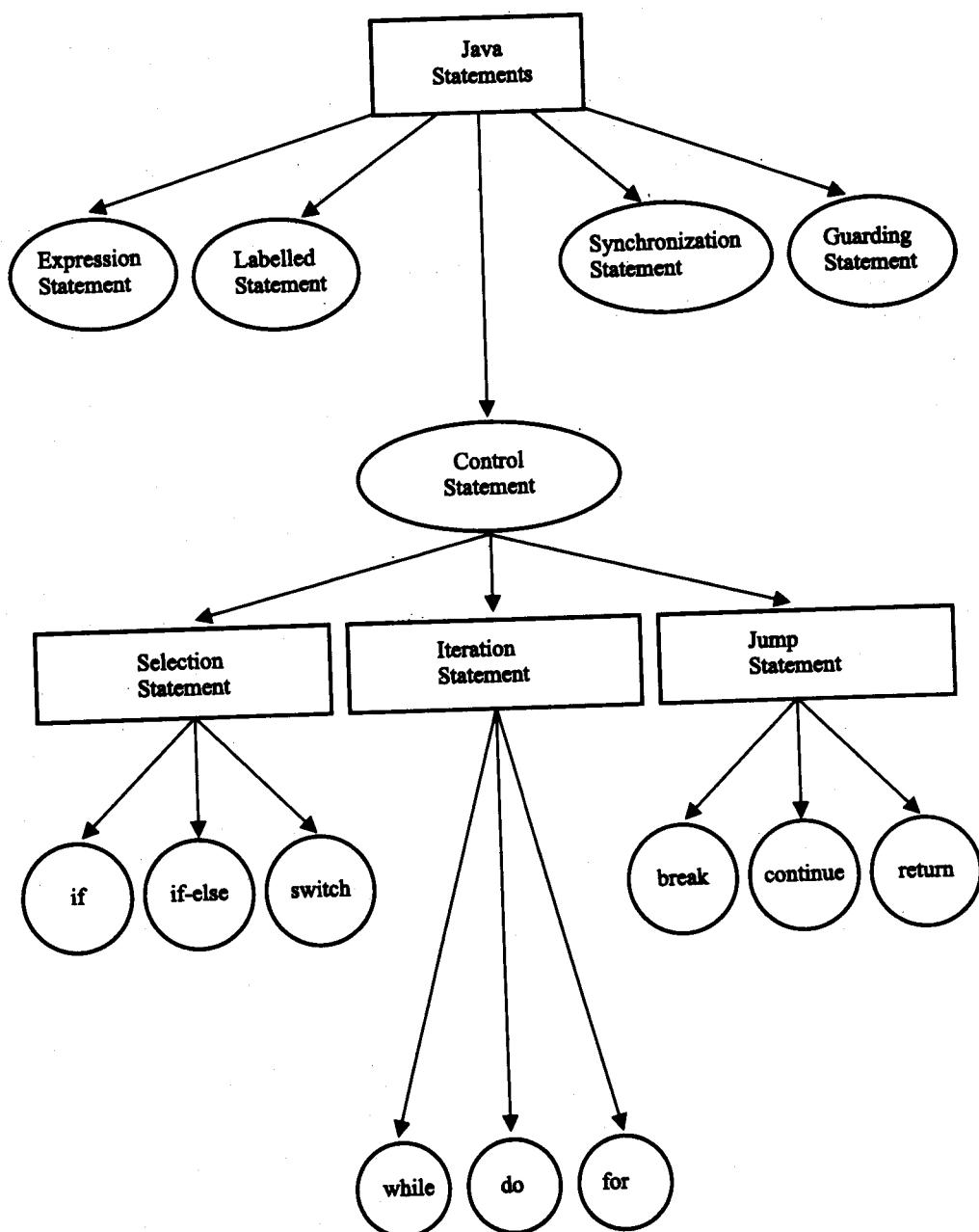


## JAVA STATEMENTS

The statements in Java are like sentences in natural languages. A statement is an executable combination of tokens ending with a semicolon ( ; ) mark. Statements are usually executed in sequence in the order in which they appear. However, it is possible to control the flow of execution, if necessary, using special statements. Java implements several types of statements as illustrated in Fig. 3.4 and described in Table 3.3. They are considered in depth as and when they are encountered.

**Table 3.3 Summary of Java Statements**

Statement	Description	Remarks
Empty Statement	These do nothing and are used during program development as a place holder.	same as C and C++
Labelled Statement	Any Statement may begin with a label. Such labels must not be keywords, already declared local variables, or previously used labels in this module. Labels in Java are used as the arguments of Jump statements, which are described later in this list.	Identical to C and C++ except their use with jump statements
Expression Statement	Most statements are expression statements. Java has seven types of Expression statements: Assignment, Pre-Increment, Pre-Decrement, Post-Increment, Post-Decrement, Method Call and Allocation Expression.	Same as C++
Selection Statement	These select one of several control flows. There are three types of selection statements in Java: if, if-else, and switch.	Same as C and C++
Iteration Statement	These specify how and when looping will take place. There are three types of iteration statements: while, do and for.'	Same as C and C++ except for jumps and labels
Jump Statement	Jump Statements pass control to the beginning or end of the current block, or to a labeled statement. Such labels must be in the same block, and continue labels must be on an iteration statement. The four types of Jump statement are break, continue, return and throw.	C and C++ do not use labels with jump statements.
Synchronization Statement	These are used for handling issues with multi-threading.	Not available in C and C++
Guarding Statement	Guarding statements are used for safe handling of code that may cause exceptions (such as division by zero). These statements use the keywords try, catch, and finally.	Same as in C++ except finally statement.



Classification of Java statements

**3.8****IMPLEMENTING A JAVA PROGRAM**

Implementation of a Java application program involves a series of steps. They include:

- Creating the program
- Compiling the program
- Running the program.

Remember that, before we begin creating the program, the Java Development Kit (JDK) must be properly installed on our system.

### **Creating the Program**

We can create a program using any text editor. Assume that we have entered the following program:

---

#### **Program 3.4 Another simple program for testing**

---

```
class Test
{
    public static void main (String args[])
    {
        System.out.println("Hello!");
        System.out.println("Welcome to the world of Java.");
        System.out.println("Let us learn Java.");
    }
}
```

---

We must save this program in a file called `Test.java` ensuring that the filename contains the class name properly. This file is called the *source file*. Note that all Java source files will have the extension `.java`. Note also that if a program contains multiple classes, the file name must be the classname of the class containing the `main` method.

### **Compiling the Program**

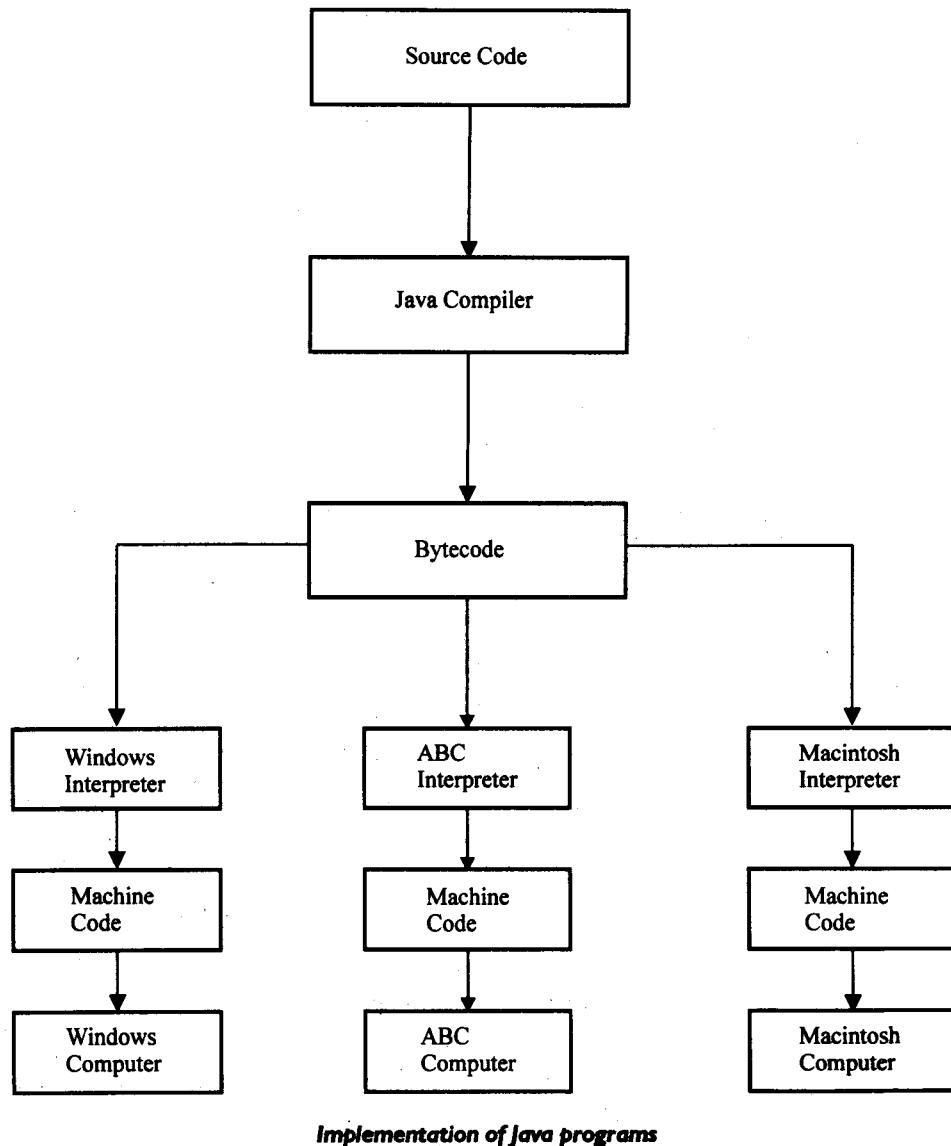
To compile the program, we must run the Java Compiler `javac`, with the name of the source file on the command line as shown below:

`javac Test.java`

If everything is OK, the `javac` compiler creates a file called `Test.class` containing the bytecodes of the program. Note that the compiler automatically names the bytecode file as

`<classname>.class`

Fig. 3.5



## Running the Program

We need to use the Java interpreter to run a stand-alone program. At the command prompt, type

Java Test

Now, the interpreter looks for the main method in the program and begins execution from there. When executed, our program displays the following:

Hello!  
Welcome to the world of Java.  
Let us learn Java.

Note that we simply type "Test" at the command line and not "Test.class" or "Test.java".

## Machine Neutral

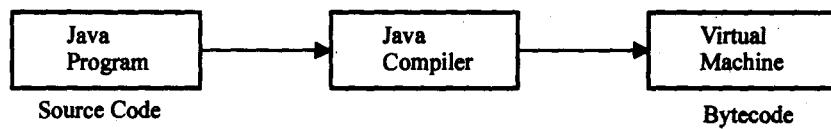
The compiler converts the source code files into bytecode files. These codes are machine-independent and therefore can be run on any machine. That is, a program compiled on an IBM machine will run on a Macintosh machine.

Java interpreter reads the bytecode files and translates them into machine code for the specific machine on which the Java program is running. The interpreter is therefore specially written for each type of machine. Fig. 3.5 illustrates this concept.

## JAVA VIRTUAL MACHINE

All language compilers translate source code into *machine code* for a specific computer. Java compiler also does the same thing. Then, how does Java achieve architecture neutrality? The answer is that the Java compiler produces an intermediate code known as bytecode for a machine that does not exist. This machine is called the *Java Virtual Machine* and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer. Figure 3.6 illustrates the process of compiling a Java program into bytecode which is also referred to as *virtual machine code*.

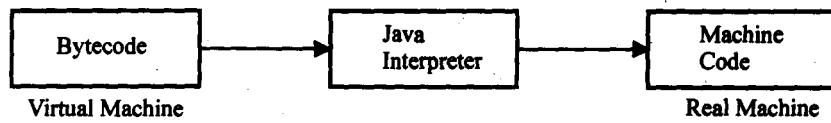
**Fig. 3.6**



**Process of compilation**

The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine as shown in Fig. 3.7. Remember that the interpreter is different for different machines.

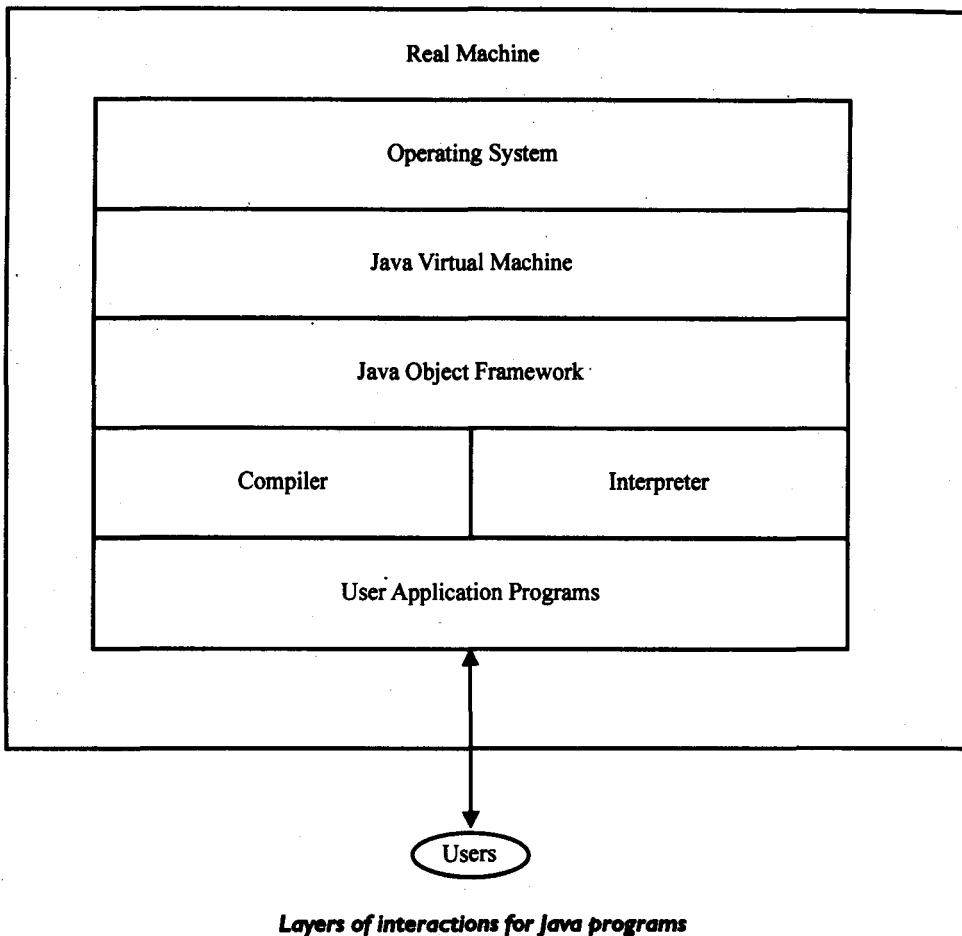
**Fig. 3.7**



**Process of converting bytecode into machine code**

Figure 3.8 illustrates how Java works on a typical computer. The Java object framework as the intermediary between the user programs and the virtual machine which in turn acts as the intermediary between the operating system and the Java object framework.

**Fig 3.8**



## **3.10 COMMAND LINE ARGUMENTS**

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in Java programs by using what are known as command line arguments. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution. It may be recalled that Program 3.4 was invoked for execution at the command line as follows:

```
java Test
```

Here, we have not supplied any command line arguments. Even if we supply arguments, the program does not know what to do with them.

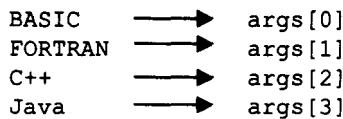
We can write Java programs that can receive and use the arguments provided in the command line. Recall the signature of the main( ) method used in our earlier example programs:

```
public static void main (String args[ ])
```

As pointed out earlier, args is declared as an array of strings (known as string objects). Any arguments provided in the command line (at the time of execution) are passed to the array args as its elements. We can simply access the array elements and use them in the program as we wish. For example, consider the command line

```
java Test BASIC FORTRAN C++ Java
```

This command line contains four arguments. These are assigned to the array args as follows:



The individual elements of an array are accessed by using an index or subscript like args[ i ]. The value of i denotes the position of the elements inside the array. For example, args[ 2 ] denotes the third element and represents C++. Note that Java subscripts begin with 0 and not 1. (Arrays and strings are discussed in detail in Chapter 9.)

### **Program 3.5 Use of command line arguments**

---

```
/*
 * This program uses command line
 * arguments as input.
 */
Class ComLineTest
{
    public static void main(String args[ ])
    {
        int count, i=0;
        String string;
        count = args.length;
        System.out.println("Number of arguments = " + count);
        while (i < count) .
        {
            string = args[i];
            i = i + 1;
            System.out.println(i+ " : " + "Java is " +string+ " !");
        }
    }
}
```

Program 3.5 illustrates the use of command line arguments. Compile and run the program with the command line as follows:

```
java ComLineT ••t Simple Object Oriented Distributed Robust Secure  
Portable. Multithreaded Dynamic
```

Upon execution, the command line arguments Simple, Object\_Oriented, etc. are passed to the program through the array args as discussed earlier. That is the element args[ 0 ] contains Simple, args[ 1 ] contains Object\_Oriented, and so on. These elements are accessed using the loop variable i as an index like

```
name = args[i]
```

The index i is incremented using a while loop until all the arguments are accessed. The number of arguments is obtained by statement

```
count = args.length;
```

The output of the program would be as follows:

```
Number of arguments = 8  
1 : Java is Simple!  
2 : Java is Object_Oriented!  
3 : Java is Distributed!  
4 : Java is Robust!  
5 : Java is Secure!  
6 : Java is Portable!  
7 : Java is Multithreaded!  
8 : Java is Dynamic!
```

Note how the output statement concatenates the strings while printing.

## PROGRAMMING STYLE

Java is a *freeform* language. We need not have to indent any lines to make the program work properly. Java system does not care where on the line we begin typing. While this may be a license for bad programming, we should try to use this fact to our advantage for producing readable programs. Although several alternate styles are possible, we should select one and try to use it with total consistency.

For example, the statement

```
SYstem.out.println("Java is Wonderful!")
```

can be written as

```
System.out.println  
("Java is Wonderful!");
```

or, even as

```
System
```

```
out
print in
(
uJava is Wonderful!"
);
```

In this book, we follow the format used in the example programs of this chapter.



## SUMMARY

Java is a general-purpose, object-oriented language. In this chapter, we have discussed some simple application programs to familiarize the readers with basic Java structure and syntax. We have also discussed the basic elements of the Java language and steps involved in creating and executing a Java application program ..

### KEY TERMS

**class, Identifier, main, public, static, void, string, args, println, double, Math, sqrt, package, import, Interface, Tokens, Literals, Operators, Separators, Unicode, Keyword, Statement, Virtual Machine, Command Line Arguments, Freeform.**

### REVIEW QUESTIONS

---

- 3.1 Describe the structure of a typical Java program.
- 3.2 Why do we need the import statement?
- 3.3 What is the task of the main method in a Java program?
- 3.4 What is a token? List the various types of tokens supported by Java.
- 3.5 Why can't we use a keyword as a variable name?
- 3.6 Enumerate the rules for creating identifiers in Java.
- 3.7 What are the conventions followed in Java for naming identifiers? Give examples.
- 3.8 What are separators? Describe the various separators used in Java.
- 3.9 What is a statement? How do the Java statements differ from those of C and C++?
- 3.10 Describe in detail the steps involved in implementing a stand-alone program.
- 3.11 What are command line arguments? How are they useful?
- 3.12 Java is freeform language. Comment.

# Chapter 4



## INTRODUCTION

A programming language is designed to process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing data is accomplished by executing a sequence of instructions constituting a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *Wammar*). Every program instruction must conform precisely to the syntax rules of the language.

Like any other language, Java has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to Java language.



## CONSTANTS

Constants in Java refer to fixed values that do not change during the execution of a program. Java supports several types of constants as illustrated in Fig. 4.1.

### Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer constants are:

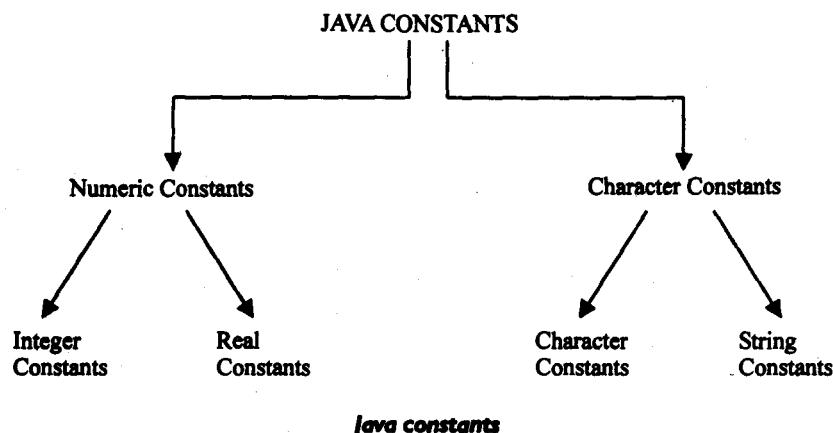
123      -321      0      654321

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 '750      20 ,000      \$1000

are illegal numbers.

**Fig. 4.1**



An *Octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading O. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by Ox or OX is considered as *hexadecimal* integer (hex integer). They may also include alphabets A through F or a through f. A letter A through F represents the numbers 10 through 15. Following are the examples of valid hex integers.

0x2            0x9F            0xbcd            0x

We rarely use octal and hexadecimal numbers in programming.

## Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or~*point*) constants. Further examples of real constants are:

0.0083 -0.75 435.36

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the *fractional* part, which is an integer. It is possible that the number may not have digits before the decimal point or, digits after the decimal point. That is,

215 .. 95 -.71

are all valid real numbers.

A real number may also be expressed in *scientific notation*. For example, the value 215.65 may be written as  $2.1565 \times 10^2$  in exponential notation.  $\times 10^2$  means multiply by 10<sup>2</sup>. The general form is:

***mantissa e exponent***

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The exponent is an integer with an optional *plus* or *minus* sign. The letter e separating the mantissa and the exponent can be written in either lowercase-or: uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in floating point *form*. Examples of legal floating point constants are:

0.65e4      12e-2      1.5e+5      3.18E3      -1.2E-1

Embedded white (blank) space is not allowed, in any numeric constant.

Exponential notation is useful for representing numbers that are either very larger or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to -3.68E-7.

A floating point constant may thus comprise four parts:

- a whole number
- a decimal point
- a fractional part
- an exponent

### Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Examples of character constants are:

'5'      'X'      ';'      "

Note that the character constant '5' is not the same as the number 5. The last constant is a blank space.

### String Constants

A string constant is a sequence of characters enclosed between double quotes. the characters may be alphabets, digits, special characters and blank spaces. Examples are:

"Hello Java"      "1997"      "WELL DONE"      "?...!"      "5+3"      "X"

### Backslash Character Constants

Java supports some special backslash character constants that are used in output methods. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 4.1. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known *escape sequences*.

**Table 4.1 BKlclluh ChanIct:w ConatantI**

Constant'	MeanIn,
'\b'	back space
,\f'	form feed
'\n'	new line
,\r'	carriage return
'\t'	horizontal tab
'\' ,	sineJe quote
,\""	double quote
'\\'	backslash



## VARIABLES

A ~ is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. In Chapter 3, we had used several variables. For instance, we used variables length and breadth to store the values of length and breadth of a room.

A variable name can be chosen by the programmer in a meaningful way so as to reflect what it represents in the program. Some examples of variable names are:

- average
- height
- total\_height
- classStrength

As mentioned earlier, variable names may consist of alphabets, digits, the underscore( ) and dollar characters, subject to the following conditions:

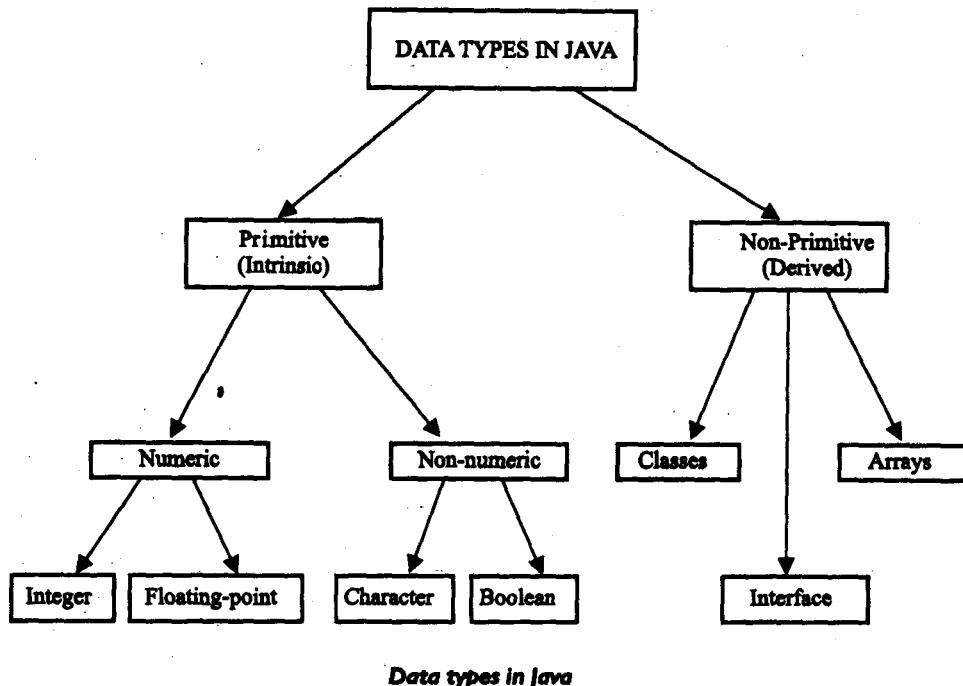
1. They must not begin with a digit.
2. Uppercase and lowercase are distinct. This means that the variable Total is not the same as total or TOTAL.
3. It should not be a keyword.
4. White space is not allowed.
5. Variable names can be of any length.



## DATA TYPES

Every variable in Java has a data type. Data types specify the size and type of values that can be stored. Java language is rich in its *data types*. The variety of data types available allow the

programmer to select the type appropriate to the needs of the application. Data types fall under various categories as shown in Fig. 4.2. Primitive types (also called intrinsic or built-in types) are discussed in detail in this Chapter. Derived types (also known as reference types) are discussed later as and when they are encountered.

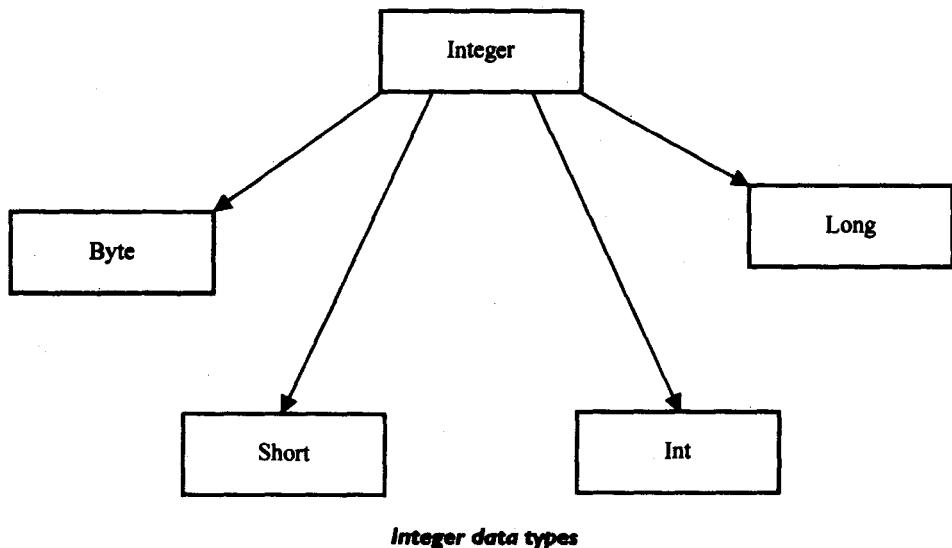
**Fig. 4.2**

### Integer Types

Integer types can hold whole numbers such as 123, -96, and 5639. The size of the values that can be stored depends on the integer data type we choose. Java supports four types of integers as shown in Fig. 4.3. They are byte, short, int, and long. Java does not support the concept of unsigned types and therefore all Java values are signed meaning they can be positive or negative. Table 4.3 shows the memory size and range of all the four integer data types.

**Table 4.3 Size and Range of Integer Types**

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

**Fig. 4.3**

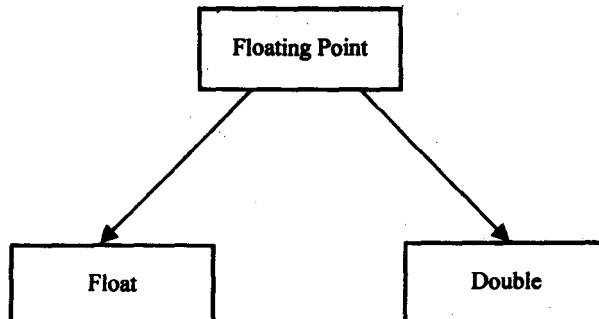
It should be remembered that wider data types require more time for manipulation and therefore it is advisable to use smaller data types, wherever possible. For example, instead of storing a number like 50 in an Int type variable, we must use a byte variable to handle this number. This will improve the speed of execution of the program.

We can make integers **long** by appending the letter L or l at the end of the number. Example:

123L      or      123l

### Floating Point Types

Integer types can hold only whole numbers and therefore we use another type known as floating point type to hold numbers containing fractional parts such as 27.59 and -1.375 (known as floating point constants). There are two kinds of floating point storage in Java as shown in FigAA.

**Fig. 4.4**

**Floating point data types**

The float type values are *single-precision* numbers while the double types represent *double-precision* numbers. Table 4.4 gives the size and range of these two types.

Table 4.4 Size and Range of Floating Point Types

Type	Size	Minimum value	Maximum value
float	4 bytes	3.4e-038	3.4e+038
double	8 bytes	1.7e-308	1.7e+308

Floating point numbers are treated as double-precision quantities. To force them to be in single-precision 'mode, we must append f or F to the numbers. Example:

```
1.23f
7.56923eSF
```

Double-precision types are used when we need greater precision in storage of floating point numbers. All mathematical functions, such as sin, cos and sqrt return double type values.

Floating point data types support a special value known as Not-a-Number (NaN). NaN is used to represent the result of operations such as dividing zero by zero, where an actual number is not produced. Most operations that have NaN as an operand will produce NaN as a result.

### Character Type

In order to store character constants in memory, Java provides a character data type called char. The char type assumes a size of 2 bytes but, basically, it can hold only a single character.

### Boolean Type

Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a boolean type can take: true or false. Remember, both these words have been declared as keywords. Boolean type is denoted by the keyword boolean and uses only one bit of storage.

All comparison operators (see Chapter 5) return boolean type values. Boolean values are often used in selection and iteration statements.



## DECLARATION OF VARIABLES

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. The place of declaration (in the program) decides the scope of the variable.

A variable must be declared before it is used in the program.

A variable can be used to store a value of any data type. That is, the name has nothing to do with the type. Java allows any properly formed variable to have any declared data type. The declaration statement defines the type of variable. The general form of declaration of a variable is:

```
type variable1, variable2, ......., variableN;
```

Variables are separated by commas. A declaration statement must end with a semicolon. Some valid declarations are:

```
int      count;
float    x, y;
double   pi;
byte     b;
char    c1, c2, c3;
```

## GIVING VALUES TO VARIABLES

A variable must be given a value after it has been declared but before it is used in an expression. This can be achieved in two ways:

1. By using an assignment statement
2. By using a read statement

### Assignment Statement

A simple method of giving value to a variable is through the assignment statement as follows:

```
variableName = value;
```

For example:

```
initialValue = 0;
finalValue  = 100;
yes        = 'x';
```

We can also string assignment expressions as shown below:

```
x = y = z = 0;
```

It is also possible to assign a value to a variable at the time of its declaration. This takes the form:

```
type variableName = value;
```

Examples:

```

int      finalValue   =      100;
char     yes          =      'X';
double   total        =      75.36;

```

The process of giving initial values to variables is known as the *initialization*. The ones that are not initialized are automatically set to zero.

The following are valid Java statements:

```

float x, y, z;           // declares three float variables
int m = 5, n = 10;       // declares and initializes two int variables
int m, n = 10;           // declares m and n and initializes n

```

## Read Statement

We may also give values to variables interactively through the keyboard using the `readLine()` method as illustrated in Program 4.1.

### Program 4.1 Reading data from keyboard

```

import java.io.DataInputStream;

class Reading
{
    public static void main (String args[])
    {
        DataInputStream in = new DataInputStream(System.in);
        int intNumber = 0;
        float floatNumber = 0.0f;

        try
        {
            System.out.println("Enter an Integer: ");
            intNumber = Integer.parseInt(in.readLine());
            System.out.println("Enter a float number: ");
            floatNumber =
                Float.valueOf(in.readLine()).floatValue();
        }

        catch (Exception e) { }

        System.out.println ("intNumber = " + intNumber);
        System.out.println("floatNumber = " + floatNumber);
    }
}

```

The interactive input and output of Program 4.1 are shown below:

```
Enter an integer:  
123  
Enter a float number:  
123.45  
intNumber = 123  
floatNumber = 123.45
```

The `readLine()` method (which is invoked using an object of the class `DataInputStream`) reads the input from the keyboard as a string which is then converted to the corresponding data type using the data type wrapper classes. See Chapter 9 for more about wrapper classes.

Note that we have used the `try` and `catch` to handle any errors that might occur during the reading process. Java requires this. See Chapter 13 for more details on error handling.



## SCOPE OF VARIABLES

Java variables are actually classified into three kinds:

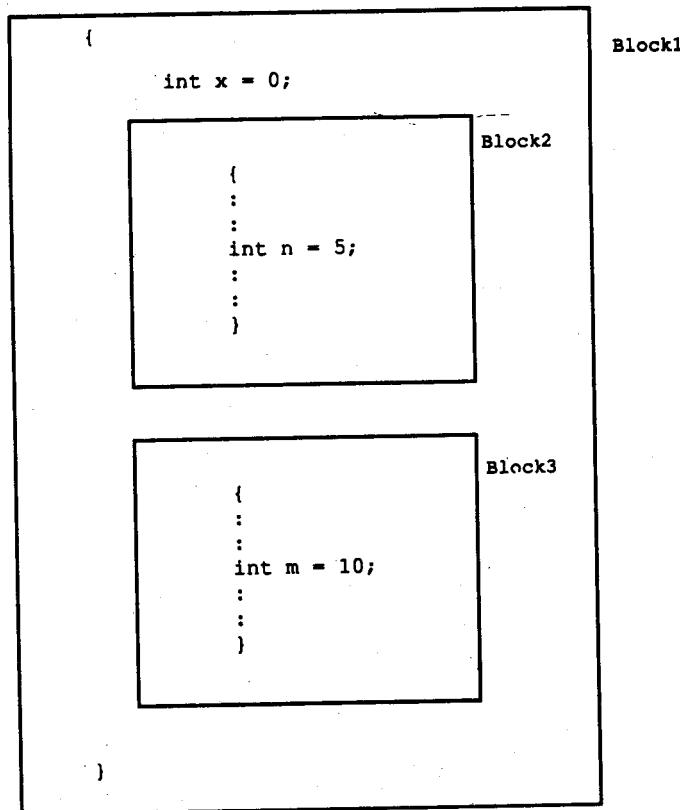
- *instance* variables,
- *class* variables, and
- *local* variables.

Instance and class variables are declared inside a class. Instance variables are created when the objects are instantiated and therefore they are associated with the objects. They take different *values* for each object. On the other hand, class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable. Instance and class variables will be considered in detail in Chapter 8.

Variables declared and used inside methods are called *local variables*. They are called so because they are not available for use outside the method definition. Local variables can also be declared inside program blocks that are defined between an opening brace `{` and a closing brace `}`. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will cease to exist. The area of the program where the variable is accessible (i.e., usable) is called its *scope*.

We can have program blocks within other program blocks (called nesting) as shown in Fig. 4.5. Each block can contain its own set of local variable declarations. We cannot, however, declare a variable to have the same name as one in an outer block. In Fig. 4.5, the variable `x` declared in Block1 is available in all the three blocks. However, the variable `n` declared in Block2 is available only in Block2, because it goes out of the scope at the end of Block2. Similarly, `m` is accessible only in BlockJ.

Note that we cannot declare the variable `x` again in Block2 or BlockJ (This is perfectly legal in C and C++).

**Nested program blocks**

## **4.8**

### **SYMBOLIC CONSTANTS**

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant "pi". Another example is the total number of students whose mark-sheets are analysed in a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs. They are:

1. Problem in modification of the program.
2. Problem in understanding the program.

## Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

## Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of a symbolic name to such constants frees us from these problems. For example, we may use the name STRENGTH to denote the number of students and PASS\_MARK to denote the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names STRENGTH and PASS\_MARK in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is declared as follows:

```
| final      type symbolic-name = value; |
```

Valid examples of constant declaration are:

```
final int      STRENGTH    = 100;
final int      PASS_MARK   = 50;
final float    PI          = 3.14159;
```

Note that:

1. Symbolic names take the same form as variable names. But, they are written in CAPITALS to visually distinguish them from normal variable names. This is only a convention, not a rule.
2. After declaration of symbolic constants, they should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
3. Symbolic constants are declared for types. This is not done in C and C++ where symbolic constants are defined using the #define statement.
4. They can NOT be declared inside a method. They should be used only as class data members in the beginning of the class.



## TYPE CASTING

We often encounter situations where there is a need to store a value of one type into a variable of another type. In such situations, we must cast the value to be stored by preceding it with the type name in parentheses. The syntax is:

```
| type variable1 = (type) variable2; |
```

The process of converting one data type to another is called *casting*.

Examples:

```
int m = 50;
byte n = (byte)m;
long count = (long)m;
```

Casting is often necessary when a method returns a type different than the one we require.

Four integer types can be cast to any other type except boolean. Casting into a smaller type may result in a loss of data. Similarly, the float and double can be cast to any other type except boolean. Again, casting to smaller type can result in a loss of data. Casting a floating point value to an integer will result in a loss of the fractional part. Table 4.5 lists those casts, which are guaranteed to result in no loss of information.

Table 4.5 Casts That Result In No Loss of Information

From	To
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

## Automatic Conversion

For some types, it is possible to assign a value of one type to a variable of a different type without a cast. Java does the conversion of the assigned value automatically. This is known as *automatic type conversion*. Automatic type conversion is possible only if the destination type has enough precision to store the source value. For example, int is large enough to hold a byte value. Therefore,

```
byte b = 75;
int a = b;
```

are valid statements.

The process of assigning a smaller type to a larger one is known as *widening* or *promotion* and that of assigning a larger type to a smaller one is known as *narrowing*. Note that narrowing may result in loss of information.

Program 4.2 illustrates the creation of variables of basic types and also shows the effect of type conversions.

**Program 4.2 Creation and casting of variables**

```
class TypeWrap
{
    public static void main (String args[])
    {
        System.out.println("Variables-created");
        char c = .•x";
        byte b = 50;
        shorts = 1996;
        int i = 123456789;
        long l = 1234567654321L;
        float f1 = 3.142F;
        floa~ f2 = 1.2e-SF;
        double d2 = 0.000000987;

        System.out.println("      c = " + c);
        System.out.println("      b = " + b);
        System.out.println("      s = " + s);
        System.out.println("      i = " + i);
        System.out.println("      l="+"1");
        System.out.println("      f1 = " + f1);
        System.out.println("      f2 = " + f2);
        System.out.println("      d2 = " + d2);

        System.out.println("      ");
        System.out.println("Types converted");
        short sl = (short)b;
        short s2 = (short)i; // Produces incorrect result
        float n1 = (float)l;
        int m1 = (int)f1; // Fractional part is lost

        System.out.println("      (short)b = " + sl);
        System.out.println("      (short)i = " + s2);
        System.out.println("      (float)l = " + n1);
        System.out.println("      (int)f1 = " + m1);

    }
}
```

Output of Program 4.2 is as follows:

```
Variables created
c = x
b = 50
s = 1996
i = 123456789
l = 1234567654321
f1 = 3.142
f2 = 1.2e-005
92 = 9.87e-007

Types converted
(short)b = 50
(short)i = -13035
(float)l = 1.23457e+012
(int)f1 = 3
```

Note that floating point constants have a default type of double. What happens when we want to declare a float variable and initialize it using a constant? Example:

```
float x = 7.56;
```

This will cause the following compiler error:

```
Incompatible type for declaration. Explicit cast needed to
convert double to, float."
```

This should be written as:

```
float x = 7.56F;
```

## GETTING VALUES OF VARIABLES

A computer program is written to manipulate a given set of data and to display or print the results. Java supports two output methods that can be used to send the results to the screen.

- `print( . )` method // print and wait
- `println( )` method // print a line and move to next line

The `print()` method sends information into a buffer. This buffer is not flushed until a newline (or end-of-line) character is sent. As a result, the `print()` method prints output on one line until a newline character is encountered. For example, the statements

```
System.out.print("Hello      H");
System.out.print("Java! ");
```

will display the words Hello Java! on one line and waits for displaying further information on the same line. We may force the display to be brought to the next line by printing a newline character as shown below:

```
System.out.print('\n');
```

For example, the statements

```
System.out.print("Hello");
System.out.print("\n");
System.out.print("Java!");
```

will display the output in two lines as follows:

```
Hello
Java!
```

The `println()` method, by contrast, takes the information provided and displays it on a line followed by a line feed (carriage-return). This means that the statements

```
System.out.println("Hello");
System.out.println("Java!");
```

will produce the following output:

```
Hello
Java!
```

The statement

```
System.out.println();
```

will print a blank line. Program 4.3 illustrates the behaviour of `print()` and `println()` methods ..

### **Program 4.3 Getting the result to the screen**

---

```
class Displaying
{
    public static void main(String args[])
    {
        System.out.println("Screen      Display");
        for(int i = 1; i <= 9; i++)
        {
            for(int j = 1, j <= i; j++)
            {
                System.out.print(~      " ");
                System.out.print(i);
            }
            System.out.print("\n");
        }
        System.out.println("Screen      Display Done");
    }
}
```

Program 4.3 displays the following on the screen:

Screen	Display	
	<b>1</b>	
2	2	
3	3 3	
4	4 4 4	
5	5 5 5 5	
6	6 6 6 6 6	
7	7 7 7 7 7 7	
8	8 8 8 8 8 8 8	
9	9 9 9 9 9 9 9 9	
Screen	Display	Done

## STANDARD DEFAULT VALUES

In Java, every variable has a default value. If we don't initialize a variable when it is first created, Java provides default value to that variable type automatically as shown in Table 4.6.

**Table 4.6 Default Values for Various Types**

Type of variable	Default value
byte	Zero : (byte) 0
Short	Zero: (short) 0
int	Zero: 0
long	Zero: OL
float	0.Of
double	0.Od
char	null character
boolean	false
reference	null

## SUMMARY

This chapter has provided us with a brief description of Java constants and variables and how they are represented inside the computer. We have also seen how the variables are declared and initialized in Java.

Converting one type of data to another is often necessary during implementation of a program. We have discussed how data type conversion is achieved in Java without loss of accuracy.

All programs must 'read, manipulate and display data. We discussed briefly how values are assigned to variables and how the results are displayed on the screen. These concepts will be applied for developing larger programs in the forthcoming chapters ..

**KEY TERMS**

**Data, Information, Syntax, Constants, Variables, Integer, Decimal, Octal, Hexadecimal, Real constants, Floating point constants, Character constants, Backslash characters, Reference types, Boolean, NaN, Initialization, Scope, Instance variables, Class variables, Local variables, Nesting, Casting, Widening, Narrowing.**

**REVIEW QUESTIONS**

- 4.1 What is a constant?
- 4.2 What is a variable?
- 4.3 How are constants and variables important in developing programs?
- 4.4 List the eight basic data types used in Java. Give examples.
- 4.5 What is scope of a variable?
- 4.6 What is type casting? Why is it required in programming?
- 4.7 What is initialization? Why is it important?
- 4.8 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations?
- 4.9 What are symbolic constants? How are they useful in developing programs?
- 4.10 Which of the following are invalid constants and why?
 

0.0001	5 • 1.5	RS 75.50
+100	75.45E-2	"15.75"
-45.6	-1.45e( +4)	0.000001234
- 4.11 Which of the following are invalid variable names and why?
 

Minimum	first.Name	nl+n2
doubles	3rd-row	N\$
float	Sum Total	Total-Marks
- 4.12 Find errors, if any, in the following declaration statements:
 

```
Intx;
float length, HEIGHT;
double = p,q;
character Cl;
final int TOTAL;
final pi = 3.142;
long int m;
```
- 4.13 Write a program to determine the sum of the following harmonic series for a given value of n:

$$1 + 1/2 + 1/3 + \dots + 1/n$$

The value of n should be given interactively through the keyboard.

- 4.14 Write a program to read the price of an item in decimal form (like 75.95) and print the output in paise (like 7595 paise).

- 4.15 Write a program to convert the given temperature in fahrenheit to celsius using the following conversion formula

$$C = \frac{F - 32}{1.8}$$

and display the values in a tabular form.

---

## **Chapter 5**

# **Operators and Expressions**

**5.1**

## **INTRODUCTION**

Java supports a rich set of operators. We have already used several of them, such as `=`, `+`, `-` and `*`. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions.

Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators.
7. Bitwise operators
8. Special operators

**5.2**

## **ARITHMETIC OPERATORS**

Java provides all the basic arithmetic operators. They are listed in Table 5.1. The operators `+`, `-`, `*`, and `/` all work the same way as they do in other languages. These can operate on any built-in numeric data type of Java. We cannot use these operators on boolean type. The unary minus operator, in effect, multiplies its single operand by `-1`. Therefore, a number preceded by a minus sign changes its sign.

**Table 5.1 Arithmedc Operaton**

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulodivision

Arithmetic operators are used as shown below:

$$\begin{array}{ll} a-b & a+b \\ a*b & a/b \\ a\%b & -a^* b \end{array}$$

Here a and b may be variables or constants and are known as *operands*.

## Integer Arithmetic

When both the operands in a single arithmetic expression such as  $a+b$  are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. In the above examples, if a and b are integers, then for  $a = 14$  and  $b = 4$  we have the following results:

$$\begin{array}{ll} a-b & = 10 \\ a+b & = 18 \\ a*b & = 56 \\ a/b & = 3 \text{ (decimal part truncated)} \\ a\%b & = 2 \text{ (remainder of integer division)} \end{array}$$

$a\%b$ , when a and b are integer types, gives the result of division of a by b after truncating the divisor. This operation is called the *integer division*.

For modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

$$\begin{array}{ll} -14 \% 3 & = -2 \\ -14\%-3 & = -2 \\ 14 \% - 3 & = 2 \end{array}$$

(Note that module division is defined as:  $a\%b = a - (a/b)*b$ , where  $a/b$  is the integer division).

## Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

Unlike C and C++, modulus operator % can be applied to the floating point data as well. The floating point modulus operator returns the floating point equivalent of an integer division. What this means is that the division is carried out with both floating point operands, but the resulting divisor is treated as an integer, resulting in a floating point remainder. Program 5.1 shows how arithmetic operators work on floating point values.

### **Program 5.1 Floating point arithmetic**

---

```
class FloatPoint
{
    public static void main (String args[])
    {
        float a = 20.5F, b = 6.4F;
        System.out.println(~a = " + a);
        System.out.println(~b = " + b);
        System.out.println(~a+b = " + (a+b));
        System.out.println(~a-b = " + (a-b));
        System.out.println(~a*b = " + (a*b));
        System.out.println(~a/b = " + (a/b»);
        System.out.println(~a%b = " + (a%b));
    }
}
```

---

The output of Program 5.1 is follows:

```
a = 20.5
b = 6.4
a+b = 26.9
a-b = 14.1
a*b = 131.2
a/b = 3.20313
a%b = 1.3
```

### **Mixed-mode Arithmetic**

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression. If either operand is of the real type, then the other operand is converted to real and the real arithmetic is performed. The result will be a real. Thus

15/10.0 produces the result 1.5

whereas

15/10 produces the result 1

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

## 5.3

**RELATIONAL OPERATORS**

We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '`<`' , meaning 'less than'. An expression such as

`a < b` or `x < 20`

containing a relational operator is termed as a *relational expression*. The value of relational expression is either true or false. For example, if `x = 10`, then

`x < 20` is true

while

`20 < x` is false.

Java supports six relational operators in all. These operators and their meanings are shown in Table 5.2,

**Table 5.2 Relational Operator**

Operator	Meaning
<code>&lt;</code>	is less than
<code>&lt;=</code>	is less than or equal to
<code>&gt;</code>	is greater than
<code>&gt;=</code>	is greater than or equal to
<code>==</code>	is equal to
<code>!=</code>	is not equal to

A simple relational expression contains only one relational operator and is of the following form:

| ae-1 relational operator ae-2 |

`ae-1` and `ae-2` are arithmetic expressions, which may be simple constants, variables or combination of them. Table 5.3 shows some examples of simple relational expressions and their values.

**Table 5.3 Relational Expressions**

Expression	Value
<code>4.5 &lt;= 10</code>	TRUE
<code>4.5 &lt; -10</code>	FALSE
<code>-J5 &gt;= 0</code>	FALSE
<code>10 &lt; 7+5</code>	TRUE
<code>a + b == c+d</code>	TRUE .

- Only if the sum of values of `a` and `b` is equal to the sum of values of `c` and `d`.

- **Har**''' Jtwe: A Prim8r

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators. Program 5.2 shows the implementation of relational operators.

## Program 5.2 Implementation of relational operators

---

```
class RelationalOperators
{
    public static void main (String args[])
    {
        float a = 15.0F, b = 20.75F, c = 15.0F;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("a < b is " + (a < b));
        System.out.println("a > b is " + (a > b));
        System.out.println("a == c is " + (a == c));
        System.out.println("a <= c is " + (a <= c));
        System.out.println("a >= b is " + (a >= b));
        System.out.println("b != c is " + (b != c));
        System.out.println("b == a+c is " + (b == a+c));
    }
}
```

---

The output of Program 5.2 would be:

```
a = 15
b = 20.75
c = 15
a < b is true
a > b is false
a == c is true
a <= c is true
a >= b is false
b != c is true
b == a+c is false
```

Relational expressions are used in *decision statements* such as, if and while to decide the course of action of a running program. Decision statements are discussed in detail in Chapters 6 and 7.



## LOGICAL OPERATORS

In addition to the relational operators, Java has three logical operators, which are given in Table 5.4.

Table 5.4 LoskaI Operators

Operator	Meanin~
&~	loioal AND
	logical OR
!	loioal NOT

The logical operators `&&` and `||` are used when we want to form compound conditions by combining two or more relations. An example is:

`a > b && x == 10`

An expression of this kind which combines two or more relational expressions is termed as a *logical expression* or a *comporu:relatiOnal expn?ssion*. Like the simple relational expressions, a logical expression also yields a value of true or false, according to the *truth table* shown in Table S.S.-The logical expression given above is true only if both `a>b` and `x == 10` are true. If either (or both) of them are false, the expression is false.

Table 5.5 Truth Table

		Value of the expression			
op-1	op-2	op-1 && op-2	op-1    op-2	op-3	op-2
true	true	true	true	true	true
true	false	false	true	true	false
false	true	false	true	false	true
false	false	false	false	false	false

Note:

- `op-1 && op-2` is true if both `op-1` and `op-2` are true and false otherwise .
- `op-1 || op-2` is false if both `op-1` and `op-2` are false and true otherwise.

Some examples of the usage of logical expressions are:

1. `if (age>S5 && salary<1000)`
2. `if (number<0 || number>100)`



## ASSIGNMENT OPERATORS

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator, `=`. In addition, Java has a set of 'shorthand' assignment operators which are used in the form

|      v op- exp;      |

where `v` is a variable, `exp` is an expression and `op` is a Java binary operator. The operator `op =` is known as the shorthand assignment operator.

The assignment statement

`v op= exp;`

is equivalent to

`v = v op (exp);`

with `v` accessed only once. Consider an example

`x += y+1;`

This is same as the statement

`x = x+(y+1);`

The shorthand operator `+=` means 'add `y+1` to `x`' or 'increment `x` by `y+1`'. For `y = 2`, the above statement becomes

`x.+= 3;`

and when this statement is executed, 3 is added to `x`. if the old value of `x` is, say 5, then the new value of `x` is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 5.6.

**Table 5.6 Shorthand Assignment Operators**

Statement with simple assignment operator	Statement with shorthand operator
<code>a = a+1</code>	<code>a += 1</code>
<code>a = a*1</code>	<code>a -= 1</code>
<code>a = a*(n+1)</code>	<code>a *= n+1</code>
<code>a = a/ (n+1)</code>	<code>a /= n+1</code>
<code>a = a%b</code>	<code>a %= b</code>

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. Use of shorthand operator results in a more efficient code.



## 5.6 INCREMENT AND DECREMENT OPERATORS

Java has two very useful operators not generally found in many other languages. These are the increment and decrement operators:

`++` and `--`

The operator `++` adds 1 to the operand while `--` subtracts 1. Both are unary operators and are used in the following form:

```
++m;    or    m++;
--m;    or    m--;
++m;    is equivalent to m = m + 1; (or m += 1);
--m;    is equivalent to m = m - 1; (or m -= 1);
```

We use the increment and decrement operators extensively in for and while loops. (See Chapter 7.)

While `++m` and `m++` mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of `y` and `m` would be 6. Suppose, if we rewrite the above statement as

```
m = 5;
y = m++;
```

then, the value of `y` would be 5 and `m` would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand. Program 5.3 illustrates this.

---

### Program 5.3 Increment operator Illustrated

```
class IncrementOperator
{
    public static void main(String args[])
    {
        int m = 10, n = 20;
        System.out.println("      m = " + m);
        System.out.println("      n = " + n);
        System.out.println("      ++m = " + ++m);
        System.out.println("      n++ = " + n++);
        System.out.println("      m = " + m);
        System.out.println("      n = " + n);
    }
}
```

## 7. Programming with Java: A Primer

Output of Program 5.3 is as follows:

```
m = 10  
n = 20  
++m = 11  
n++ = 20  
m = 11  
n = 21
```

Similar is the case, when we use `++` (or `--`) in subscripted variables. That is, the statement

```
a[i++] = 10
```

is equivalent to

```
a[i] = 10  
i = i+1
```

## 5.7 CONDITIONAL OPERATOR

The character pair `?:` is a ternary operator available in Java. This operator is used to construct conditional expressions of the form

```
| exp1 ? exp2 : exp3 |
```

where `exp1`, `exp2`, and `exp3` are expressions.

The operator `?:` works as follows: `exp1` is evaluated first. If it is nonzero (true), then the expression `exp2` is evaluated and becomes the value of the conditional expression. If `exp1` is false, `exp3` is evaluated and its value becomes the value of the conditional expression. Note that only one of the expressions (either `exp2` or `exp3`) is evaluated. For example, consider the following statements:

```
a = 10;  
b = 15;  
x = (a > b) ? a : b;
```

In this example, `x` will be assigned the value of `b`. This can be achieved using the if..else statement as follows:

```
if(a > b)  
    x = a;  
else  
    x = b;
```

## 5.8 BITWISE OPERATORS

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or

shifting them to the right or left. Bitwise operators may not be applied to float or double. Table 5.7 lists the bitwise operators. They are discussed in detail in Appendix D.

Table 5.7 Bitwise Operators

Operator	Meaning		
&	bitwise	AND	
!	bitwise	OR	
^	bitwise	exclusive OR	
-	one's complement		
«	shift left		
»	shift right		
»>	shift right with zero fill		

## SPECIAL OPERATORS

Java supports some special operators of interest such as instanceof operator and member selection operator (.)�

### instanceof Operator

The instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.

Example:

```
person instanceof student
```

is true if the object person belongs to the class student; otherwise it is false.

### Dot Operator

The dot operator (.) is used to access the instance variables and methods of class objects. Examples:

```
personl.age      / / Reference to the variable age
personl.salary() / / Reference to the method salary()
```

It is also used to access classes and sub-packages from a package.

## ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. Java can handle any complex mathematical expressions. Some of the examples

of Java expressions are shown in Table 5.8. Remember that Java does not have an operator for exponentiation.

Table 5.8 Expressions

Algebraic expression	Java expression
$ab-c$	$a*b-c$
$(m+n)(x+y)$	$(m+n)*(x+y)$
$\frac{ab}{c}$	$a*b/c$
$Jx^2+2x+1$	$J*x*x+2*x+1$
$\frac{x+c}{y}$	$x/y+c$



## EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form

$| \quad variable \quad = \quad expression; \quad |$

'Variable' is any valid Java variable name. When the statement is encountered, the *expression* is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evalution statements are

```
x = a*b-c;
y = b/c*a;
z = a-b/c+d;
```

The blank space around an operator is optional and is added only to improve readability. When these statements are used in program, the variables a,b,c and d must be defined before they are used in the expressions.



## PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without any parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in Java:

<i>High priori ty</i>	* / %
<i>Low priori ty</i>	+ -

The basic evaluation procedure includes two left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered.

During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement:

$$x = a-b/3+c^2-1$$

When  $a = 9$ ,  $b = 12$ , and  $c = 3$ , the statement becomes

$$x = 9-12/3+3^2-1$$

and is evaluated as follows:

*FIrst pass*

$$\text{Step1: } x = 9-4+3^2-1 \quad (12/3 \text{ evaluated})$$

$$\text{Step2: } x = 9-4+6-1 \quad (3^2 \text{ evaluated})$$

*Second pass*

$$\text{Step3: } x = 5+6-1 \quad (9-4 \text{ evaluated})$$

$$\text{Step4: } x = 11-1 \quad (5+6 \text{ evaluated})$$

$$\text{Step5: } x = 10 \quad (1;1-1 \text{ evaluated})$$

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

$$9-12/(3+3)*(2-1)$$

Whenever the parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

*r;irst pass*

$$\text{Step1: } 9-12/6*(2-1).$$

$$\text{Step2: } 9-12/6*1$$

*eond pass*

$$\text{Step3: } 9-2*1$$

$$\text{Step4: } 9-2$$

*rthird pass*

$$\text{Step5: } 7$$

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remain the same as 5 (i.e., equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing one. For example

$$9-(12/(3+3)*2)-1 = 4$$

whereas

$$9-((12/3)+3*2)-1 = -2$$

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

**5.13****TYPE CONVERSIONS IN EXPRESSIONS****Automatic Type Conversion**

Java permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. We know that the computer, considers one operator at a time, involving two operands. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type.

If byte, short and int variables are used in an expression, the result is always promoted to int, to avoid overflow. If a single long is used in the expression, the whole expression is promoted to long. Remember that all integer values are considered to be int unless they have the I or L appended to them. If an expression contains a float operand, the entire expression is promoted to float. If any operand is double, result is double. Table 5.9 provides a reference chart for type conversion.

Table 5.9 Automatic Type Conversion Chart

	char	byte	short	int	long	float	double
char	int	int	int	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	double						
double							

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits.
3. long to int causes dropping of the excess higher order bits.

## Casting a Value

We have already discussed how Java performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

```
ratio = female_number/male_number
```

Since `female_number` and `male_number` are declared as integers in the program, the decimal part of the result of the division would be lost and `ratio` would not represent a correct figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

```
ratio = (float)female_number/male_number
```

The operator `(float)` converts the `female_number` to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator `(float)` affect the value of the variable `female_number`. And also, the type of `female_number` remains as `int` in the other parts of the program.

The process of such a local conversion is known as *casting a Value*. The general form of a cast is:

$$\boxed{(\text{type-name}) \quad \text{expression}}$$

where `type-name` is one of the standard data types. The `expression` may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 5.10.

Table 5.10 Use of Casts

Examples	Action
<code>x = (int) 7.5</code>	7.5 is converted to integer by truncation
<code>a = (int)21.3/(int)4.5</code>	Evaluated as $21/4$ and the result would be 5
<code>b = (double)sum/n</code>	Division is done in floating point mode.
<code>y = (int) (a+b)</code>	The result of $a + b$ is converted to integer.
<code>z = (int) a+b</code>	$a$ is converted to integer and then added to $b$ .
<code>p = cost&lt;double&gt;x)</code>	Converts $x$ to double before using it as parameter.

Casting can be used to round-off a given value to an integer. Consider the following statement:

- Programming with Java: A Primer

```
x = (int) (y+0.5);
```

If y is 27.6, y + 0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to x. Of course, the expression being cast is not changed.

When combining two different types of variables in an expression, never assume the rules of automatic conversion. It is always a good practice to explicitly force the conversion. It is more safer. For example, when y and p are double and m is int, the following two statements are equivalent.

```
y = p+m;
y = p+(double)m;
```

However, the second statement is preferable.

Program 5.4 illustrates the use of casting in evaluating the equation

$$\text{sum} = \sum_{i=1}^n \frac{1}{i}$$

---

#### **ProgramS.4 illustration of the use of casting operation**

---

```
class Casting
{
    public static void main (String args[])
    {
        float sum;
        int i;
        sum = 0.0F;
        for(i = 1; i <= 10; i++)
        {
            sum = sum + 1/(float)i;
            System.out.print("      i = " + i);
            System.out.println("      sum = " + sum);
        }
    }
}
```

---

Program 5.3 produces the following output:

```
i = 1  sum = 1
i = 2  sum = 1.5
i = 3  sum = 1.83333
i = 4  sum = 2.08333
i = 5  sum = 2.28333
i = 6  sum = 2.45
i = 7  sum = 2.59286
i = 8  sum = 2.71786
i = 9  sum = 2.82897
i = 10 sum = 2.92897
```

3.14

## OPERATOR PRECEDENCE AND ASSOCIATIVITY

Each operator in Java has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the associativity property of an operator. Table 5.11 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence (rank 1 indicates the highest precedence level and 14 the lowest). The list also includes those operators which we have not yet discussed.

Table 5.11 Summary of Java Operators

Operator	Description	Associativity	Rank
.	Member selection	Left to right	1
()	Function call		
[]	Array element reference		
-	Unary minus	Right to left	2
++.	Increment		
--	Decrement		
!	Logical negation		
-	Ones complement		
(type)	Casting		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
>>>	Right shift with zero fill		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
instanceof	Type comparison		
==	Equality	Left to right	7
!=	Inequality		

(Continued)

- [Fh.oar•tJg rrlh J8vs: A Primer](#)

**Table 5.11 (Continued)**

Operator	Description			Associativity			Rank
&	Bitwise	AND		Left	to	right	8
“	Bitwise	XOR		Left	to	right	9
	Bitwise	OR		Left	to	right	10
&&	Logical	AND		Left	to	right	11
	Logical	OR		Left	to	right	12
?:	Conditional	operator		Right	to	left	13
=	Assignment	operators		Right	to	left	14
op=	Shorthand	assignment					

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

```
if (x = 10+15 && y<10)
```

The precedence rules say that the addition operator has a higher priority than the logical operator (**&&**) and the relational operators{== and <}. Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

```
if (x == 25 && y<10)
```

The next step is to determine whether x is equal to 25 and y is less than 10. If we assume a value of 20 for x and. 5 for y, then

```
x == 25 is FALSE
y < 10 is TRUE
```

Note that since the operator < enjoys a higher priority compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally we get:

```
if(FALSE && TRUE)
```

Because one of the conditions is FALSE, the compound condition is FALSE.



## MATHEMATICAL FUNCTIONS

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Java supports these basic math functions through Math class defined in the java.lang package. Table 5.12 lists the math functions defined in the Math class. These functions should be used as follows:

```
Math. function_name ( )
```

Example:

```
double y = Math.sqrt(x);
```

**Table 5.12 Math Functions**

Functions	Action
sin(x)	Returns the sine of the angle x in radians
cos(x)	Returns the cosine of the angle x in radians
tan (x)	Returns the tangent of the angle x in radians
asin(y)	Returns the angle whose sine is y
acos(y)	Returns the angle whose cosine is y
atan(y)	Returns the angle whose tangent is y
atan2(x,y)	Returns the angle whose tangent is x/y
pow(x,y)	Returns x raised to y (xr)
exp(x)	Returns e raised to x (ex)
log (x)	Returns the natural logarithm of x
sqrt(x)	Returns the square root of x
ceil (x)	Returns the smallest whole number greater than or equal to x. (Rounding up)
iloor(x)	Returns the largest whole number less than or equal to x (Rounded down)
rint(x)	Returns the truncated value of x.
abs(a)	Returns the absolute value of a
max(a,b)	Returns the maximum of a and b
min(a,b)	Returns the minimum of a and b

Note: *x* and *y* are double type parameters. *a* and *b* may be ints, *s*, floats and doubles.

## 5.16 SUMMARY

We have discussed all the basic data types and operators available in Java and also seen their use in expressions. Type conversions and order of precedence of operators during the evaluation of expressions have been highlighted. Program 5.5 winds up our discussions by demonstrating the use of different types of expressions.

Finally, it is important to note that all Java types have fixed sizes. There is no ambiguity and all Java types are machine-independent.

**Program 5.5 Demonstration of Java expressions**

```

class ExpressWrap
{
    public static void main (String args[])
    {
        II Declaration and Initialization
        int a = 10, b ~ 5, c = 8, d = 2;
        float x = 6.4F, y = 3.0F;

        1.1 Order of Evaluation
        int answer1 = a * b +c / d;
        int answer2 = a * (b + c) /d;

        II Type Conversions
        float answer3 = a / c;
        float answer4 = (float)a / c;
        float answerS = a I y;

        II Modulo Operations
        int answer6 = a % c;
        float answer7 = x % y;

        I/ Logical Operations
        boolean bool1 = a > b && c > d;
        boolean bool2 = a < b && c > d;
        boolean bool3 = a < b || c > d;
        boolean bool4 = !(a-b == c);

        System.out.println("Order of Evaluation");
        System.out.println("      a * b + c I d = " + answer1);
        System.out.println("      a * (b + c) I d= " + answer2);

        System.out.println("TypeConversions");
        System.out.println("      a / c = " + answer3);
        System.out.println("      (float)a / c = " + answer4);
        System.out.println("      a / y = " + answerS);

        System.out.println("Modulo Operations");
        System.out.println("      a % c = " + answer6);
        System.out.println("      x % y = "+ answer7);
    }
}

```

(Continued)

*Program 5.5 (Continued)*

---

```
System.out.println("LogicalOperations");
System.out.println("      a > b && c > d = " + bool1);
System.out.println("      a < b && c > d = " + bool2);
System.out.println("      a < b || c > d = " + bool3);
System.out.println("      !(a-b == c) = " + bool4);
}
}
```

---

Program 5.5 outputs the following:

```
Order of Evaluation
a * b + c / d = 54
a * (b + c) / d = 65
Type Conversions
a / c = 1
(float)a / c = 1.25
a / y = 3.33333
Modulo Operations
a % c = 2
x % y = 0.4
Logical Operations
a > b && c > d = true
a < b && c > d = false
a < b || c > d = true
!(a-b == c) = true
```

## KEY TERMS

**Operands, Integer arithmetic, Real arithmetic, Mixed-mode arithmetic, Relational expression, Logical expression, Truth table, Ternary operator, Conditional operator, Increment operator, Decrement operator, Dot operator, instanceof operator, Casting, Operator precedence, Associativity**

## REVIEW QUESTIONS

---

- 5.1 Which of the following arithmetic expressions are valid?
- |                        |                             |
|------------------------|-----------------------------|
| (a) $25/3 \% 2$        | (e) $-14 \% 3$              |
| (b) $+9/4 + 5$         | (f) $15.25 + -5.0$          |
| (c) $7.5 \% 3$         | (g) $(5/3) * 3 + 5 \% 3$    |
| (d) $14 \% 3 + 7 \% 2$ | (h) $21 \% (\text{int})4.5$ |

5.2 Write Java assignment statements to evaluate the following equations:

(a) Area  $= \pi r^2 + 2\pi rh$

(b) Torque  $= \frac{2m_1 m_2}{m_1 + m_2} * g$

(c) Side  $= \sqrt{a^2 + b^2 - 2ab \cos(x)}$

(d) Energy  $= \text{mass} \left( \text{acceleration} * \text{height} + \frac{\text{velocity}^2}{2} \right)$

5.3 Identify unnecessary parentheses in the following arithmetic expressions.

(a)  $(x - (y/5) + z) \% 8) + 25$

(b)  $((x-y) * p) + q$

(c)  $(m*n) + (-x/y)$

(d)  $x/(3*y)$

5.4 Find errors, if any, in the following assignment statements and rectify them.

(a)  $x = y = z = 0.5, 2.0 - 5.75;$

(b)  $m = ++a * 5;$

(c)  $y = \text{sqrt}(100);$

(d)  $p^* = x/y;$

(e)  $s = /5;$

(f)  $a = b++ - c * 2$

5.5 Determine the value of each of the following logical expressions if  $a = 5$ ,  $b = 10$  and  $c = -6$

(a)  $a > b \&\& a < c$

(b)  $a < b \&\& a > c$

(c)  $a == c \mid\mid b > a$

(d)  $b > 15 \&\& c < 0 \mid\mid a > 0$

(e)  $(a/2.0 == 0.0 \&\& b/2.0 != 0.0) \mid\mid c < 0.0$

5.6 The straight-line method of computing the early depreciation of the value of an item is given by

$$\text{Depreciation} = \frac{\text{Purchase price} - \text{Salvage value}}{\text{Years of service}}$$

Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given.

5.7 Write a program that will read a real number from the keyboard and print the following output in one line:

Smallest integer  
not less than  
than the number

The given number

Largest integer  
not greater than  
the number

- 5.8 The total distance travelled by a vehicle in  $t$  seconds is given by

$$\text{distance} = ut + (at^2)/2$$

where  $u$  is the initial velocity (metres per second),  $a$  is the acceleration (metres per second $^2$ ). Write a program to evaluate the distance travelled at regular intervals of time, given the values of  $u$  and  $a$ . The program should provide the flexibility to the user to select his own time intervals and repeat the calculations for different values of  $u$  and  $a$ .

- 5.9 In inventory management, the Economic Order Quantity for a single item is given by

$$\text{EOQ} = \sqrt{\frac{2 * \text{demand rate} * \text{setup costs}}{\text{holding cost per item per unit time}}}$$

and the optimal Time Between Orders

$$\text{TBO} = \sqrt{\frac{2 * \text{setup costs}}{\text{demand rate} * \text{holding cost per item per unit time}}}$$

Write a program to computer EOQ and TBO, given demand rate (items per unit time), setup costs (per order), and the holding cost (per item per unit time).

- 5.10 For a certain electrical circuit with an inductance  $L$  and resistance  $R$ , the damped natural frequency is given by

$$\text{Frequency} = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$$

It is desired to study the variation of this frequency with  $C$  (capacitance). Write a program to calculate the frequency for different values of  $C$  starting from 0.01 to 0.1 in steps of 0.01.

---

## **Chapter 6**

# **Decision Making and**

### **INTRODUCTION**

A Java program is a set of statements, which are normally executed sequentially in the order in which they appear. This happens when options or repetitions of certain calculations are not necessary. However, in practice, we have a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

When a program breaks the sequential flow and jumps to another part of the code, it is called *branching*. When the branching is based on a particular condition, it is known as *conditional branching*. If branching takes place without any decision, it is known as *unconditional branching*.

Java language possesses such decision making capabilities and supports the following statements known as *control* or *decision making* statements to implement branching.

1. if statement
2. switch statement
3. Conditional operator statement

In this Chapter, we shall discuss the features, capabilities and applications of these statements which are also classified as *selection* statements.

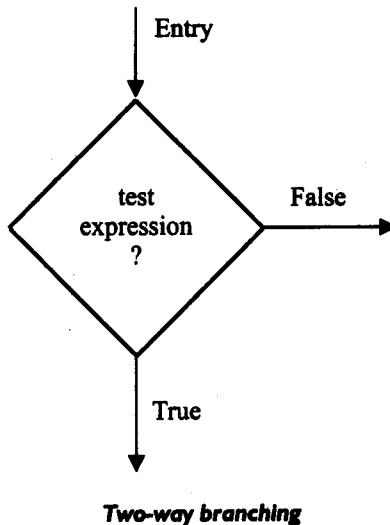
### **DECISION MAKING WITH IF STATEMENT**

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a *two-way* decision statement and is used in conjunction with an expression. It takes the following form:

```
if (test expression)
```

It allows the computer to evaluate the *expression* first and then, depending on whether the value of the *expression* (relation or condition) is 'true' or 'false', it transfers the control to a particular statement. This point of program has two paths to follow, one for the *true* condition and the other for the *false* condition as shown in Fig. 6.1.

**Fig. 6.1**



Some examples of decision making, using if statement are:

1. if (bank balance is zero)  
borrow money
2. if (room is dark)  
put on lights
3. if (code is 1)  
person is male
4. if (age is more than 55)  
person is retired

The if statement may be implemented in different forms depending on the complexity of conditions to be tested.

1. Simple **if** statement
2. **if .. else** statement
3. Nested **if .. else** statement
4. **else if** ladder



6.3

## SIMPLE IF STATEMENT

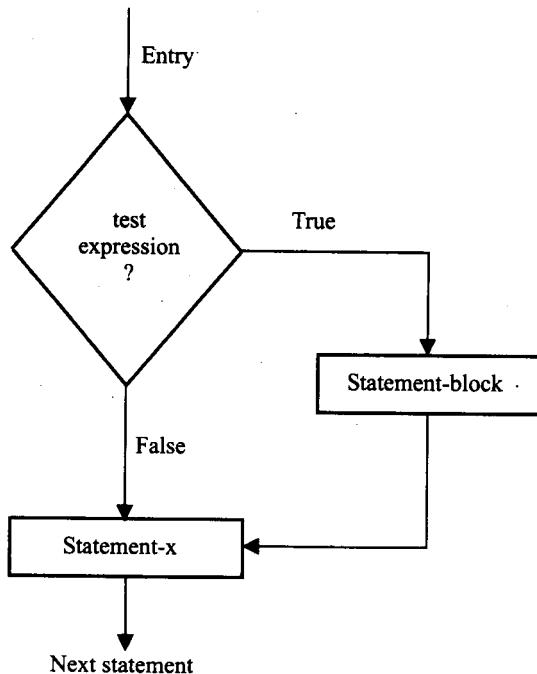
The general form of a simple if statement is

```
if(test expression)
{
    statement-block;
}
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the *statement-block* will be skipped and the execution will jump to the *statement-x*.

It should be remembered that when the condition is true both the *statement-block* and the *statement-x* are executed in sequence. This is illustrated in Fig. 6.2.

**Fig. 6.2**



**Flowchart of simple if control**

Consider the following segment ("") of a program that is written for processing of marks obtained in an entrance examination.

```

-----
if (category == SPORTS)
{
    marks = marks + bonus_marks;
}
System.out.println(marks);
-----
.
```

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus\_marks are added to his marks before they are printed. For others, bonus\_marks are not added.

Consider a case having two test conditions, one for weight and another for height. This is done using the compound relation

```
if(weight < 50 && height> 170) count = count +1;
```

This would have been equivalently done using two if statements as follows:

```
if (weight<50)
    if(height>170)
        count = count+1;
```

If the value of weight is less than 50, then the following statement is executed, which in turn is another if statement. This if statement tests height and if the height is greater than 170, then the count is incremented by 1. Program 6.1 illustrates the implementation of the above concept.

### ***Program 6.1 Counting with if statement***

---

```
class IfTest
{
    public static void main (String args[])
    {
        int i, count, count1, count2 ;-
        float [] weight = { 45.0F,55.0F,47.0F,51.0F,54.0F      } ;
        float [] height = { 176.5F,174.2F,168.0F,170.7F,169.0F      } ;
        count = 0;
        count1 = 0;
        count2 = 0;

        for (i = 0; i <= 4; i++)
        {
```

---

(Continued)

### Program 6.1 (Continued)

---

```
if(weight[i] < 50.0 && height[i] > 170.0)
{
    count1 = count1 + 1;
}
count = count + 1; // Total persons

}
count2 = count - count1;

System.out.println("Number      of persons with . . .");
System.out.println("Weight<50      and height>170 = "+count1);
System.out.println("Others      = " + count2);
}
```

---

The output of Program 6.1 will be:

```
Number of persons with . .
Weight<50 and height>170 ~ 1
Others = 4
```



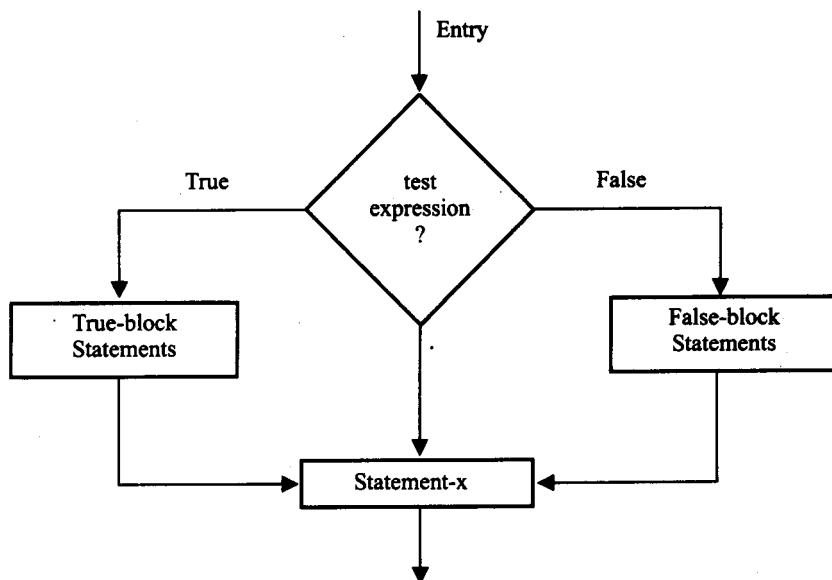
## THE IF.. ELSE STATEMENT

The if...else statement is an extension of the simple if statement. The general form is

```
if (test expression)
{
    True-block statement(s);
}
else
{
    False-block statement(s);
}
statement-x;
```

If the *test expression* is true, then the *true-block statement(s)* immediately following the if statement, are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. This is illustrated in Fig. 6.3. In both the cases, the control is transferred subsequently to the *statement-x*.

Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The if statements to do this may be written as follows:

**Fig. 6.3****Flowchart of if...else control**

```

.....
if (code == 1)
    boy = boy + 1;
if (code == 2)
    girl = girl + 1;
.....
.....

```

The first test determines whether or not the student is a boy. If yes, the number of boys is increased by 1 and the program continues to the second test. The second test again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both. The above program segment can be modified using the **else** clause as follows:

```

.....
.....
if (code == 1)
    boy = boy + 1;
else
    girl = girl + 1;
xxx;
.....

```

Here, if the code is equal to 1, the statement `boy = boy + 1;` is executed and the control is transferred to the statement `xxx,` after skipping the `else` part. If the code is not equal to 1, the statement `boy = boy + 1;` is skipped and the statement in the `else` part `girl = girl + 1;` is executed before the control reaches the statement `xxx.`

Program 6.2 counts the even and odd numbers in a list of numbers using the `if...else` statement. `number[]` is an array variable containing all the numbers and `number.length` gives the number of elements in the array.

### **Program 6.2   Experimenting with if...e/se statement**

---

```
class IfElseTest
{
    public static void main (String args[])
    {
        int number [] = { 50, 65, 56, 71, 81 };
        int even = 0, odd = 0;

        for (int i = 0; i < number.length; i++)
        {
            if ((number[i] % 2) == 0) // Decide even or odd
            {
                even += 1; // counting EVENnumbers
            }
            else
            {
                odd += 1; // counting ODDnumbers
            }
        }
        System.out.println("Even      Numbers : " + even +
                           "      Odd Numbers : " + odd);
    }
}
```

---

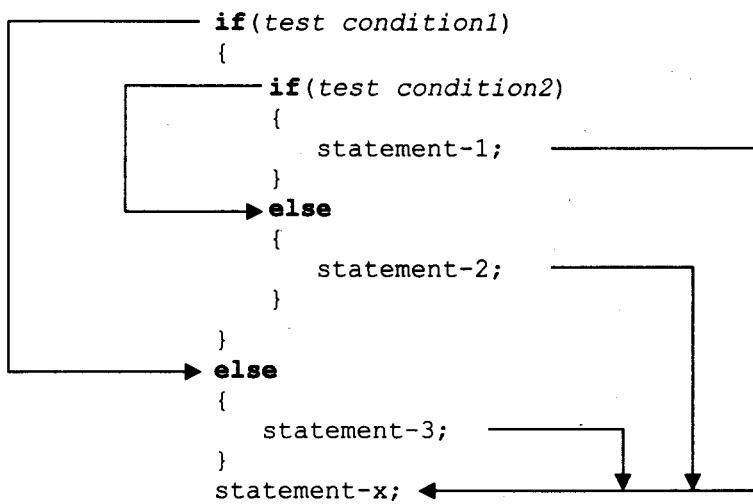
Output of Program 6.2:

Even Numbers : 2    Odd Numbers : 3



### **.NESTIN~ OF IF..•ELSE STATEMENTS**

When a series of decisions are involved, we may have to use more than one `if...else` statement in *nested* form as follows:



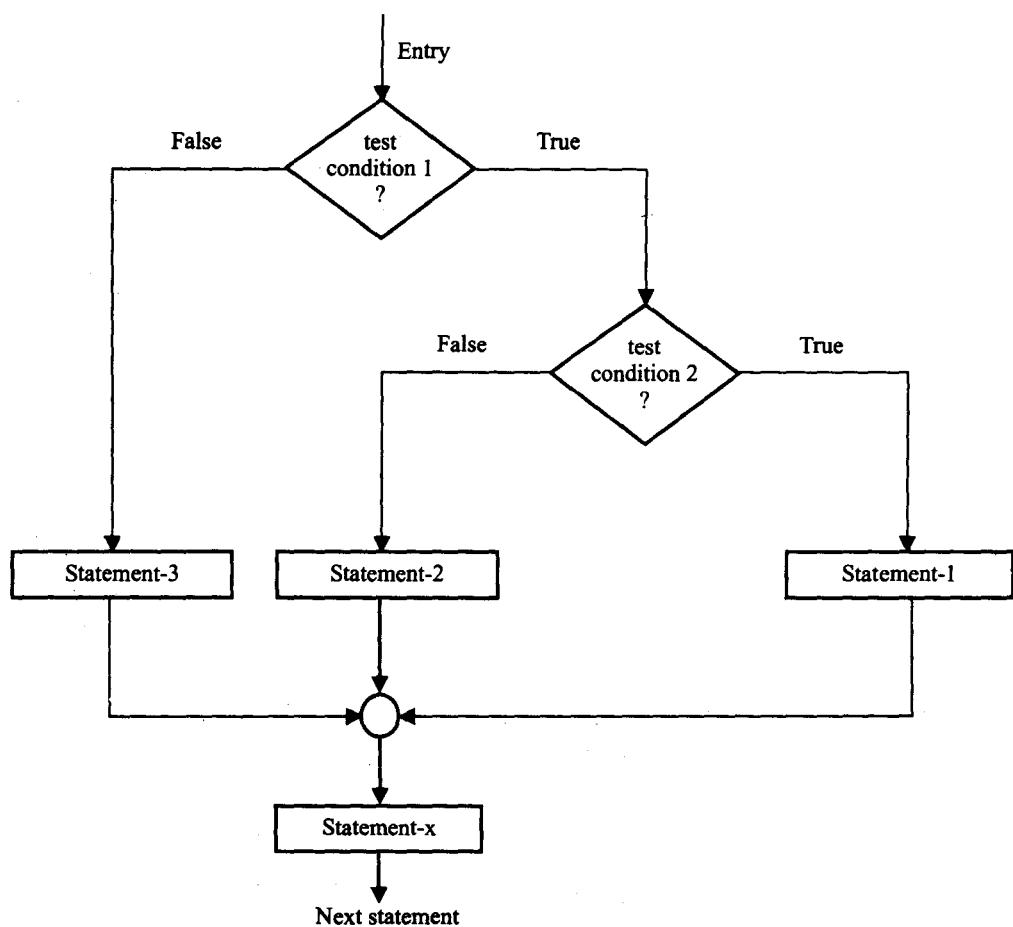
The logic of execution is illustrated in Fig. 6.4. If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the *balance* held on 31st December is given to every one, irrespective of their balances, and 5 per cent is given to female account holders if their balance is more than RS5000. This logic can be coded as follows:

```

.....
if (sex is female)
{
    if (balance > 5000)
        bonus = 0.05 * balante;
    else
        bonus = 0.02 * balance;
}
else
{
    bonus = 0.02 * balance;
}
balance = balance + bonus;
.....
.....

```

Fig. 6.4

Flowchart of nested if...else statements

When nesting, care should be exercised to match every if with an else. Consider the following alternative to the above program (which looks right at the first sight):

```

if (sex is female)
    if (balance > 5000)
        bonus = 0.05 * balance;
else
    bonus = 0.02 * balance;
balance = balance + bonus;
  
```

There is an ambiguity as to over which if the else belongs to. In Java an else is linked to the closest non-terminated if. Therefore, the else is associated with the inner if and there is no else option for the outer if. This means that the computer is trying to execute the statement

```
balance = balance + bonus;
```

without really calculating the bonus for the male account holders.

Consider another alternative, which also looks correct:

```
if (sex is female)
{
    if (balance > 5000)
        bonus = 0.05 * balance;
}
else
    bonus = 0.02 * balance;
balance = balance + bonus;
```

In this case, else is associated with the outer if and therefore bonus is calculated for the male account holders. However, bonus for the female account holders, whose balance is equal to or less than 5000 is not calculated because of the missing else option for the inner if.

Program 6.3 employs nested if...else statements to determine the largest of three given numbers.

### **Program 6.3 Nesting, if...else statements**

---

```
class IfElseNesting
{
    public static void main (String args[])
    {
        int a = 325, b = 712, c = 478;
        System.out.print("Largest value is : ");
        if (a > b)
        {
            if (a > c)
            {
                System.out.println(a);
            }
            else
            {
                System.out.println(c);
            }
        }
        else
        {
```

---

(Continued)

## Program 6.3 (Continued)

```

if (c > b)
{
    System.out.println(c);
}
else
{
    System.out.println(b);
}
}

```

F

## Output of Program 6.3:

Largest value is : 712



## THE ELSE IF LADDER

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if. It takes the following general form:

```

if(condition1)
    statement-1;

else if(condition 2)
    statement-2;

else if(condition 3)
    statement-3;
    .....
else if (condition n)
    statement-n;

else
    default-statement;

statement-x;

```

This construct is known as the else if ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as the true condition is found, the statement associated with it is executed and the control is transferred to the *statement-x* (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the *default-statement* will be executed. Fig. 6.5 shows the logic of execution of else if ladder statements.

Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Average marks	Grade
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

This grading can be done using the else if ladder as follows:

```

if (marks > 79)
    grade = "Honours";
else if (marks > 59)
    grade = "First Division";

else if (marks > 49)
    grade = "Second Division";

else if (marks > 39)
    grade = "Third Division";

else
    grade = "Fail";

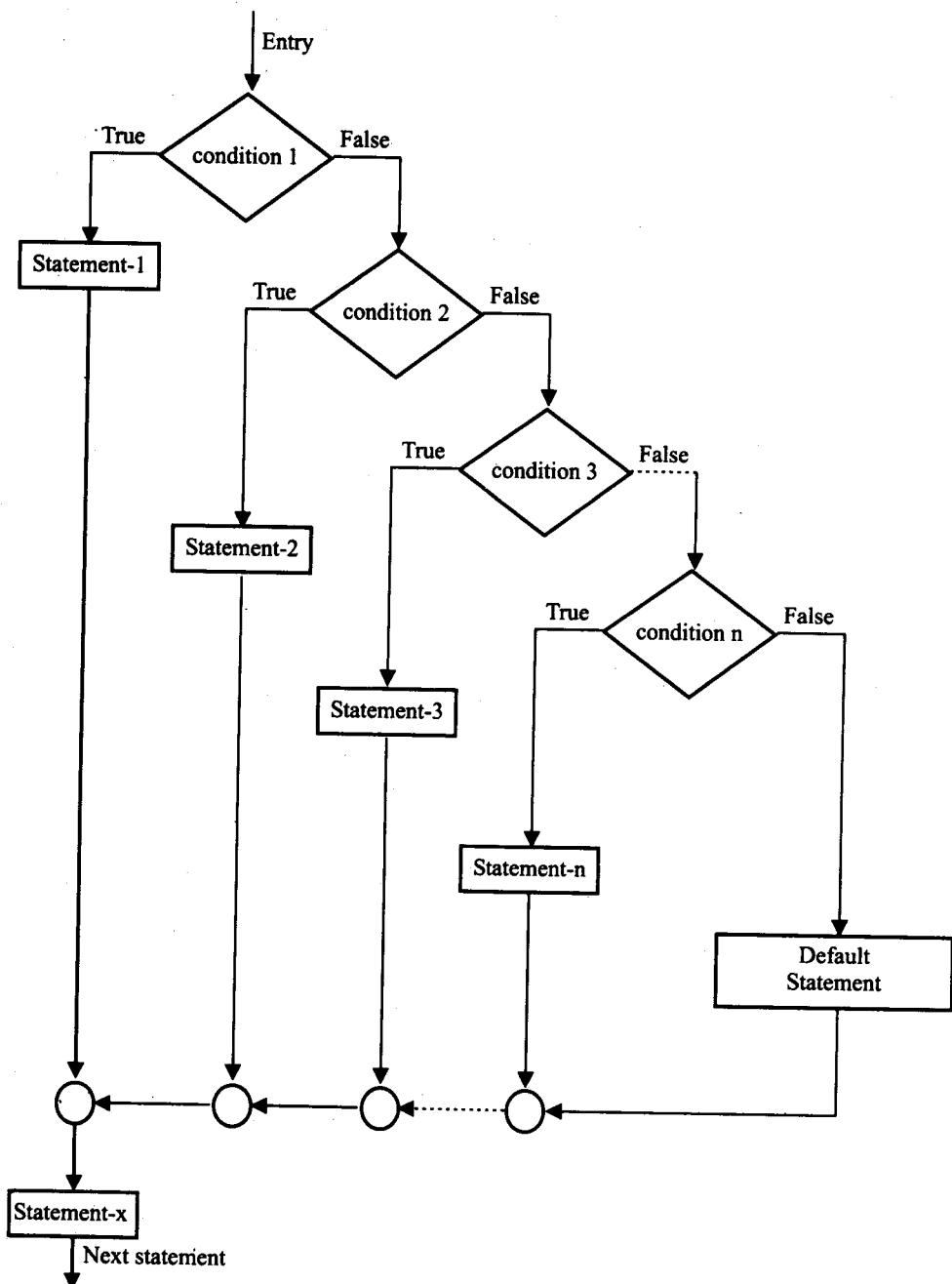
System.out.println("Grade: " + grade);

```

Consider another example given below:

```

-----
-----
if (code == 1)
    colour = "RED";
else if (code == 2)
    colour = "GREEN";
else if (code == 3)
    colour = "WHITE";
else
    colour = "YELLOW";
-----
-----
```

**Fig. 6.5****Flowchart of else if ladder**

Code numbers other than 1,2 or 3 are considered to represent YELWW colour. The same results can be obtained by using nested if..else statements.

```
if (code != 1)
    if (code != 2)
        if (code != 3)
            colour = "YELLOW";
    .1..
    colour = "WHITE";
.1..
colour = "GREEN";
.1..
colour = "RED";
```

In such situations, the choice of the method is left to the programmer. However, in order to choose an if structure that is both effective and efficient, it is important that the programmer is fully aware of the various forms of an if statement and the rules governing their nesting.

Program 6.4 demonstrates the use of if .. else ladder in analysing a mark list.

#### **Program 6.4 Demonstration of else if ladder**

---

```
class ElseIfLadder
{
    public static void main(String args[])
    {
        int rollNumber[] = { 111, 222, 333, 444 };
        int marks[] = { 81, 75, 43, 58 };
        for (int i = 0; i < rollNumber.length; i++)
        {
            if (marks[i] > 79)
                System.out.println(rollNumber[i] + " Honours");
            else if (marks[i] > 59)
                System.out.println(rollNumber[i] + " I Division");
            else if (marks[i] > 49)
                System.out.println(rollNumber[i] +
                    " II Division");

            else
                System.out.println(rollNumber[i] + " FAIL");
        }
    }
}
```

---

Program 6.4 produces the following output:

```

111 Honours
222 I Division
333 FAIL
444 II Division

```

## THE SWITCH STATEMENT

We have seen that when one of the many alternatives is to be selected, we can design a program using if statements to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the designer of the program. Fortunately, Java has a built-in multiway decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:

```

switch (expression)
{
    case value-1:
        block-1
        break;
    case value-2:
        block-2
        break;
    .....
    .....
    default:
        default-block
        break;
}
statement-x;

```

The *expression* is an integer expression or characters. 'Value-1, 'Value-2 .... are constants or constant expressions (evaluable to an integral constant) and are known as *case labels*. Each of these values should be unique within a switch statement. *block-1, block-2....* are statement lists and may contain zero or more statements. There is no need to put braces around these blocks but it is important to note that case labels end with a colon (:).

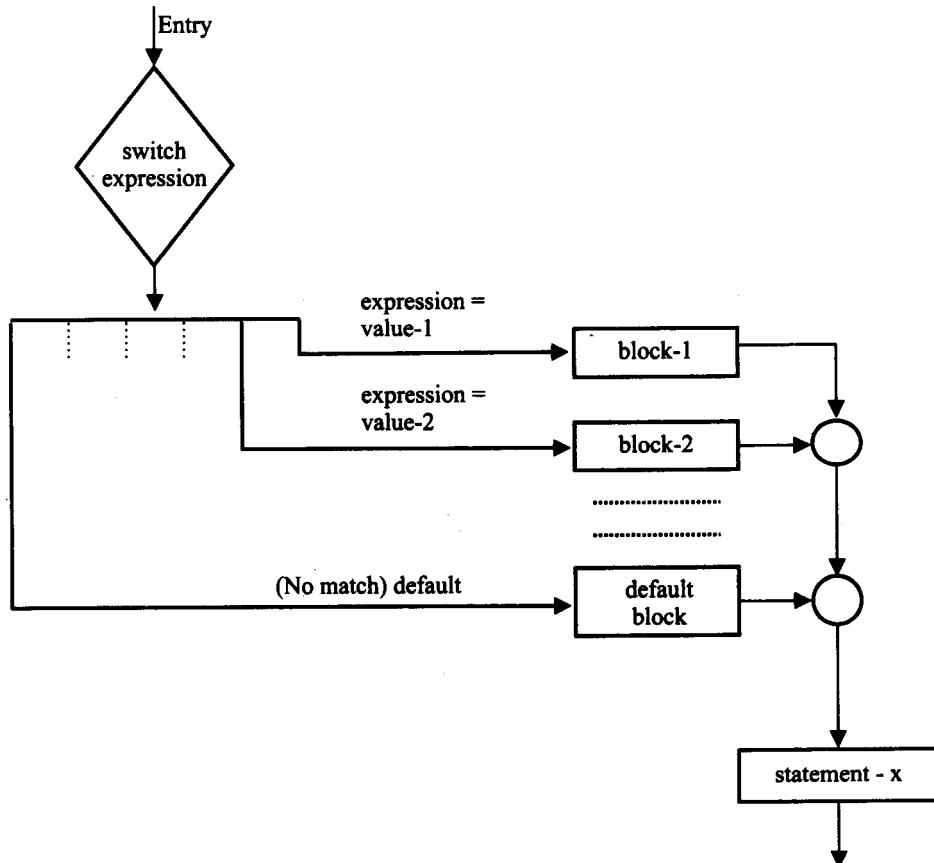
When the switch is executed, the value of the expression is successively compared against the values 'Value-1, 'Value-2, .... If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement; transferring the control to the *statement-x* following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place when all matches fail and the control goes to the *statement-x*.

The selection process of switch statement is illustrated in the flowchart shown in Fig. 6.6.

**Fig. 6.6**



Selection process of the switch statement

The switch statement can be used to grade the students as discussed in the last section. This is illustrated below:

```

-----
-----
index = marks/10i
8"itch(index)
{
    case 10:
    case 9:
  
```

```

case 8:
    grade = "Honours";
    break;
case 7:
case 6:
    grade = "First Division";
    break;
case 5:
    grade = "Second Division";
    break;
case 4:
    grade = "Third Division";
    break;
default:
    grade = "Fail";
    break;
}

System.out.println(grade);
.....
.....

```

Note that we have used a conversion statement

```
index = marks/10;
```

where, index is defined as an integer. The variable index takes the following integer values.

Marks	Index
100	10
90 - 99	9
80 - 89	8
70 - 79	7
60 - 69	6
50 - 59	5
40 - 49	4
30 - 39'	3
20 - 29	2
10 - 19	1
0 - 9	0

The segment of the program illustrates two important features. First, it uses empty cases. The first three cases will execute the same statements

```
grade = "Honours";
break;
```

Same is the case with case 7 and case 6. Second, default condition is used for all other cases where marks is less than 40.

**Program 6.5 illustrates the use of switch for designing a menudriven interactive program.**

### **Program 6.5 Testing the switch()**

---

```
class CityGuide
{
    public static void main (String args[])
    {
        char choice;
        System.out.println("Select      your choice");
        System.out.println("      M ---> Madras");
        System.out.println("      B ---> Bombay");
        System.out.println("      C ---> Calcutta");
        System.out.print("Choice      ----->");
        System.out.flush();

        try
        {
            switch (choice = (char)System.in.read())
            {
                case 'M':
                case 'm': System.out.println("Madras      : Booklet  5");
                            break;
                case 'B':
                case 'b': System.out.println("Bombay     : Booklet  9");
                            break;
                case 'C':
                case 'c': System.out.println("Cal-cutta: Booklet15");
                            break;
                default  : System.out.println("Invalid      Choice  (IC)");
            }
        }
        catch (Exception e)
        {
            System.out.println("I/O      Error");
        }
    }
}
```

---

**Output of the Program 6.5:**

```
Run1
Select your choice
M ---> Madras
B ---> Bombay
C ---> Calcutta
```

```

Choice----->      m
Madras   : Booklet  5

Run2
Select  your choice
M ...--> Madras
B ----> Bombay
C ---> Calcutta
Choice  ----->  M
Madras   : Booklet  5

Run3
Select  your choice
M --->  Madras
B --->  Bombay
C --->  Calcutta
Choice  ----->c
Calcutta   : .Booklet15

```



## THE 1: OPERATOR

The Java language has an unusual operator, useful for making two-way decisions. This operator is a combination of? and :, and takes three operands. This operator is popularly known as the conditional operator. The general form of use of the *conditional operator* is as follows:

`conditional expression ? expression1 : expression2`

The *conditional expression* is evaluated first. If the result is true, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```

if  (x < 0)
    flag  = 0;
else
    flag  = 1;

```

can be written as

```
flag  = (x<0) ? 0 : 1;
```

Consider the evaluation of the following function:

```

y = 1.5x + 3      for  x <= 2
Y = 2x + 5        for  x  > 2

```

This can be evaluated using the-conditional operator as follows:

```
y = (x>2) ? (2*x+5) : (1.5*x+3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If  $x$  is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as

```
salary = (x!=40)? ((x<40)?(4*x+100):(4.5*x+150)): 300;
```

The same can be evaluated using if...else statements as follows:

```
if (x<=40)
    if (x<40)
        salary = 4*x+100;
    else
        salary = 300;
else
    salary = 4.5*x+150;
```

When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use if statements when more than a single nesting of conditional operator is required.

## 6.9 SUMMARY

We have discussed in this chapter the features of the following selection statements supported by Java:

- if statement
- switch statement
- ?: operator statement

We have seen the various forms of application of these statements and discussed how they can be used to solve real-life problems. Control execution is an extremely important tool in programming. The concepts discussed here will be certainly useful in developing complex systems.

### KEY TERMS

**Decision Making, Branching, Control, Conditional Branching, Ladder Selection, Switch, Conditional Operator.**

## REVIEW QUESTIONS

---

- 6.1 . Determine whether the following are true or false;
- When if statements are nested, the last else gets associated with the nearest if without an else.
  - One if can have more than one else clause
  - A switch statement can always be replaced by a series of if••else statements.
  - A switch expression can be of any type.
  - A program stops its execution when a break statement is encountered.
- 6.2 In what ways does a switch statement differ from an if statement?
- 6.3 Find errors, if any, in each of the following segments:
- `if(x+y = z && y>0)`
  - `if (code>1) ;  
    a = b+c  
    else  
        a = 0`
  - `if(p<0) || (q<0)`
- 6.4 The following is a segment of a program:
- ```
x = 1;  
y = 1;  
if (n>0)  
    x = x+1;  
    y = y-1;
```
- What will be the values of x and y if n assumes a value of (a) 1 and (b)0.
- 6.5 Rewrite each of the following without using compound relations:
- `if (grade<=59 && grade>=50)  
    second = second + 1`
  - `if (number>100 || number<0)  
    System.out.print("Out of range");  
else  
    sum = sum + number;`
  - `if ((M1>60 && M2>60 || T>200)  
    • y=1;  
else  
    y";'0;`
- 6.6 Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.
- 6.7 A set of two linear equations with two unknowns  $x_1$  and  $x_2$  is given below:

$$\begin{aligned} ax_1 + bx_2 &= m \\ cx_1 + dx_2 &= n \end{aligned}$$

The set has a unique solution

$$x_1 = \frac{md - bn}{ad - cb}$$

$$x_2 = \frac{na - mc}{ad - cb}$$

Provided the denominator  $ad - cd$  is not equal to zero.

Write a program that will read the values of constants  $a, b, c, d, m$  and  $n$  and compute the values of  $x_1$  and  $x_2$ . An appropriate message should be printed if  $ad - cb = 0$ .

- 6.8 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students who have obtained marks

- a) in the range 81 to 100,
- b) in the range 61 to 80,
- c) in the range 41 to 60, and
- d) in the range 0 to 40.

The program should use a minimum number of if statements.

- 6.9 Admission to a professional course is subject to the following conditions:

- a) Marks in mathematics  $\geq 60$
- b) Marks in physics  $\geq 50$
- c) Marks in chemistry  $\geq 40$
- d) Total in all three subjects  $\geq 200$   
(or)

Total in mathematics and physics  $\geq 150$

Given the marks in the three subjects, write a program to process the applications to list the eligible candidates.

- 6.10 Write a program to print a two-dimensional Square Root Table as shown below, to provide the square root of any number from 0 to 9.9. For example, the value  $x$  will give the square root of 3.2 and  $y$  the square root of 3.9.

Square Root Table

| Number | 0.0 | 0.1 | 0.2 | ..... | 0.9 |
|--------|-----|-----|-----|-------|-----|
| 0.0    |     |     |     |       |     |
| 1.0    |     |     |     |       |     |
| 2.0    |     |     |     |       |     |
| 3.0    |     |     | x   |       | y   |
|        |     |     |     |       |     |
| 9.0    |     |     |     |       |     |

6.11 Shown below is a Floyd's triangle.

```
1  
23  
456  
7 8 9 10  
11 ..... 15
```

```
79 ..... 91
```

- (a) Write a program to print this triangle.  
(b) Modify the program to produce the following form of Floyd's triangle.

```
1  
01  
101  
0101  
10101
```

6.12 A cloth showroom has announced the following seasonal discounts on purchase of items:

| Purchase amount | Discount   |                |
|-----------------|------------|----------------|
|                 | Mill cloth | Handloom items |
| 0 - 100         | -          | 5.0%           |
| 101- 200        | 5.0%       | 7.5%           |
| 201- 300        | 7.5%       | 10.0%          |
| Above 300       | 10.0%      | 15.0%          |

Write a program using switch and if statements to compute the net amount to be paid by a customer.

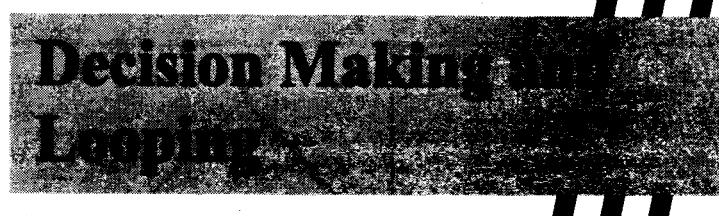
6.13 Write a program that will read the value of x and evaluate the following function

$$y = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$$

using

- (a) nested if statements,  
(b) else if statements, and  
(c) conditional operator?:

# **Chapter 7**



## **INTRODUCTION**

A computer is well suited to perform repetitive operations. It can do it tirelessly for 10,100 or even 10,000 times. Every computer language must have features that instruct a computer to perform such repetitive tasks. The process of repeatedly executing a block of statements is known as *looping*. The statements in the block may be executed any number of times, from zero to *infinite* number. If a loop continues forever, it is called an *infinite* loop.

Java supports such looping features which enable us to develop concise programs containing repetitive processes without using unconditional branching statements like goto statement. Remember, Java does not define goto statement.

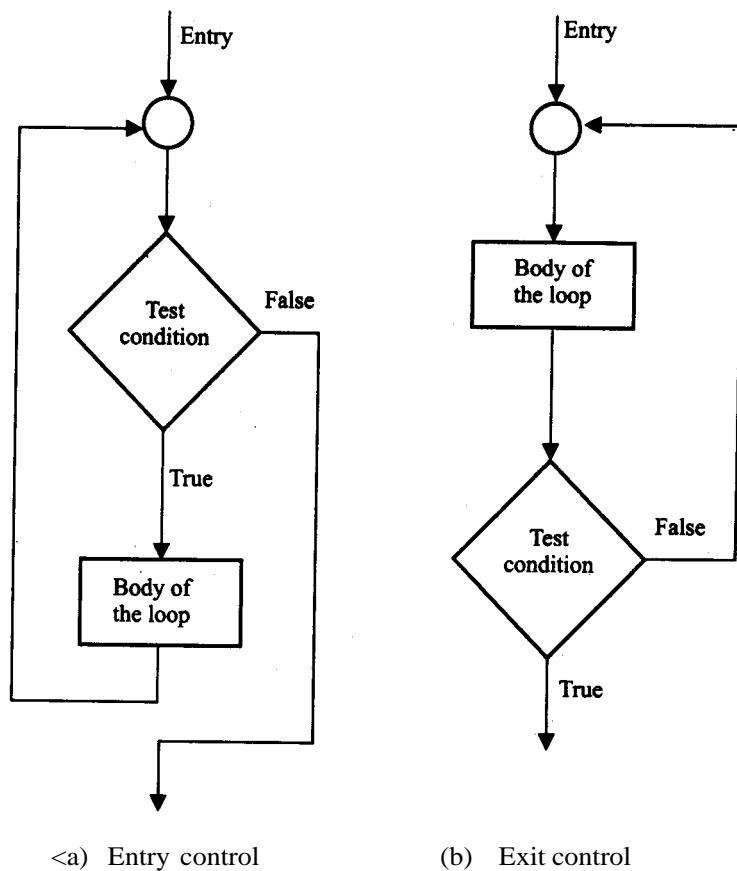
In looping, a sequence of statements are executed until some conditions for the termination of the loop are satisfied. A *program* loop therefore consists of two segments, one known as the *body of the loop* and the other known as the control statement. The *control statement* tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled* loop or as *exit-controlled* loop. The flowcharts in Fig. 7.1 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the *tst* condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an *infinite* loop and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a counter.
2. Execution of the statements in the loop.

**Fig. 7.1**

Loop control structures

3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The test may either be to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met with.

The Java language provides for three constructs for performing loop operations. They are:

- 1 The while statement
- 2 The do statement
- 3 The for statement

We shall discuss the features and applications of each of these statements in this chapter.



## 7.2 THE WHILE STATEMENT

The simplest of all the looping structures in Java is the **while** statement. The basic format of the **while** statement is

```
Initialization;
while (test condition)
{
    Body of the loop
}
```

The while is an *entry-controlled* loop statement. The *test* condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

Consider the following code segment:

```
.....
.....
sum = 0;
n = 1;

while (n <= 10)
{
    sum = sum + n * n;
    n = n+1;
}
System.out.print("Sum = "+ sum);
.....
.....
```

The body of the loop is executed 10 times for  $n = 1, 2, \dots, 10$  each time adding the square of the value of  $n$ , which is incremented inside the loop. The test condition may also be written as  $!n < 11$ , the result would be the same. Program 7.1 illustrates the use of the while for reading a string of characters from the keyboard. The loop terminates when  $c = '\n'$ , the newline character.

## 7.1 UsintJ while loop

---

```

class WhileTest
{
    public static void main(String args[])
    {
        StringBuffer string = new StringBuffer();
        char c;
        System.out.println("Enter a string H");
        try
        {
            while ((c = (char)System.in.read()) != '\n')
            {
                string.append(c); // Append character
            }
        }
        catch (Exception e)
        {
            System.out.println("Error in input");
        }
        System.out.println("You have entered " + string);
        System.out.println(string);
    }
}

```

---

Given below is the output of Program 7.1:

```

Enter a string
Java is a true Object-Oriented Language
      You have entered ...
Java is a true Object-Oriented' Language

```



### THE DO STATEMENT

The **while** loop construct that we have discussed in the previous section makes a test condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes the form:

```

Initialization;
do
{
    Body of the loop
}
while (test condition)

```

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test condition* in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

Since the *test condition* is evaluated at the bottom of the loop, the do...while construct provides an *-controlled* loop and therefore the body of the loop is always executed at least once.

Consider an example:

```

.....
.....
.i = 1;
sum = 0;

do
{
    sum = sum + i;
    i = i+2;
}
while(sum < 40 || i < 10);
.....
.....

```

The loop will be executed as long as one of the two relations is true. Program 7.2 illustrates the use of do...while loops for printing a multiplication table.

---

#### Program 7.2 Println, multiplication table usln, do...while loop

---

```

class DoWhileTest
{
    public static void main (String args[ ])
    {
        int row, column, y;
        System.out.println("Multiplication Table \n");
        row = 1;

```

(Continued)

---

**Program 7.2 (Continued)**

---

```
do
{
    column = 1;

    do
    {
        y = row * column;
        System.out.print("      " + y);
        column = column + 1;
    }
    while (column <= 3);

    System.out.println("\n");
    row = row + 1;
}
while (row <= 3);
```

---

Program 7.2 uses two do-while loops in nested form and produces the following output:

Multiplication Table

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 4 | 6 |
| 3 | 6 | 9 |



## THE FOR STATEMENT

The for loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the for loop is

```
for (initialization ; test condition ; increment)
{
    Body of the loop
}
```

The execution of the for statement is as follows:

1. *Initialization* of the *control Variables* is done first, using assignment statements such as `i = 1` and `count = 0`. The variables `i` and `count` are known as loop-control variables.
2. The value of the control variable is tested using the *test condition*. The test condition is a relational expression, such as `i < 10` that determines when the loop will exit. If the

condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.

3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as  $i = i + 1$  and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

Consider the following segment of a program

```
for (x = 0 ; x <= 9 ; x = x+1)
{
    System.out.println(x);
}
```

This for loop is executed 10 times and prints the digits 0 to 9 in a vertical line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the increment section,  $x = x + 1$ .

The for statement allows for negative increments. For example, the loop discussed above can be written as follows:

```
for (x = 9 ; x > ~ 0 ; x = x-1)
    System.out.println(x);
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9 ; x < 9; x = x-1)
{
    .....
    .....
}
```

will never be executed because the test condition fails at the very beginning itself.

Let us consider the problem of sum of squares of integers discussed in Section 7.2. This problem can be coded using the for statement as follows:

```
.....
.....
sum = 0;
for (n = 1; n <= 10; n = n+1)
{
    sum = sum + n*n;
}
.....
.....
```

## The body of the loop

```
sum = sum + n*n;
```

is executed 10 times for  $n = 1, 2, \dots, 10$  each time incrementing the sum by the square of the value of  $n$ .

One of the important points about the for loop is that all the three actions, namely *initialization*, *testitW* and *incrementitW*, are placed in the for statement itself, thus making them visible to the programmers and users, in one place. The for statement and its equivalent of while- and do statements are shown in Table 7.1.

Table 7.1 Comparison of the Three Loops

| for                                                                  | while                                                         | do                                                                   |
|----------------------------------------------------------------------|---------------------------------------------------------------|----------------------------------------------------------------------|
| for (n=1;n<=10;++n)<br>{<br>.....<br>.....<br>}<br>.....<br>n = n+1; | n = 1<br>while (n<=10)<br>{<br>.....<br>.....<br>}<br>n = n+1 | n = 1<br>do<br>{<br>.....<br>.....<br>}<br>n = n+1<br>while (n<=10); |

Program 7.3 illustrates the use of for loop for computing and printing the "power of 2" table.

## Program 7.3 Computln, the 'power of 2' usln, for loop

```
class ForTest  
{  
    public static void main (String args[ ])  
    {  
        long p;  
        int n;  
        double q;  
        System.out.println("2      to power -n      n      2 to power n");  
  
        p = 1;  
        for (n = 0; n < 10; ++n)  
        {  
            if (n == 0)  
                p = 1;  
            else  
                p = p * 2;  
            q = 1.0 / (double)p;
```

(Continued)

## Program 1.3 (Continued)

```

        System.out.println("      " + q + "      " + n +
                "      " + p);
    }
}
}

```

---

Output of Program 7.3 would be:

| 2 to power -n | n | 2 to power n |
|---------------|---|--------------|
| 1             | 0 | 1            |
| 0.5           | 1 | 2            |
| 0.25          | 2 | 4            |
| 0.125         | 3 | 8            |
| 0.0625        | 4 | 16           |
| 0.03125       | 5 | 32           |
| 0.015625      | 6 | 64           |
| 0.00390625    | 7 | 128          |
| 0.00195313    | 8 | 256          |
| 0.00195313    | 9 | 512          |

### Additional Features of for Loop

The for loop has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the for statement. The statements.

```

p = 1;
for (n=0; n<17; ++n)

```

can be rewritten as

```

for (p=1, n=0; n<17; ++n)

```

Notice that the initialization section has two parts  $p = 1$  and  $n = 1$  separated by a comma. Like the initialization section, the increment section may also have more than one part. For example, the loop

```

for (n=1, m=50; n<=m; n=n+1, m=rn-1
{
    .....
    .....
}

```

is perfectly valid, The multiple arguments in the increment section are separated by *commas*.

The third feature is that the test condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example that follows:

```
sum = 0 ;
for (i = 1, i < 20 && sum < 100: Hi)
{
    -----
    -----
}
```

The loop uses a compound test condition with the control variable *i* and external variable *sum*. The loop is executed as long as both the conditions *i < 20* and *sum < 100* are true. The *sum* is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialization and increment sections. For example, a statement of the type

```
for (x = (m+n)/2: x > 0: x = x/2)
```

is perfectly valid.

Another unique aspect of for loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
-----
-----
m = 5:
for ( :m != 100 : )
{
    System.out.println(m):
    m = m+5:
}
-----
-----
```

Both the initialization and increment sections are omitted in the **for** statement. The initialization has been done before the **for** statement and the control variable is incremented inside the loop. In such cases, the sections are left blank. However, the semicolons separating the sections must remain. If the test condition is not present, the for statement sets up an infinite loop.

We can set up time delay loops using the null statement as follows:

```
for (j = 1000: j > 0: j = -j-1)
:
```

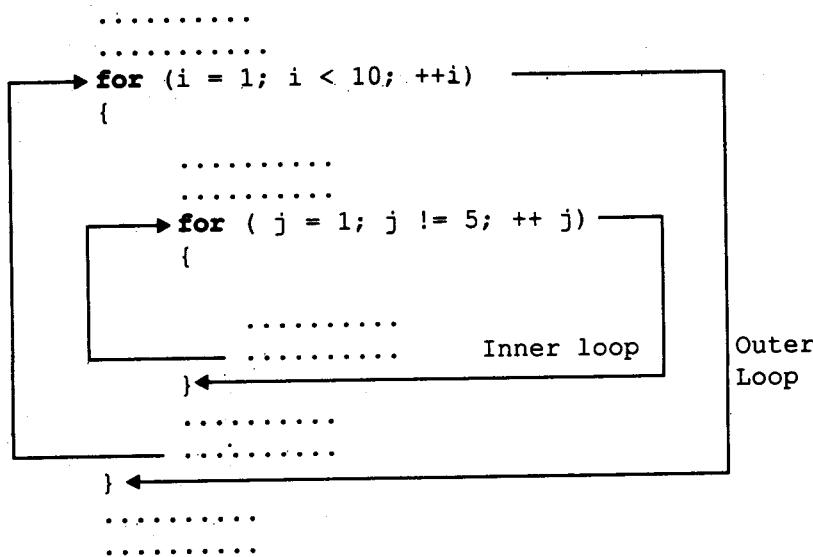
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *empty* statement. This can also be written as

```
for (j=1000: j > 0: j = j-1):
```

This implies that the compiler will not give an error message if we place a semicolon by mistake at the end of a for statement. The semicolon will be considered as an *empty* statement and the program may produce some nonsense.

## Nesting of for Loops

Nesting of loops, that is, one for statement within another for statement, is allowed in Java. We have used this concept in Program 7.2. Similarly, for loops can be nested as follows:



The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each for statement.

A program segment to print a multiplication table using for loops is shown below:

```

.....
.....
for (row = 1; row <= ROWMAX; ++row)
{
    for (column = 1; column <= COLMAX; ++ column)
    {
        Y = row * column
        System.out.print(" " + Y);
    }
    System.out.println();
}
.....
.....

```

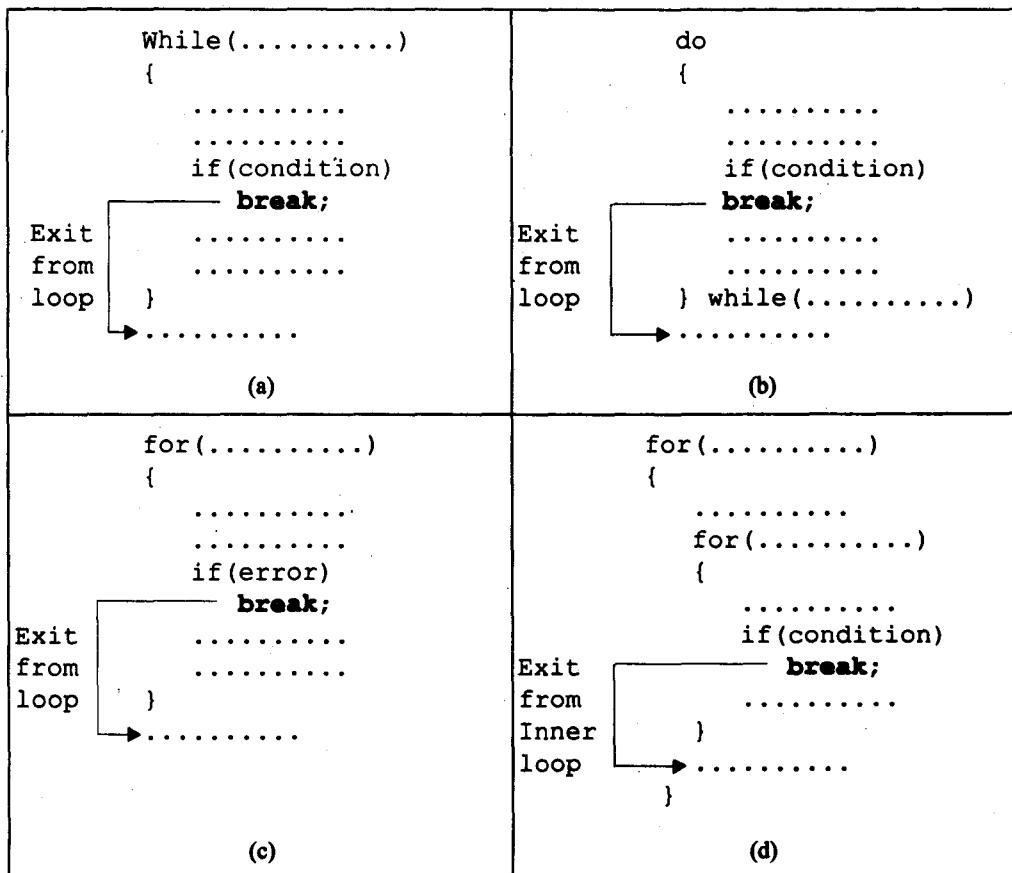
The outer loop controls the rows while the inner one controls the columns.



## JUMPS IN LOOPS

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names a 100 times must be terminated as soon as the desired name is found. Java permits a jump from one statement to the end or beginning of a loop as well as a jump out of a loop.

Fig. 7.2



**Exiting a loop with break statement**

## Jumping Out of a Loop

An early exit from a loop can be accomplished by using the break statement. We have already seen the use of the break in the switch statement. This statement can also be used within while, do or for loops as illustrated in Fig. 7.2.

When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

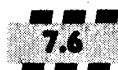
## Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the break statement, Java supports another similar statement called the continue statement. However, unlike the break which causes the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is simply

**Continue;**

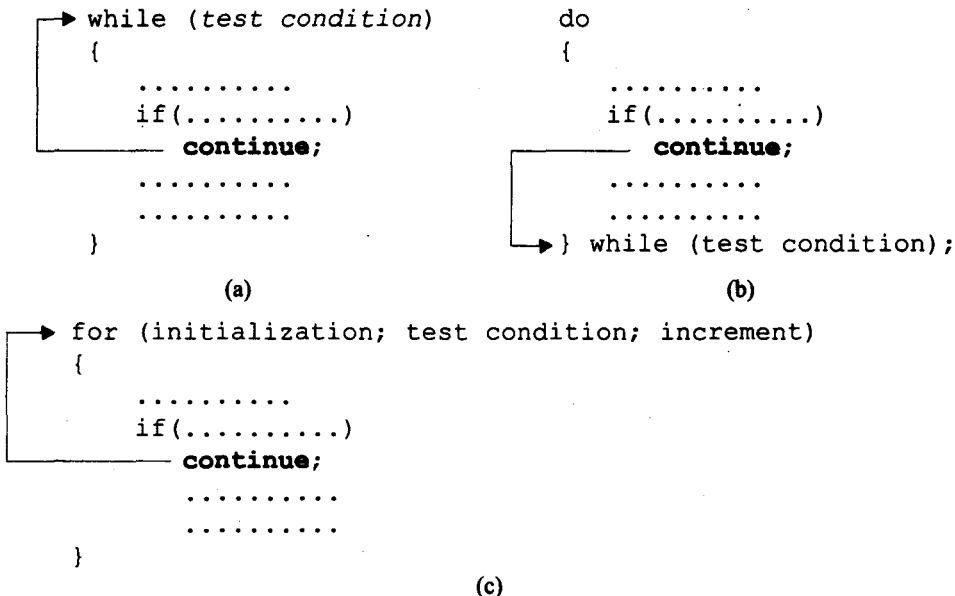
The use of the continue statement in loops is illustrated in Fig. 7.3. In while and do loops, continue causes the control to go directly to the *test condition* and then to continue the iteration process. In the case of for loop, the *increment* section of the loop is executed before the *test condition* is evaluated.



## LABELLED LOOPS

In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, place it before the loop with a colon at the end. Example:

```
loop1 :    for (          )
{           .....
.....
}
. . .
```

**Fig. 7.3*****Bypassing and continuing in loops***

A block of statements can be labelled as shown below:

```

block1:      {
    .....
    .....
}

block2:      {
    ...
    ...
    }
    .....
}

```

We have seen that a simple break statement causes the control to jump outside the nearest loop and a simple continue statement restarts the current loop. If we want to jump outside a nested loops or to continue a loop that is outside the current one, then we may have to use the labelled break and labelled continue statements. Example:

```

→ outer:   for (int m=1; m<11; m++)
{
    for (int n=1; n<11; n++)
    {
        System.out.print(" " + m*n);
        if ( n == m )
            continue outer;
    }
}

```

Here, the **continue** statement terminates the inner loop when  $n = m$  and continues with the next iteration of the outer loop (counting  $m$ ).

Another example:

```

L L
o o
o o
P P
1 2
→ loop1:   for (int i = 0; i < 10; i++)
{
    → loop2:   while (x < 100)
    {
        y = i * x;
        if (y > 500)
            break loop1; ——————
        .....
        .....
        .....
    }
    .....
}

```

Jumping  
out of  
both loops

Here, the label **loop1** labels the outer loop and therefore the statement

**break loop1;**

causes the execution to break out of both the loops. Program 7.4 illustrates the use of **break** and **continue** statements.

#### Program 7.4 Use of **continue** and **break** statements

```

class ContinueBreak
{
    public static void main(String args[])
    {
        LOOP1 : for (int i = 1; i < 100; i++)
        {
            System.out.println(" ");

```

(Continued)

---

```

        if (i >= 10) break;
        for (int j = 1; j < 100; j++)
        {
            System.out.print("    * ");
            if (j == i)
                continue LOOP1;
        }
        System.out.println("Termination      by BREAK");
    }
}

```

---

Program 7.4 produces the following output:

```

*
*
* *
*
* * *
*
* * * *
*
* * * * *
*
* * * * * *
*
* * * * * * *
*
* * * * * * * *
Termination by BREAK

```



The IOQpsand conditional statements covered here and in Chapter 6 are the backbone of any programming language. We discussed in this chapter the following loop constructs:

- while structure
- do structure
- for structure

They would be extremely useful in developing concise, compact and structured programs. We have also seen how to use the break and continue statements to skip or jump out of a loop, if need be.

#### KEY TERMS

**Looping, Infinite Loop, Entry-control, Exit-control, Nesting, Empty Statement, Labelled Loops, Continue.**

**REVIEW QUESTIONS**

- 7.1 Compare in terms of their functions, the following pairs of statements:
- while and do....while.
  - while and for.
  - break and continue.
- 7.2 Analyze each of the program segments that follow and determine how many times the body of each loop will be executed.
- |                                                                            |                                                                                                    |
|----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| (a) <pre>x = 5; y = 50; while (x &lt;= y) {     x = y/x;     ..... }</pre> | (b) <pre>m = 1; dQ {     .....     m = m+2; } while (m &lt; 10)</pre>                              |
| (c) <pre>int i; for(i=0; i&lt;=5; i = i+2/3) {     ..... }</pre>           | (d) <pre>int m = 10; int n = 7 while (m % n &gt;= 0) {     .....     m = m+1;     n = n+2; }</pre> |
- 7.3 Find errors, if any, in each of the following looping segments. Assume that all the variables have been declared and assigned values.
- |                                                                                                  |                                                                                                          |
|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| (a) <pre>while (count != 10); {     count = 1;     sum = sum + x;     count = count + 1; }</pre> | (b) <pre>name = 0; do { name = name + 1; System.out.println("My name is John\n"); while (name = 1)</pre> |
| (c) <pre>for(x=1, x&gt;10; x = x+1) {     ..... }</pre>                                          | (d) <pre>m = 1; n = 0; for (; m+n &lt; 19; ++n) System.out.println("Hello \n"); m = m+10;</pre>          |
- 7.4 What is an empty statement? Explain its usefulness.

7.5 Given a number, write a program using while loop to reverse the digits of the number.

For example, the number

12345

should be written as

54321

(Hint: Use modulus operator to extract the last digit and the integer division by 10 to get the n-1 digit number from the n digit number).

7.6 The factorial of an integer m is the product of consecutive integers from 1 to m. That is

$$\text{factorial } m = m! = m \cdot (m-1) \cdot \dots \cdot 1.$$

Write a program that computes and prints a table of factorials for any given m.

7.7 Write a program to compute the sum of the digits of a given integer number.

7.8 The numbers in the sequence

1 1 2 3 5 8 13 21 ....

are called Fibonacci numbers. Write a program using a do...while loop to calculate and print the first m Fibonacci numbers.

(Hint: After the first two numbers in the series, each number is the sum of the two preceding numbers).

7.9 Write a program to evaluate the following investment equation

$$V = P(1+r)^n$$

and print the tables which would give the value of V for various combination of the following values of P, rand n.

P: 1000, 2000, 3000, ..., 10,000

r: 0.10, 0.11, 0.12, ..., 0.20

n: 1, 2, 3, ..., 10

(Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$$V = P(I+r)$$

$$P=V$$

That is, the value of money at the end of first year becomes the principal amount for the next year and so on).

7.10 Write a program to print the following outputs using for loops.

(a) 1

2 2'

333

44-4 4

5 5 5 5 5

(b) g g g g g

gg g g g

g g g

g g

g

(c) 1

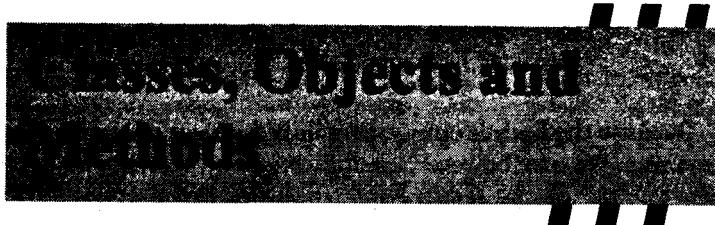
2 2

333

4 4 4 4

55555

# **Chapter 8**



## **INTRODUCTION**

Java is a true object-oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the *state* and *behaviour* of the basic program components known as *objects*. Classes create objects and objects use methods to communicate between them. That is all about object-oriented programming.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java, the data items are called *fields* and the functions are called *methods*. Calling a specific method in an object is described as sending the object a message.

A class is essentially a description of how to make an object that contains fields and methods. It provides a sort of *template* for an object and behaves like a basic data type such as int. It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OOP concepts such as *encapsulation*, *inheritance* and *polymorphism*.

## **DEFINING A CLASS**

As stated earlier, a class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create "variables" of that type using declarations that are similar to the basic type declarations. In Java, these variables are termed as *instances* of classes, which are the actual *objects*. The basic form of a class definition is:

```
class classname [extends superclassname]
{
    [ variable declaration; ]
    [ methods declaration; ]
}
```

Everything inside the square brackets is optional. This means that the following would be a valid class definition:

```
class Empty
{ }
```

Because the body is empty, this class does not contain any properties and therefore cannot do anything. We can, however, compile it and even create objects using it. C++ programmers may note that there is no semicolon after closing brace.

*classname* and *superclassname* are any valid Java identifiers. The keyword *extends* indicates that the properties of the *superclassname* class are extended to the *classname* class. This concept is known as *Inheritance* and is discussed in Section 8.11.

## ADDING VARIABLES

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called *instance Variables* because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables. Example:

```
class Rectangle
{
    int length;
    int width;
}
```

The class Rectangle contains two integer type instance variables. It is allowed to declare them in one line as

```
int length, width;
```

Remember these variables are only declared and therefore no storage space has been created in the memory. Instance variables are also known as *member Variables*.

## ADDING METHODS

A class with only data fields (and without methods that operate on that data) has no life. The objects created by such a Class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside

the body of the class but immediately after the declaration of instance variables. The general form of a method declaration is

```
type methodname (parameter-list)
{
    method-body;
}
```

Method declarations have four basic parts:

- The name of the method (*methodname*)
- The type of the value the method returns (*type*)
- A list of parameters (*parameter-list*)
- The body of the method

The type specifies the type of value the method would return. This could be a simple data type such as int as well as any class type. It could even be void type, if the method does not return any value. The *methodname* is a valid identifier. The *parameter list* is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. The variables in the list are separated by commas. In the case where no input data are required, the declaration must retain the empty parentheses. Examples:

|                           |                            |
|---------------------------|----------------------------|
| (int m, float x, float y) | <i>// Three parameters</i> |
| ()                        | <i>// Empty list</i>       |

The *body* actually describes the operations to be performed on the data. Let us consider the Rectangle class again and add a method `getData ()` to it.

```
class Rectangle
{
    int length;
    int width;

    void getData(int x, int y)
    {
        length = x;
        width = y;
    }
}
```

Note that the method has a return type of void because it does not return any value. We pass two integer values to the method which are then assigned to the instance variables length and width. The `getData` method is basically added to provide values to the instance variables. Notice that we are able to use directly length and width inside the method.

Let us add some more properties to the class. Assume that we want to compute the area of the rectangle defined by the class. This can be done as follows:

```

class Rectangle
{
    .int length, width; // Combined declaration

    void getData(int x , int y)
    {
        length = x ;
        width = y ;
    }
    int rectArea( )
    {
        int area = length * width;
        return(area);
    }
}.

```

---

The new method `rectArea()` computes area of the rectangle and *returns* the result. Since the result would be an integer, the return type of the method has been specified as `int`. Also note that the parameter list is empty.

Remember that while the declaration of instance variables (and also local variables) can be combined as

```
int length, width;
```

the parameter list used in the method header should always be declared independently separated by commas. That is,

```
void getData(int x,y) // Incorrect
```

is illegal.

Now, our class `Rectangle` contains two instance variables and two methods. we can add more variables and methods, if necessary.

Most of the times when we use classes, we will have many methods and variables within the class. Instance variables and methods in classes are accessible by all the methods in the class but a method cannot access the variables declared in other methods. Example:

```

class Access
{
    int x ;

    void method1 ( )
    {
        int y ;
        x = 10 ;      // legal
        y = x ;      // legal
    }
}

```

```

void method2( )
{
    int z ;
    x = 5 ;           // legal
    z = 10;          // legal
    y = 1 ;          // illegal
}

```

---



## CREATING OBJECTS

As pointed out earlier, an object in Java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as *instantiating* an object.

Objects in Java are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type Rectangle.

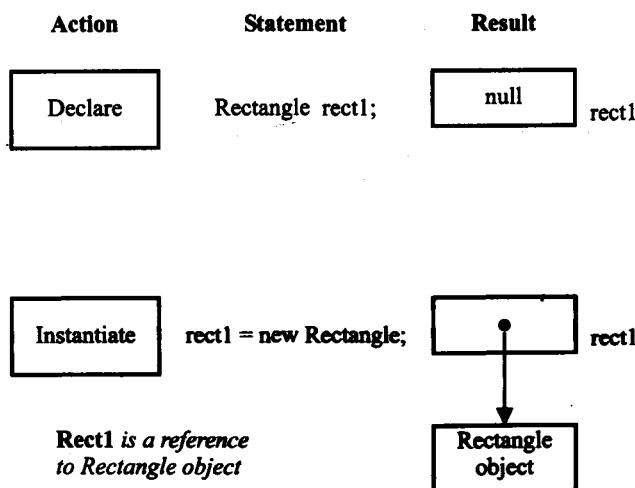
```

Rectangle rect1;           // declare
rect1 = new Rectangle();    // instantiate

```

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable rect1 is now an object of the Rectangle class (see Fig. 8.1)

**Fig. 8.1**



**Creating object references**

Both statements can be combined into one as shown below:

```
Rectangle rect1 = new Rectangle();
```

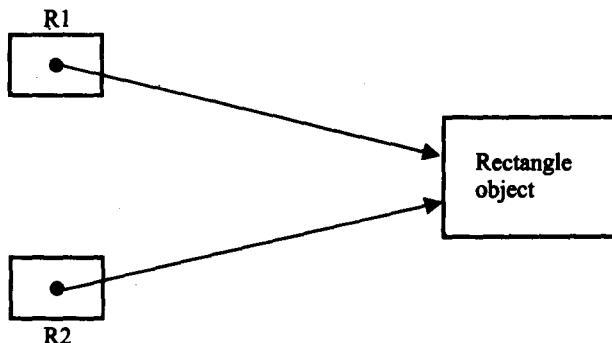
The method `Rectangle()` is the default constructor of the class. We can create any number of objects of `Rectangle`. Example:

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
and so on.
```

It is important to understand that each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another. It is also possible to create two or more references to the same object (see Fig. 8.2).

**Fig. 8.2**

```
Rectangle R1 = new Rectangle();
Rectangle R2 = R1;
```



Both R1 and R2 refer to the same object.

Asslpln, one object reference variable to ,mother



## ACCESSING CLASS MEMBERS

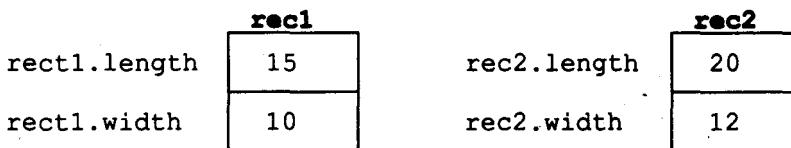
Now that we have created objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program. Remember, all variables must be assigned values before they are used. Since we are outside the class, we cannot access the instance variables and the methods directly. To do this, we must use the concerned object and the `dot` operator as shown below:

```
objectname. variable name
objectname . methodname (parameter-list);
```

Here *objectname* is the name of the object, *<variablename>* is the name of the instance variable Inside the object that we wish to access, *methodname* is the method that we wish to call, and *parameter-list* is a comma separated list of "actual values" (or expressions) that must match in type and number with the parameter list of the methodname declared in the class. The instance variables of the Rectangle class may be accessed and assigned values as follows:

```
rect1.length      = 15;
rect1.width       = 10;
rect2.length      = 20;
rect2. width      = 12;
```

Note that the two objects rect1 and rect2 store different values as shown below:



This is one way of assigning values to the variables in the objects. Another way and more convenient way of assigning values to the instance variables is to use a method that is declared inside the .class.

In our case, the method `getData` can be used to do this work. We can call the `getData` method on any Rectangle object to set the values of both length and width. Here is the code segment to achieve this.

```
Rectangle rect1 = new Rectangle(); // Creating an object
rect1. getData(15,10); // Calling the method using the object
```

This code creates `rect1` object and then passes in the values 15 and 10 for the x and y parameters of the method `getData`. This method then assigns these values to length and width variables respectively. For the sake of convenience, the method is again shown below:

```
void getData(int x, int y)
{
    length = x;
    width = Y ;
}
```

Now that the object `rect1` contains values for its variables, we can compute the area of the rectangle represented by `rect1`. This again can be done in two ways.

- The first approach is to access the instance variables using the dot operator and compute the area. That is,
 

```
int areal = rect1.length * rect1. width ;
```
- The second approach is to call the method `rectArea` declared inside the class. That is,
 

```
int areal = rect1. rectArea(); // Calling the method
```

Program 8.1 illustrates the concepts discussed so far.

## Pi..... B.1 Application of classes and objects

---

```
class Rectangle
{
    int length, width;           II Declaration of variables
    void getData(int x, int y)   II Definition of method
    {
        length = x;
        width = y;
    }

    int rectArea ()             II Definition of another method
    {
        int area = length * width;
        return (area);
    }
}

class RectArea                   II Class with main method
{
    public static void main (String args[])
    {
        int areal, area2;
        Rectangle rect1 = new Rectangle();           II Creating objects
        Rectangle rect2 = new Rectangle();

        rect1.length = 15;                         II Accessing variables
        rect1.width = 10;

        areal = rect1.length * rect1.width;

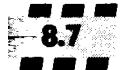
        rect2.getData(20, 12);                     II Accessing methods
        area2 = rect2.rectArea();

        System.out.println("Areal      = " + areal);
        System.out.println("Area2      = " + area2);
    }
}
```

---

Program 8.1 would output the following:

```
Areal = 150
Area2 = 240
```



## CONSTRUCTORS

We know that all objects that are created must be given initial values. We have done this earlier using two approaches. The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.

The second approach takes the help of a method like `getData` to initialize each object individually using statements like,

```
rect1.getData(15,10);
```

It would be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a *constructor*, that enables an object to initialize itself when it is created.

Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even void. This is because they return the instance of the class itself.

Let us consider our Rectangle class again. We can now replace the `getData` method by a constructor method as shown below:

---

```
class Rectangle
{
    int length ;
    int width ;

    Rectangle(int x, int y) // Constructor method
    {
        length = x ;
        width = y ;
    }
    int rectArea()
    {
        return(length * width);
    }
}
```

---

Program 8.2 illustrates the use of a constructor method to initialize an object at the time of its creation.

### *Program 8.2 Application of constructors*

---

```
class Rectangle
{
    int length, width ;
    Rectangle(int x, int y)           // Defining constructor
    {
```

---

*Continued*

---

**Pr.gram 8.2 (Continued)**

---

```
length = x ;
width = Y ;
}

int rectArea()
{
    return (length * width);
}

class RectangleArea
{
    public static void main (string args[])
    {
        Rectangle rect1 = new Rectangle (15,10) ; // Calling constructor
        int areal = rect1.rectArea() ;
        System.out.println("Area1 = " + areal) ;
    }
}
```

---

Output of Program 8.2:

Areal = 150



## METHODS OVERLOADING

In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called *method Overloading*. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as *polymorphism*.

To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique. Note that the method's return type does not play any role in this. Here is an example of creating at) overloaded method.

---

```
class Room'
{
    float length ;
    float breadth ;
```

```

Room(float    x,  float    y)           // constructor 1
{
    length    =  x  ;
    breadth   =  y  ;
}

Room(float    x)                      // constructor2
{
    length    =  breadth   =  x  ;
}

int  area(  )
{
    return    (length    *  breadth)    ;
}
}

```

---

Here, we are overloading the constructor method Room( ). An object representing a rectangular room will be created as

Room room1 = new Room(25.0,15.0); // using constructor1

On the other hand, if the room is square, then we may create the corresponding object as

Room room2 = new Room(20.0); // using constructor2



## STATIC MEMBERS

We have seen that a class basically contains two sections. One declares variables and the other declares methods. These variables and methods are called *instance Variables* and *instance methods*. This is because every time the class is instantiated, a new copy of each of them is created. They are accessed using the objects (with dot operator).

Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

```

static int count  ;
static int max(intx,      int y);

```

The members that are declared static as shown above are called *static members*. Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as *class Variables* and *class methods* in order to distinguish them from their counterparts, instance variables and instance methods.

Static variables are used when we want to have a variable common to all instances of a class. One of the most common examples is to have a variable that could keep a count of how many

objects of a class have been created. Remember, Java creates only one copy for a static variable which can be used even if the class is never actually instantiated.

Like static variables, static methods can be called without using the objects. They are also available for use by other classes. Methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. Java class libraries contain a large number of class methods. For example, the Math class of Java library defines many static methods to perform math operations that can be used in any program. We have used earlier statements of the types

```
float x = Math.sqrt(25.0);
```

The method sqrt is a class method (or static method) defined in Math class.

We can define our own static methods as shown in Program 8.3.

### **Program 8.3 Defining and using static members**

---

```
class Mathoperation
{
    static float mul(float x, float y);
    {
        return x*y;
    }

    static float divide(float x, float y)
    {
        return x/y ;
    }
}

class MathApplication
{
    public void static main (string args[ ])
    {
        float a = MathOperation.mul(4.0,5.0)      ;
        float b = MathOperation.divide(a,2.0)      ;
        System.out.println(b      = +   b)      ;
    }
}
```

---

Output of Program 8.3:

```
b = 10.0
```

Note that the static methods are called using class names. In fact, no objects have been created for use. Static methods have several restrictions:

1. They can only call other static methods.

2. They can only access static data.
3. They cannot refer to this or super in any way.

## NESTING OF METHODS

We discussed earlier that a method of a class can be called only by an object of that class (or class itself, in the case of static methods) using the dot operator. However, there is an exception to this. A method can be called by using only its name by another method of the same class. This is known as *nesting of methods*.

Program 8.4 illustrates the nesting of methods inside a class. The class Nesting defines one constructor and two methods, namely largest() and display(). The method display() calls the method largest( ) to determine the largest of the two numbers and then displays the result.

### - Program 8.4 Nesting of methods

---

```
class Nesting
{
    int m, n;

    Nesting(int x, int y)           // constructor method
    {
        m = x;
        n = y;
    }

    int largest( )
    {
        if(m == n)
            return (m);
        else
            return(n);
    }

    void display( )
    {
        int large = largest( );    // calling a method
        System.out.println("largest value = " +large);
    }
}

class NestingTest
{
```

---

(Continued)

**Program 8.4 (C)Continued)**


---

```

public static void main (String args[ ] )
{
    Nesting nest = new Nesting(50,      40) ;
    nest.display(      );
}

```

---

Output of Program 8.4 would be:

Largest value = 50

A method can call any number of methods. It is also possible for a called method to call another method. That is, method 1 may call method2, which in turn may call method3.



## 8.11 INHERITANCE: EXTENDING A CLASS

Reusability is yet another aspect of OOP paradigm. It is always nice if we could reuse something that already exists rather than creating the same all over again. Java supports this concept. Java classes can be reused in several ways. This is basically done by creating new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called *inheritance*. The old class is known as the *base class* or *super class* or *parent class* and the new one is called the *subclass* or *derived class* or *child class*.

The inheritance allows subclasses to inherit all the variables and methods of their parent classes. Inheritance may take different forms:

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many subclasses)
- Multilevel inheritance (Derived from a derived class)

These forms of inheritance are shown in Fig. 8.3. Java does not directly implement multiple inheritance. However, this concept is implemented using a secondary inheritance path in the form of *interfaces*. Interfaces are discussed in Chapter 10.

### Defining a Subclass

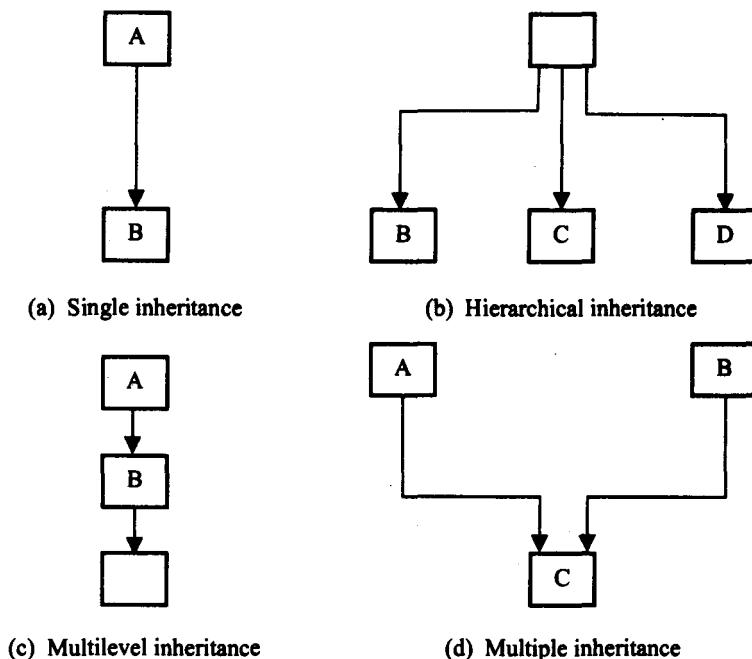
A subclass is defined as follows:

```

class subclassname extends superclassname
{
    variables declaration ;
    methods declaration ;
}

```

The keyword **extends** signifies that the properties of the *superclassname* are extended to the *subclassname*. The subclass will now contain its own variables and methods as well those of the

**Fig. 8.3**

#### Forms of Inheritance

superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it. Program 8.5 illustrates the concept of single inheritance.

#### Program 8.5 Application of *single* Inheritance

```
class Room
{
    int length;
    int breadth;
    Room(int' x, int y)
    {
        length      =  x;
        breadth     =  y;
    }
    int area ( )
    {
        return  (length * breadth);
    }
}
```

(Continued)

**Program 8.5 (Continued)**

---

```
class BedRoom extends Room           // Inheriting Room
{
    int height;

    BedRoom(int x, int y, int z)
    {
        super(x,y);                // pass values to superclass
        height = z;
    }

    int volume ( )
    {
        return (length * breadth * height);
    }
}

class InherTest
{
    public static void main (String args[ ])
    {
        BedRoom rooml = new BedRoom(14,12,10);
        int areal = rooml.area();          // superclass method
        int volumel = rooml.volume();     // baseclass method
        System.out.println("Areal = "+ areal);
        System.out.println("Volumel = "+ volumel);
    }
}
```

---

The output of Program 8.5 is:

```
Areal = 168
Volumel = 1680
```

The program defines a class Room and extends it to another class BedRoom. Note that the class BedRoom defines its own data members and methods. The subclass BedRoom now includes three instance variables, namely, length, breadth and height and two methods, area and volume.

The constructor in the derived class uses the super keyword to pass values that are required by the base constructor. The statement

```
BedRoom rooml = new BedRoom(14,12,10);
```

calls first the BedRoom constructor method, which in turn calls the Room constructor method by using the super keyword.

Finally, the object room! of the subclass BedRoom calls the method area defined in the super class as well as the method volume defined in the subclass itself.

### Subclass Constructor

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method of the superclass. The keyword super is used subject to the following conditions.

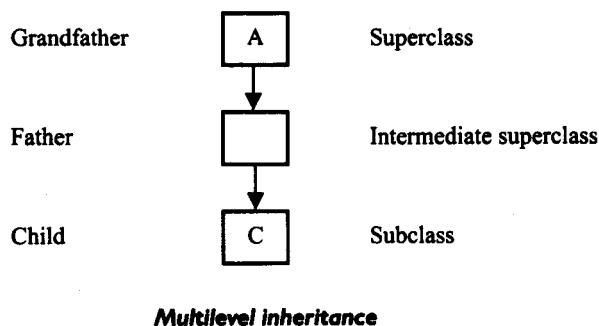
- super may only be used within a subclass constructor method
- The call to superclass constructor must appear as the first statement within the subclass constructor
- The parameters in the super call must match the order and type of the instance variable declared in the superclass.

Program 8.5 illustrated the use of super( ) method for passing parameters to a superclass.

### Multilevel Inheritance

A common requirement in object-oriented programming is the use of a derived class as a super class. Java supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes as shown in Fig. 8.4.

**Fig. 8.4**



The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The chain ABC is known as *inheritance path*.

A derived class with multilevel base classes is declared as follows:

```

class A
{
    .....
}

```

```

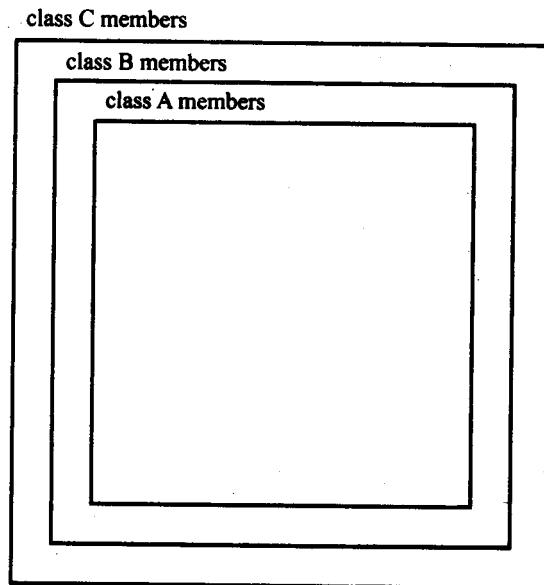
class B extends A      // First level
{
    .....
}

class C extends B      // Second level
{
    .....
}

```

This process may be extended to any number of levels. The class C can inherit the members of both A and B as shown in Fig. 8.5.

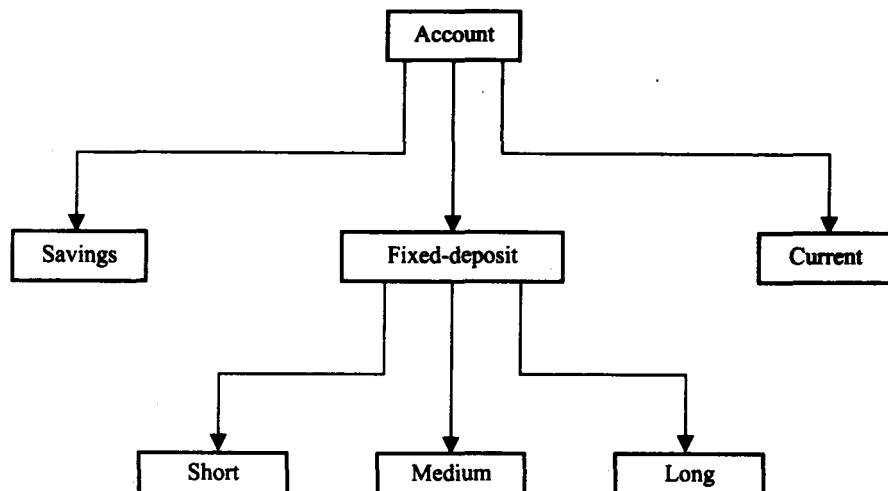
**Fig. 8.5**



**C contains B which contains A**

### Hierarchical Inheritance

Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level. As an example, Fig. 8.6 shows a hierarchical classification of accounts in a commercial bank. This is possible because all the accounts possess certain common features.

**Fig. 8.6*****Hierarchical classification of bank accounts***

## OVERRIDING METHODS

We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in subclasses without having to define the methods again in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behaviour when that method is called. That means, we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as *overriding*. Program 8.6 illustrates the concept of overriding. The method `display()` is overridden.

---

### Program 8.6 illustration of method overriding,

---

```

class Super
{
    int x ;
    Super(int x)
    {
        this.x = x ;
    }
  
```

---

(Continued)

---

**Program 8.6 (Continued)**

---

```
void diaplay( )           // method defined
{
    System.out.println("Super      x = " + x);
}
}

class Sub extends Super
{
    int y ;
    Sub(int x, int y)
    {
        super(x) ;
        this.y = Y ;
    }

    void diaplay( )           // method defined again
    {
        System.out.println("Super      x = " + x) ;
        System.out.println("Sub      y = " + y) ;
    }
}

class OverrideTest
{
    public static void main(String args[])
    {
        Sub s1 = new Sub(100,200) ;
        s1.display() ;
    }
}
```

---

Output of Program 8.6:

```
Super x = 100
Sub Y = 200
```

## FINAL VARIABLES AND METHODS

All methods and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword **final** as a modifier. Example:

```
final int SIZE = 100 ;
final void showstatus( ) { ..... }
```

Making a method final ensures that the functionality defined in this method will not be altered in any way. Similarly, the value of a final variable can never be changed. Final variables, behave like class variables and they do not take any space on individual objects of the class.



## FINAL CLASSES

Sometimes we may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called *final class*. This is achieved in Java using the keyword final as follows:

```
final class Aclass { }
final class Bclass extends Someclass { }
```

Any attempt to inherit these classes will cause an error and the compiler will not allow it.

Declaring a class final prevents any unwanted extensions to the class. It also allows the compiler to perform some optimisations when a method of a final class is invoked.



## FINALIZER METHODS

We have seen that a constructor method is used to initialize an object when it's declared. This process is known as *initialization*. Similarly, Java supports a concept called *finalization*, which is just opposite to initialization. We know that Java run-time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or window system fonts. The garbage collector cannot free these resources. In order to free these resources we must use a *finalizer* method. This is similar to *destructors* in C++.

The finalizer method is simply finalize( ) and can be added to any class. Java calls that method whenever it is about to reclaim the space for that object. The finalize method should explicitly define the tasks to be performed.



## ABSTRACT METHODS AND CLASSES

We have seen that by making a method final we ensure that the method is not redefined in a subclass. That is, the method can never be subclassed. Java allows us to do something that is exactly opposite to this. That is, we can indicate that a method must always be redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword abstract in the method definition. Example:

```
abstract class Shape
{
    .....
    .....
    abstract void draw( );
    .....
}
```

When a class contains one or more abstract methods, it should also be declared abstract as shown in the example above.

While using abstract classes, we must satisfy the following conditions:

- We cannot use abstract classes to instantiate objects directly. For example,

```
Shape s = new Shape();
```

is illegal because shape is an abstract class.

- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.



## VISIBILITY CONTROL

We stated earlier that it is possible to inherit all the members of a class by a subclass using the keyword extends. We have also seen that the variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in Java by applying *visibility modifiers* to the instance variables and methods. The visibility modifiers are also known as *access modifiers*. Java provides three types of visibility modifiers: public, private and protected. They provide different levels of protection as described below.

### public Access

Any variable or method is visible to the entire class in which it is defined. What if we want to make it visible to all the classes outside this class? This is possible by simply declaring the variable or method as public. Example:

```
public int number;  
public void sum() { .....
```

A variable or method declared as public has the widest possible visibility and accessible everywhere. In fact, this is what we would like to prevent in many programs. This takes us to the next levels of protection.

### friendly Access

In many of our previous examples, we have not used public modifier, yet they were still accessible in other classes in the program. When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.

The difference between the "public" access and the "friendly" access is that the public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages. (A package is a group of related classes stored separately. They are explored in detail in Chapter 11). A package in Java is similar to a source file in C.

## protected Access

The visibility level of a "protected" field lies in between the public access and friendly access. That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Note that non-subclasses in other packages cannot access the "protected" members.

## private Access

private fields enjoy the highest degree of protection. They are accessible only within their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. A method declared as private behaves like a method declared as final. It prevents the method from being subclassed. Also note that we cannot override a non-private method in a subclass and then make it private.

## private protected Access

A field can be declared with two keywords private and protected together like:

```
private protected int codeNumber ;
```

This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package. Table 8.1 summarises the visibility provided by various access modifiers.

**Table 8.1 Visibility of Fields in a Class**

| Access location                  | Access modifier<br>→ | public | protected | friendly (default) | private protected | private |
|----------------------------------|----------------------|--------|-----------|--------------------|-------------------|---------|
| Same class                       |                      | Yes    | Yes       | Yes                | Yes               | Yes     |
| Subclass in same package         |                      | Yes    | Yes       | Yes                | Yes               | No      |
| Other classes in same package    |                      | Yes    | Yes       | Yes                | No                | No      |
| Subclass in other packages       |                      | Yes    | Yes       | No                 | Yes               | No      |
| Non-subclasses in other packages |                      | Yes    | No        | No                 | No                | No      |

## Rules of Thumb

The details discussed so far about field visibility may be quite confusing and seem complicated. Given below are some simple rules for applying appropriate access modifiers.

1. Use public if the field is to be visible everywhere.
2. Use protected if the field is to be visible everywhere in the current package and also subclasses in other packages.
3. Use "default" if the field is to be visible everywhere in the current package only.
4. Use private protected if the field is to be visible only in subclasses, regardless of packages.
5. Use private if the field is *not* to be visible anywhere except in its own class.



## SUMMARY

Classes, objects, and methods are the basic components used in Java programming. The concept of classes is at the root of Java's design. We have discussed in detail the following in this chapter:

- How to define a class
- How to create objects
- How to add methods to classes
- How to extend or reuse a class
- How to write application programs

We have also discussed various features that could be used to restrict the access to certain variables and methods from outside the class. The concepts discussed here provides the basics of writing not only standalone application programs but also applets for use on Internet.

## KEY TERMS



## REVIEW QUESTIONS

- 8.1 What is class? How does it accomplish data hiding?
- 8.2 How do classes help us to organise our programs?

- 8.3 What are the three parts of a simple, empty class?
- 8.4 What are objects? How are they created from a class?
- 8.5 How is a method defined?
- 8.6 When do we declare a member of a class static?
- 8.7 What is a constructor? What are its special properties?
- 8.8 How do we invoke a constructor?
- 8.9 What is inheritance and how does it help us create new classes quickly?

- 8.10 Describe different forms of inheritance with examples.
- 8.11 Describe the syntax of single inheritance in Java.
- 8.12 Compare and contrast overloading and overriding methods.

- 8.13 When do we declare a method or class final?
- 8.14 When do we declare a method or class abstract?
- 8.15 Discuss the different levels of access protection available in Java.

8.16 Design a class to represent a bank account. Include the following members:

Data members

- Name of the depositor
- Account number
- Type of account
- Balance amount in the account

Methods

- To assign initial values
- To deposit an amount
- To withdraw an amount after checking balance
- To display the name and balance

- 8.17 Modify the program of Question 8.16 to incorporate a constructor to provide initial values.
- 8.18 Assume that a bank maintains two kinds of account for its customers, one called savings account and the other current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level, a service charge is imposed.

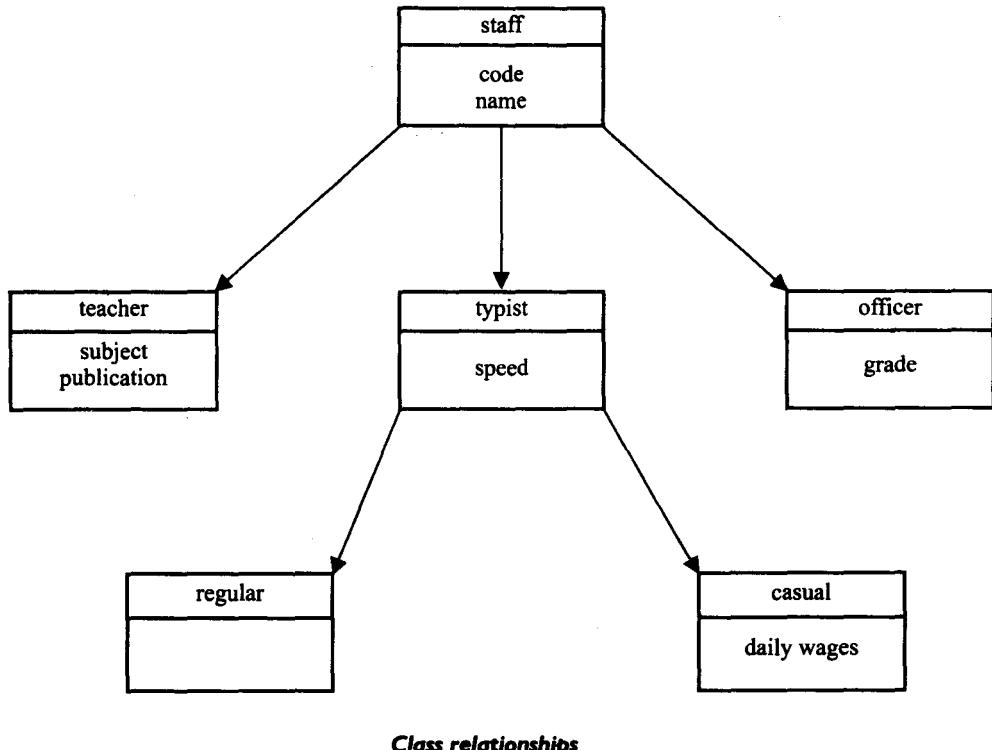
Create a class Account that stores customer name, account number and type of account. From this derive the classes Carr-acct and Sav-acct to make them more specific to their requirements. Include the necessary methods in order to achieve the following tasks:

- (a) Accept deposit from a customer and update the balance .
- (b) Display the balance.
- (c) Compute and deposit interest.
- (d) Permit withdrawal and update the balance.
- (e) Check for the minimum balance, impose penalty, if necessary, and update the balance.

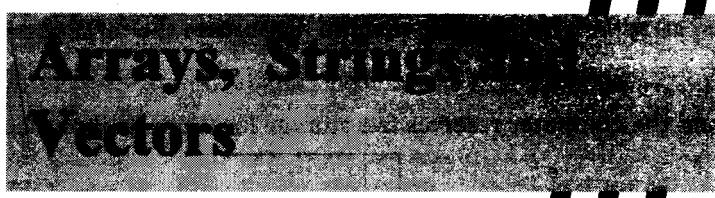
Do not use any constructors. Use methods to initialize the class members.

- 8.19 Modify the program of Question 8.18 to include constructors for all the three classes.
- 8.20 An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationships are shown in Fig. 8.7. The figure also shows the minimum information required for each class. Specify all the classes and define methods to create the database and retrieve individual information as and when required.

**Fig. 8.7**



# **Chapter 9**



## **ARRAYS**

An array is a group of contiguous or related data items that share a common name. For instance, we can define an array name salary to represent a set of salaries of a group of employees. A particular value is indicated by writing a number called *index* or *subscript* in brackets after the array name. For example,

salary[10]

represents the salary of the 10th employee. While the complete set of values is referred to as an *array*, the individual values are called *elements*. Arrays can be of any variable type.

The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, a loop with the subscript as the control variable can be used to read the entire array, perform calculations and, print out the results.



## **ONE-DIMENSIONAL ARRAYS**

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted* variable or a *one-dimensional* array. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^n x_i}{n}$$

to calculate the average of n values of x. The subscripted variable  $x_i$  refers to the *i*th element of x. In Java, single-subscripted variable x (can be expressed as

`x[1], x[2], x[3] ..... x[n]`

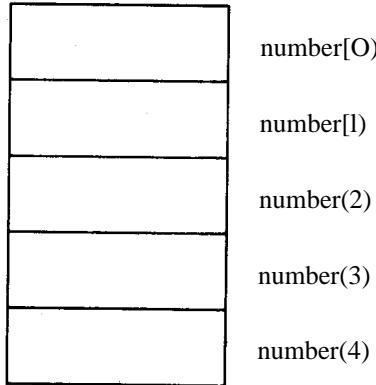
The subscript can begin with number 0. That is

`x[0]`

is allowed. For example, if we want to represent a set of five numbers, say (35,40,20,57,19), by an array variable number, then we may create the variable number as follows

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

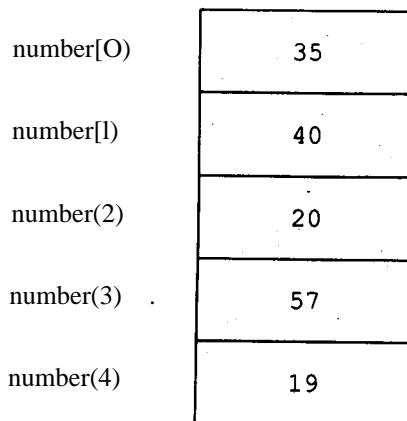
and the computer reserves five storage locations as shown below:



The values to the array elements can be assigned as follows:

```
number [0] = 35;  
number[1] = 40;  
number [2] = 20;  
number[3] = 57;  
number[4] = 19;
```

This would cause the array number to store the values shown as follows:



These elements may be used in programs just like any other Java variable. For example, the following are valid statements:

```
aNumber      =  number [0] + 10;
number [4]   =  number [0] + number[2];
number [2]   =  xeS] + y[10];
value[6]     =  number[i] * 3;
```

The subscript of an array can be integer constants, integer variables like **i**, or expressions that yield integers.

## CREATING AN ARRAY

Like any other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

1. Declare the array
2. Create memory locations
3. Put values into the memory locations.

### Declaration of Arrays

Arrays in Java may be declared in two forms:

*Form1*

type arrayname[ ];

*Form2*

type[ ] arrayname;

Examples:

```
int          number [ ];
float        average [ ];
int [ ]      counter;
float [ ]    marks;
```

Remember, we do not enter the size of the arrays in the declaration.

### Creation of Arrays

After declaring an array, we need to create it in the memory. Java allows us to create arrays using new operator only, as shown below:

arrayname = new type[size];

Examples:

```
number      = new int[5];
average     = new float[10];
```

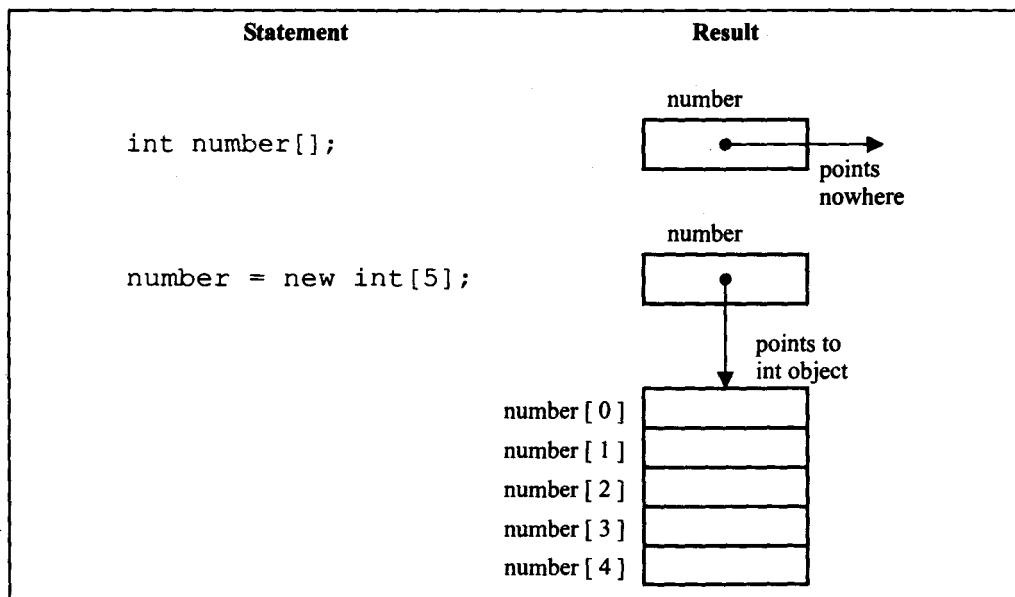
These lines create necessary memory locations for the arrays number and average and designate them as int and float respectively. Now, the variable number refers to an array of 5 integers and average refers to an array of 10 floating point values.

It is also possible to combine the two steps--declaration and creation--into one as shown below:

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

Fig. 9.1 illustrates creation of an array in memory.

**Fig. 9.1**



Creation of an array in memory

### Initialization of Arrays

The final step is to put values into the array created. This process is known as *initialization*. This is done using the array subscripts as shown below.

```
arrayname[subscript] = value ;
```

Example:

```
number [0] = 35;
number[1] = 40;
```

```
.....
.....
number [4] = 19;
```

Note that Java creates arrays starting with a subscript of 0 and ends with a value one less than the size specified.

Unlike C, Java protects arrays from overruns and underruns. Trying to access an array beyond its boundaries will generate an error message.

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

**type arrayname[ ] = {list of values};**

The array initializer is a list of values separated by commas and surrounded by curly braces. Note that no size is given. The compiler allocates enough space for all the elements specified in the list.

Example:

```
int number[ ] = {35, 40, 20, 57, 19};
```

It is possible to assign an array object to another. Example:

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

are valid in Java. Both the arrays will have the same values.

Loops may be used to initialize large size arrays. Example:

```
.....
.....
for(int i = 0; i < 100; i++)
{
    if (i < 50)
        sum[i] = 0.0;
    else
        sum[i] = 1.0;
}
.....
.....
```

The first fifty elements of the array sum are initialized to zero while the remaining are initialized to 1.0

Consider another example as shown below:

```
for(int x = 0; x < 10; x++)
average [x] = (float)x;
```

T11i\$oop initializes the array average to the values 0.0 to 9.0. Program 9.1 illustrates the use of an array for sorting a list of numbers.

Array Length,

In Java, all arrays store the allocated size in a variable named length. We can access the length of the array a using a.length. Example:

```
int aSize = a.length;
```

This information will be useful in the manipulation of arrays when their sizes are not known.

### Program 9.1 Sortln a list of numbers

---

```
class NumberSorting
{
    public static void main (String args[])
    {
        int number[] = { 55, 40, 80, 65, 71 };
        int n = number.length;

        System.out.print("Given      list:      ");
        for (int i = 0; i < n; i++)
        {
            System.out.print("      " + number[i]);
        }

        System.out.println("\n");

        // Sorting begins

        for (int i = 0; i < n; i++)
        {
            for (int j = i+1; j < n; j++)
            {
                if (number[i] < number[j])
                {
                    // Interchange values

                    int temp = number[i];
                    number[i] = number[j];
                    number[j] = temp;
                }
            }
        }

        System.out.print("Sorted      list:      ");
    }
}
```

---

(Continued)

*Program 9.1 (Continued)*


---

```

for (int i = 0; i < n; i++)
{
    System.out.print(" " + number[i]);
}
System.out.println(" ");
}
}

```

---

Program 9.1 displays the following output:

```

Given list : 55 40 80 65 71
Sorted list : 80 71 65 55 40

```



## TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There will be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four salesgirls:

|              | Item1 | Item2 | Item3 |
|--------------|-------|-------|-------|
| Salesgirl #1 | 310   | 275   | 365   |
| Salesgirl #2 | 210   | 190   | 325   |
| Salesgirl #3 | 405   | 235   | 240   |
| Salesgirl #4 | 260   | 300   | 380   |

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as  $v_{ij}$ . Here  $v$  denotes the entire matrix and  $v_{ij}$  refers to the value in the  $i$ th row and  $j$ th column. For example, in the above table  $v_{23}$  refers to the value 325. Java allows us to define such tables of items by using *two-dimensional arrays*. The table discussed above can be represented in Java as

$v[4][3]$

Two dimensional arrays are stored in memory as shown in Fig. 9.2. As with the single dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

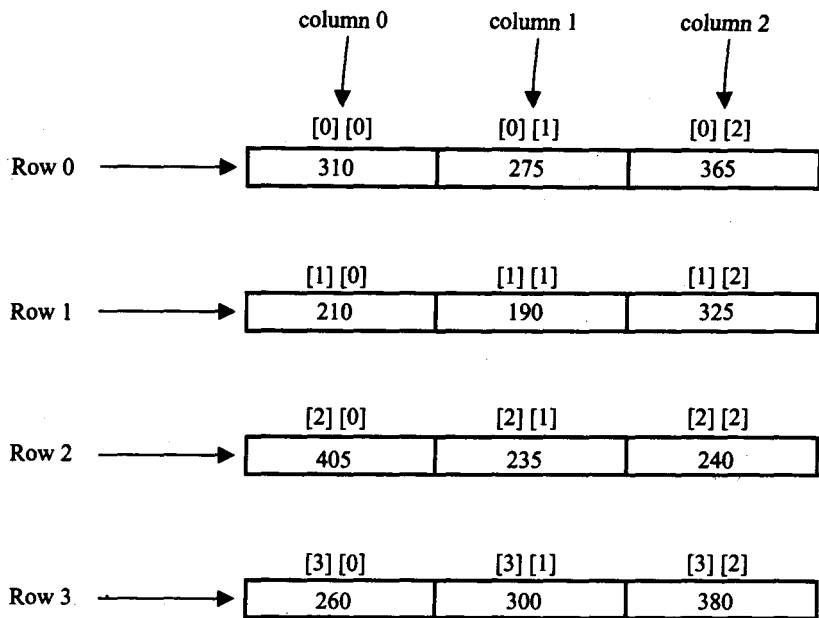
For creating two-dimensional arrays, we must follow the same steps as that of simple arrays. We may create a two-dimensional array like this:

```

int myArray[ ][ ];
myArray = new int[3][4];

```

**Fig 9.2**



**Representation of a two-dimensional array in memory**

or

```
int myArray[ ][ ] = new int[3][4];
```

This creates a table that can store 12 integer values, four across and three down.

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int table[2][3] = {0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
int table[ ][ ] = {{0,0,0}, {1,1,1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table[ ][ ] = {
    {0,0,0},
    {1,1,1}
};
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

We can refer to a value stored in a two-dimensional array by using subscripts for both the column and row of the corresponding element. Example:

```
int value = table[i][2];
```

This retrieves the value stored in the second row and third column of table matrix.

A quick way to initialize a two-dimensional array is to use nested for loops as shown below:

```
for (i = 0; i < 5; i++)
{
    for (j = 0; j < 5; j++)
    {
        if (i == j)
            table[i][j] = 1;
        else
            table[i][j] = 0;
    }
}
```

This will set all the diagonal elements to 1 and others to zero as given below:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 |

Program 9.2 illustrates the use of two-dimensional arrays in real-life situations.

### **Program 9.2 Application of two-dimensional arrays**

---

```
class MulTable
{
    final static int ROWS= 20;
    final static int COLUMNS 20;

    public static void main (String args[])
    {
        int product[][] = new int[ROWS] [COLUMNS];
        int row, column;

        System.out.println("MULTIPLICATION      TABLE-);
        System.out.println("      ");

        int i,j;
        for (i=10; i<ROWS; i++)
        {
```

---

(Continued)

---

**Ph4tunt** 9.2 (*Continued*)

```
for (j=10; j<COLUMNS; j++)
{
    .product [i] [j] = i*j;
    System.out.print("      " +product [i] [j]);
}
System.out.println("      ");
}
}
```

---

Program 9.2 produces the following output:

MULTIPLICATION TABLE

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 100 | 110 | 120 | 130 | 140 | 150 | 160 | 170 | 180 | 190 |
| 110 | 121 | 132 | 143 | 154 | 165 | 176 | 187 | 198 | 209 |
| 120 | 132 | 144 | 156 | 168 | 180 | 192 | 204 | 216 | 228 |
| 130 | 143 | 156 | 169 | 182 | 195 | 208 | 221 | 234 | 247 |
| 140 | 154 | 168 | 182 | 196 | 210 | 224 | 238 | 252 | 266 |
| 150 | 165 | 180 | 195 | 210 | 225 | 240 | 255 | 270 | 285 |
| 160 | 176 | 192 | 208 | 224 | 240 | 256 | 272 | 288 | 304 |
| 170 | 187 | 204 | 221 | 238 | 255 | 272 | 289 | 306 | 323 |
| 180 | 198 | 216 | 234 | 252 | 270 | 288 | 306 | 324 | 342 |
| 190 | 209 | 228 | 247 | 266 | 285 | 304 | 323 | 342 | 361 |

### Variable Size Arrays

Java treats multidimensional arrays as "arrays of arrays". It is possible to declare a two-dimensional array as follows:

```
int xC [] = new int[3] [];
xC [0] = new int[2];
xC [1] = new int[4];
xC [2] = new int[3];
```

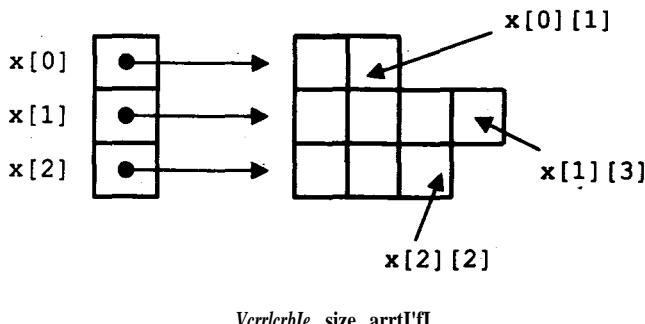
These statements create a two-dimensional array as having different lengths for each row as shown in Fig. 9.3.



### STRINGS

String manipulation is the most common part of many Java programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in Java is by using a character array. Example:

Fig. 9.3



```
char charArray[] = new char[4];
charArray[0] = 'J';
charArray[1] = 'a';
charArray[2] = 'v';
charArray[3] = 'a';
```

Although character arrays have the advantage of being able to query their length, they themselves are not good enough to support the range of operations we may like to perform on strings. For example, copying one character array into another might require a lot of book keeping effort. Fortunately, Java is equipped to handle these situations more efficiently.

In Java, strings are class objects and implemented using two classes, namely, String and StringBuffer. A Java string is an instantiated object of the String class. Java strings, as compared to C strings, are more reliable and predictable. This is basically due to C's lack of bounds-checking. A Java string is not a character array and is not NULLterminated. Strings may be declared and created as follows:

```
String stringName;
stringName = new String ("string");
```

Example:

```
String firstName;
firstName = new String ("Anil");
```

These two statements may be combined as follows:

```
String firstName = new String ("Anil");
```

Like arrays, it is possible to get the length of string using the length method of the String class.

```
int m = firstName.length();
```

Note the use of parentheses here. Java strings can be concatenated using the + operator.  
Examples:

```

String fullName = name1 + name2;
String cityl = "New" + " Delhi";

```

where name1 and name2 are Java strings containing string constants. Another example is:

```
System.out.println(firstName + " Kumar");
```

## String Arrays

We can also create and use arrays that contain strings. The statement

```
String itemArray[ ] = new String[3];
```

will create an itemArray of size 3 to hold three string constants. We can assign the strings to the itemArray element by element using three different statements or more efficiently using a for loop.

## String Methods

The String class defines a number of methods that allow us to accomplish a variety of string manipulation tasks. Table 9.1 lists some of the most commonly used string methods, and their tasks. Program 9.3 shows the use of the method compareTo( ) to sort an array of strings in alphabetical order.

**Table 9.1 Some Most Commonly Used String Methods**

| Method Call                   | Task performed                                                               |
|-------------------------------|------------------------------------------------------------------------------|
| s2 = s1.toLowerCase();        | Converts the string s1 to all lowercase                                      |
| s2 = s1.toUpperCase();        | Converts the string s1 to all Uppercase                                      |
| s2 = s1.replace ('x' , 'y' ); | Replace all appearances of x with y                                          |
| s2 = s1.trim ();              | Remove white spaces at the beginning and end of the string s1                |
| s1.equals (s2)                | Returns 'true' if s1 is equal to s2                                          |
| s1.equalsIgnoreCase(s2)       | Returns 'true' if s1 = s2, ignoring the case of characters                   |
| s1.length ()                  | Gives the length of s1                                                       |
| s1.charAt(n)                  | Gives nth character of s1                                                    |
| s1.compareTo(s2)              | Returns negative if s1 < s2, positive if s1 > s2, and zero if s1 is equal s2 |
| s1.concat(s2)                 | Concatenates s1 and s2                                                       |
| s1.substring (n)              | Gives substring starting from nth character                                  |
| s1.substring(n, m)            | Gives substring starting from nth character up to mth (not including mth)    |
| String.valueOf (p)            | Creates a string object of the parameter p (simple type or object)           |
| p.toString ()                 | Creates a string representation of the object p                              |
| s1.indexOf('x')               | Gives the position of the first occurrence of 'x' in the string s1           |
| s1.indexOf ('x' ,n)           | Gives the position of 'x' that occurs after nth position In the string s1    |
| String.valueOf(Variable)      | Converts the parameter value to string representation                        |

---

### **Program 9.3 *Alphabetical orderIn, of ser/n.,***

---

```
class StringOrdering
{
    static String name[] = {"UMadras", "UDelhi", "uAhmedabad",
                           "uCalcutta", "uBombay"};

    public static void main (String args[])
    {
        int size = name.length;
        String temp = null;

        for (int i = 0; i < size; i++)
        {
            for (int j = i+1; j < size; j++)
            {
                if (name[j] .compareTo(name[i]) < 0)
                {
                    // swap the strings
                    temp = name[i];
                    name[i] = name[j];
                    name[j] = temp;
                }
            }
        }
        for (int i = 0; i < size; i++)
        {
            System.out.println(name[i]);
        }
    }
}
```

---

Program 9.3 produces the following sorted list:

Ahmedabad  
Bombay  
Calcutta  
Delhi  
Madras

### **StringBuffer Class**

StringBuffer is a peer class of String. While String creates strings of fixed\_length, StringBuffer creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to the end. Table 9.2 lists some of the methods that are frequently used in string manipulations.

Table 9.2 Commonly Used StringBuffer Methods

| Method                 | Task                                                                                                            |
|------------------------|-----------------------------------------------------------------------------------------------------------------|
| 51.5.setCharAt(n, 'x') | Modifies the nth character to x                                                                                 |
| 51.append(52)          | Appends the string s2 to s1 at the end                                                                          |
| s1.insert(n, 52)       | Inserts the string s2 at the position n of the string s1                                                        |
| 51.setLength(n)        | Sets the length of the string s1 to n. If n<s1.length() 51 is truncated. If n>51.length() zeros are added to s1 |

Program 9.4 shows how some of the Bstring methods are used for manipulating strings.

#### Program 9.4 Manipulation of strings

```
class StringManipulation
{
    public static void main (String args[ ])
    {
        StringBuffer str = new StringBuffer("Object language");
        System.out.println("Original String :" + str);

        // obtaining string length
        System.out.println("Length of string :" + str.length());

        // Accessing characters in a string
        for (int i = 0; i < str.length(); i++)
        {
            int p = i + 1;
            System.out.println("Character at position: " + p +
                " is " + str.charAt(i));
        }

        // Inserting a string in the middle
        String aString = new String(str.toString());
        int pos = aString.indexOf(" language");
        str.insert(pos, " Oriented");
        System.out.println("Modified string: " + str);

        // Modifying characters
        str.setCharAt(6, 'l');
        System.out.println("String now: " + str);
    }
}
```

(Continued)

**Program 9.4 (Continued)**


---

```
// Appending a string at the end
str.append(" improves security.");
System.out.println("Appended string: " + str);
}
}
```

---

Output of Program 9.4 would be:

```
Original String : Object language
Length of string : 15
Character at position : 1 is o
Character at position : 2 is b
Character at position : 3 is j
Character at position : 4 is e
Character at position : 5 is c
Character at position : 6 is t
Character at position : 7 is s
Character at position : 8 is i
Character at position : 9 is a
Character at position : 10 is n
Character at position : 11 is g
Character at position : 12 is u
Character at position : 13 is a
Character at position : 14 is g
Character at position : 15 is e
Modified string : Object Oriented language
String now : Object-Oriented language
Appended string : Object-Oriented language improves security.
```



## VECTORS

C and C++ programmers will know that generic utility functions with variable arguments can be used to pass different arguments depending upon the calling situations. Java does not support the concept of variable arguments to a function. This feature can be achieved in Java through the use of the `Vector` class contained in the `java.util` package. This class can be used to create a generic dynamic array known as `vector` that can hold *objects of any type and any number*. The objects do not have to be homogenous. Arrays can be easily implemented as vectors. Vectors are created like arrays as follows:

```
Vector intVect = new Vector(); // declaring without size
Vector list = new Vector(3); // declaring with size
```

Note that a vector can be declared without specifying any size explicitly. A vector can accommodate an unknown number of items. Even, when a size is specified, this can be overlooked and a different number of items may be put into the vector. Remember, in contrast, an array must always have its size specified.

Vectors possess a number of advantages over arrays.

1. It is convenient to use vectors to store objects.
2. A vector can be used to store a list of objects that may vary in size.
3. We can add and delete objects from the list as and when required.

A major constraint in using vectors is that we cannot directly store simple data types in a vector; we can only store objects. Therefore, we need to convert simple types to objects. This can be done using the *wrapper classes* discussed in the next section. The vector class supports a number of methods that can be used to manipulate the vectors created. Important ones are listed in Table 9.3.

Table 9.3 Important Vector Methods

| Method Gall                    | Task performed                                          |
|--------------------------------|---------------------------------------------------------|
| list.addElement(item)          | Adds the item specified to the list at the end          |
| list.elementAt(10)             | Gives the name of the 10th object                       |
| list.size()                    | Gives the number of objects present                     |
| list.removeElement(item)       | Removes the specified item from the list                |
| list.removeElementAt(n)        | Removes the item stored in the nth position of the list |
| list.removeAllElements()       | Removes all the elements in the list                    |
| list.copyInto(array)           | Copies all items from list to array                     |
| list.insertElementAt (item, n) | Inserts the item at nth position                        |

Program 9.5 illustrates the use of arrays, strings and vectors. This program converts a string vector into an array of strings and displays the strings.

#### Program 9.5 *Worldn, with vectors and arrays*

```
import java.util.*;                                // Importing Vector class
class LanguageVector
{
    public static void main(String args[])
    {
        Vector list = new Vector();
        int length = args.length;
        for (int i = 0; i < length; i++)
        {
            list.addElement(args[i]);
        }
    }
}
```

(Continued)

---

**Program 9.5 (Continued)**

---

```
list.insertElementAt("COBOL", 2);

int size = list.size();
String lietArray[] = new String[size];

list.copyInto(listArray);

System.out.println("List of Languages");
for (int i = 0; i < size; i++)
{
    System.out.println(listArray[i]);
}
}
```

---

Command line input and output are:

```
C:\JAVA\prog>java LanguageVector Ada BASIC C++ FORTRAN Java

List of Languages
Ada
BASIC
COBOL
C++
FORTRAN
Java
```

**9.7**

## WRAPPER CLASSES

As pointed out earlier, vectors cannot handle primitive data types like int, float, long, char, and double. Primitive data types may be converted into object types by using the wrapper classes contained in the `java.lang` package. Table 9.4 shows the simple data types and their corresponding wrapper class types.

**Table 9.4 Wrapper Classes for Converting Simple Types**

| Simple Type          | Wrapper Class          |
|----------------------|------------------------|
| <code>boolean</code> | <code>Boolean</code>   |
| <code>char</code>    | <code>Character</code> |
| <code>double</code>  | <code>Double</code>    |
| <code>float</code>   | <code>Float</code>     |
| <code>int</code>     | <code>Integer</code>   |
| <code>long</code>    | <code>Long</code>      |

The wrapper classes have a number of unique methods for handling primitive data types and objects. They are listed in the following tables.

**Table 9.5 Converting Primitive Numbers to Object Numbers Using Constructor Methods**

| Constructor Calling                  | Conversion Action                   |
|--------------------------------------|-------------------------------------|
| Integer    IntVal = new Integer(i);  | Primitive integer to Integer object |
| Float    FloatVal = new Float(f);    | Primitive float to Float object     |
| Double    DoubleVal = new Double(d); | Primitive double to Double object   |
| Long    LongVal = new Long(l);       | Primitive long to Long object       |

Note: *i,j,d, and l* are primitive data types denoting int,float, double and long data types.

*They may be constants or variables.*

**Table 9.6 Converting Object Numbers to Primitive Numbers Using typeValue() method**

| Method Calling                          | Conversion Action           |
|-----------------------------------------|-----------------------------|
| int    i = IntVal.intValue( );          | Object to primitive integer |
| float    f = FloatVal.floatValue( );    | Object to primitive float   |
| long    l = LongVal.longValue( );       | Object to primitive long    |
| double    d = DoubleVal.doubleValue( ); | Object to primitive double  |

**Table 9.7 Converting Numbers to Strings Using to String( ) Method**

| Method Calling               | Conversion Action           |
|------------------------------|-----------------------------|
| str    = Integer.toString(i) | Primitive integer to string |
| str    = Float.toString(f);  | Primitive float to string   |
| str    = Double.toString(d); | Primitive double to string  |
| str    = Long.toString(l);   | Primitive long to string    |

**Table 9.8 Converting String Objects to Numeric Objects Using the Static Method ValueOf( )**

| Method Calling                      | Conversion Action                 |
|-------------------------------------|-----------------------------------|
| DoubleVal    = Double.valueOf(str); | Converts string to Double object  |
| FloatVal    = Float.valueOf(str);   | Converts string to Float object   |
| IntVal    = Integer.valueOf(str);   | Converts string to Integer object |
| LongVal    = Long.valueOf(str);     | Converts string to Long object    |

Note: These numeric objects may be converted to primitive numbers using the typeValue() method as shown in Table 9.6.

**Table 9.9 Converting Numeric Strings to Primitive Numbers Using Parsing Methods**

| Method | Calling                        | Conversion                           | Action |
|--------|--------------------------------|--------------------------------------|--------|
|        | int i = Integer.parseInt(str); | Converts string to primitive integer |        |
|        | long l = Long.parseLong(str);  | Converts string to primitive long    |        |

Note: *parseInt()* and *parseLong()* methods throw a *NumberFormatException* if the value of the str does not represent an integer.

Program 9.6 illustrates the use of some most commonly used wrapper class methods.

#### Program 9.6 Use of wrapper class methods

```
import java.io.*;
class Invest
{
    public static void main (String args[])
    {
        Float principalAmount = new Float(0);
        Float interestRate = new Float(0);
        int numYears = 0;

        try
        {
            DataInputStream in = new DataInputStream(System.in);

            System.out.print("Enter Principal Amount : ");
            System.out.flush();
            String principalString = in.readLine();
            principalAmount = Float.valueOf(principalString);

            System.out.print("Enter Interest Rate : ");
            System.out.flush();
            String interestString = in.readLine();
            interestRate = Float.valueOf(interestString);
            System.out.print ("Enter Number of Years : ");
            System.out.flush();
            String yearsString = in.readLine();
            numYears = Integer.parseInt(yearsString);
        }

        catch (IOException e)
        {
            System.out.println("I/O Error");
            System.exit(1);
        }
    }
}
```

(Continued)

---

**Program 9.6 (Continued)**

---

```
float value = loan(principalAmount.floatValue(),
                    interestRate.floatValue(), numYears);
printline ();
System.out.println("\Final Value = " + value);
printline ();
}

// Method to compute Final Value

static float loan(float p, float r, int n)
{
    int year = 1;
    float sum = p;
    while (year <= n)
    {
        sum = sum * (1+r);
        year = year + 1;
    }
    return sum;
}

// Method to draw a line

static void printline()
{
    for (int i = 1; i <= 30; i++)
    {
        System.out.print("\= ");
    }
    System.out.println("\n");
}
}
```

---

The output of Program 9.6 would be:

```
Enter Principal Amount : 5000
Enter Interest Rate : 0.15
Enter Number of Years : 4
=====
Final Value = 8745.03
=====
```



## SUMMARY

In this chapter, we have discussed three important Java data structures namely arrays, strings and vectors. We learned the following:

- What is an array in Java
- How are arrays used
- How does Java handle strings
- How to use the String and StringBuffer classes
- What is a vector in Java
- How to use vectors to store a list of objects that may vary in size
- How are wrapper classes useful

We have also seen some examples to illustrate where these structures can be used in programming.

### KEY TERMS

**Array, Index, Subscript, Elements, String, StringBuffer, Substring, Concatenation, Vector, Wrapper class.**

### REVIEW QUESTIONS

- 9.1 What is an array?
- 9.2 Why are arrays easier to use compared to a bunch of related variables?
- 9.3 Write a statement to declare and instantiate an array to hold marks obtained by students in different subjects in a class. Assume that there are up to 60 students in a class and there are 8 subjects.
- 9.4 Find errors, if any, in the following code segment:

```
int m;
int x[ ] = int[lO];
inf[. ] y = int[lI];
for(m=l; m<=lO; ++m)
    x[m] = y[m] *** m;
x **. y **. new int[20];
for(m=0; m<lO; ++m)
    SYstem.out.println(x[m])
```

- 9.5 Write a program for fitting a straight line through a set of points  $(x_i, y_i)$ ,  $i = 1, \dots, n$ . The straight line equation is

$$y = mx + c$$

and the values of  $m$  and  $c$  are given by

$$m = \frac{n \sum (x_i y_i) - (\sum x_i)(\sum y_i)}{n(\sum x_i^2) - (\sum x_i)^2}$$

$$c = \frac{1}{n} (\sum y_i - m \sum x_i)$$

All summations are from 1 to  $n$ .

- 9.6 The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows:

| Day | City |   |   |       |    |  |  |  |  |  |
|-----|------|---|---|-------|----|--|--|--|--|--|
|     | 1    | 2 | 3 | ..... | 10 |  |  |  |  |  |
| 1   |      |   |   |       |    |  |  |  |  |  |
| 2   |      |   |   |       |    |  |  |  |  |  |
| 3   |      |   |   |       |    |  |  |  |  |  |
| .   |      |   |   |       |    |  |  |  |  |  |
| .   |      |   |   |       |    |  |  |  |  |  |
| 31  |      |   |   |       |    |  |  |  |  |  |

Write a program to read the table elements into a two-dimensional array **temperature**, and to find the city and day corresponding to (a) the highest temperature and (b) the lowest temperature.

- 9.7 An election is contested by 5 candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the votes cast for each candidate using an array variable **count**. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot' and the program should also count the number of spoilt ballots.

- 9.8 The annual examination results of 100 students are tabulated as follows:

| Roll No. | Subject 1 | Subject 2 | Subject 3 |
|----------|-----------|-----------|-----------|
| .        |           |           |           |
| .        |           |           |           |
| .        |           |           |           |
|          |           |           |           |

Write a program to read the data and determine the following:

- (a) Total marks obtained by each student.
- (b) The highest marks in each subject and the Roll No. of the student who secured it.
- (c) The student who obtained the highest total marks.

- 9.9 Given are two one-dimensional arrays A and D which are sorted in ascending order .. Write a program to merge them into a single sorted array C that contains every item from arrays A and D, in ascending order.
- 9.10 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & & & \\ b_{n1} & \dots & \dots & b_{nn} \end{bmatrix}$$

The product of A and B is a third matrix C of size n by n where each element of C is given by the following equation.

$$C_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Write a program that will read the values of elements of A and B and produce the product matrix C.

- 9.11 How does String class differ from the StringBuffer class?
- 9.12 Write a method called

```
delete(String str, int m)
```

that returns the input string with the mth element removed.

- 9.13 Write a program to do the following:
- (a) To output the question "Who is the inventor of C++?"
  - (b) To accept an answer.
  - (c) To print out "Good" and then stop, if the answer is correct.
  - (d) To output the message "try again", if the answer is wrong.
  - (e) To display the correct answer when the answer is wrong even at the third attempt and stop.
- 9.14 Write a program to extract a portion of a character string and print the extracted string. Assume that m characters are extracted, starting with the nth character.
- 9.15 Write a program, which will read a text and count all occurrences of a particular word.
- 9.16 Write a program, which will read a string and reWrite it in the alphabetical order. For example, the word STRING should be written as GINRST.
- 9.17 What is a vector? How is it different from an array?

- 9.18 What are the applications of wrapper classes?
- 9.19 Write a program that accepts a shopping list of five items from the command line and stores them in a vector.
- 9.20 Modify the program of Question 9.19 to accomplish the following:
- To delete an item in the list.
  - To add an item at a specified location in the list.
  - To add an item at the end of the list.
  - To print the contents of the vector:

# **Chapter 10**

## **Interfaces, Multiple Inheritance**



### **INTRODUCTION**

In Chapter 8, we discussed about classes and how they can be inherited by other classes. We also learned about various forms of inheritance and pointed out that Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like

```
class A extends B extends C
{
    .....
}
```

is not permitted in Java. However, the designer\$pf Java could not overlook the importance of multiple inheritance. A large number of reid-life applications require the use of mulUple inheritance whereby we inherit methods and properties from -several distinct classes. Since C+ + like implementation of multiple inheritance proves difficult and adds complexity to the language, Java provides an alternate approach known as *interfaces* to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.



### **DEFINING INTERFACES**

An interface is basically a kind of class. Uke classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants.

Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is:

```
interface InterfaceName  
{  
    variables declaration;  
    methods declaration;  
}
```

Here, **interface** is the key word and *InterfaceName* is any valid Java variable (just like class names). Variables are declared as follows:

```
static final type VariableName = Value;
```

Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example:

```
return-type methodName1 (parameter_list);
```

Here is an example of an interface definition that contains two variables and one method

```
interface Item  
{  
    static final int code = 1001;  
    static final String name = "Fan";  
    void display ();  
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon. The class that implements this interface must define the code for the method.

Another example of an interface is:

```
interface Area  
{  
    final static float pi = 3.142F;  
    float compute (float x, float y);  
    void show ();  
}
```

## EXTENDING INTERFACES

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword **extends** as shown below:

```
interface name2 extends name1
{
    body of name2
}
```

For example, we can put all the constants in one interface and the methods in the other. This will enable us to use the constants in classes where the methods are not required. Example:

```
interface ItemConstants
{
    int code = 1001;
    string name = "Fan";
}
interface Item extends ItemConstants
{
    void display();
}
```

The interface Item would inherit both the constants code and name into it. Note that the variables name and code are declared like simple variables. It is allowed because all the variables in an interface are treated as constants although the keywords final and static are not present.

We can also combine several interfaces together into a single interface. Following declarations are valid:

---

```
interface ItemConstants
{
    int code = 1001;
    String name = "Fan";
}
interface ItemMethods
{
    void display();
}
interface Item extends ItemConstants, ItemMethods
{
    .....
    .....
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. Instead, it is the responsibility of any class that implements the derived interface to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas.

It is important to remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract methods and constants.



## 10.4 IMPLEMENTING INTERFACES

Interfaces are used as “superclasses” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname implements interfacename
{
    body of classname
}
```

Here the class *classname* “implements” the interface *interfacename*. A more general form of implementation may look like this:

```
class classname extends superclass
    implements interface1,interface2, .....
{
    body of classname
}
```

This shows that a class can extend another class while implementing interfaces.

When a class implements more than one interface, they are separated by a comma. The implementation of interfaces can take various forms as illustrated in Fig. 10.1.

Implementation of interfaces as class types is illustrated by Program 10.1. In this program, first we create an interface Area and implement the same in two different classes, Rectangle and Circle. We create an instance of each class using the new operator. Then we declare an object of type Area, the interfaceclass. Now, we assign the reference to the Rectangle object rect to area. When we call the compute method of area, the compute method of Rectangle class is invoked. We repeat the same thing with the Circle object.

---

### Program 10.1 *ImplementIn*, Interfaces

---

```
// InterfaceTestJava
interf.ceArea
{
    final static float pi = 3.14F;
```

*// Interface defined*

(Continued)

## Program 10.1 (Continued)

---

```

    float compute (float x, float y);
}

class Rectangle implements Area           // Interface implemented
{
    public float compute (float x, float y)
    {
        return (x*y) ;
    }
}

class Circle implements Area             // Another implementation
{
    public float compute (float x, float y)
    {
        return (pi *x*x) ;
    }
}

class InterfaceTest
{
    public static void main (String args[])
    {
        Rectangle rect = new Rectangle();
        Circle cir = new Circle();

        Area area;                         // Interface object

        area = rect;
        System.out.println("Area      of Rectangle = "
                           + area.compute(10,20));

        area = cir;
        System.out.println("Area      of Circle   = "
                           + area.compute(10,0));
    }
}

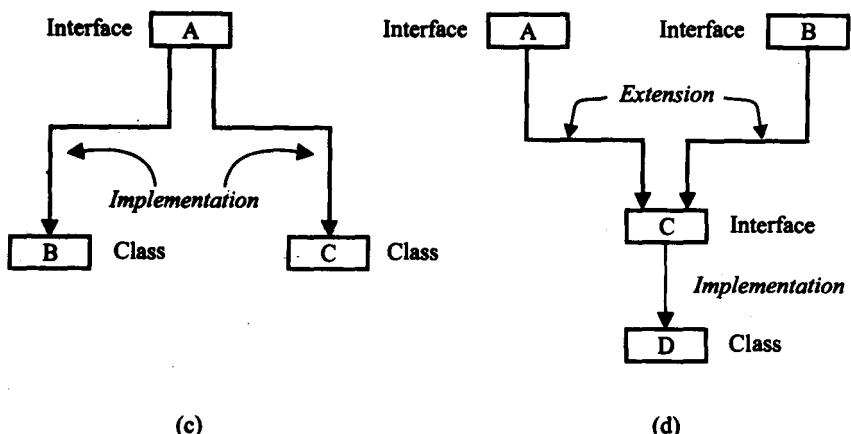
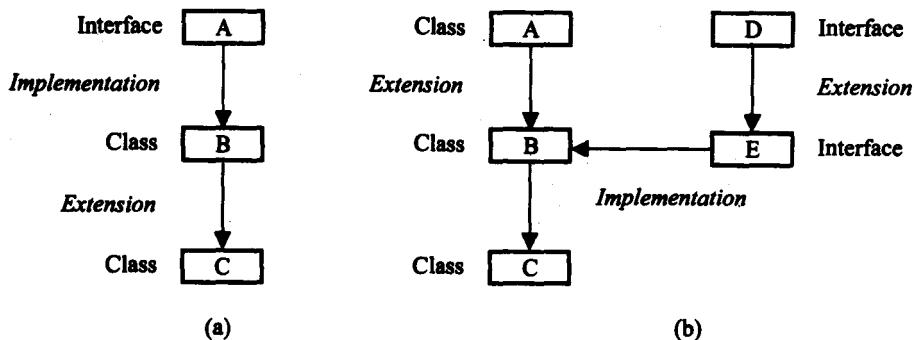
```

---

The Output is as follows:

Area of Rectangle = 200  
Area of Circle = 314

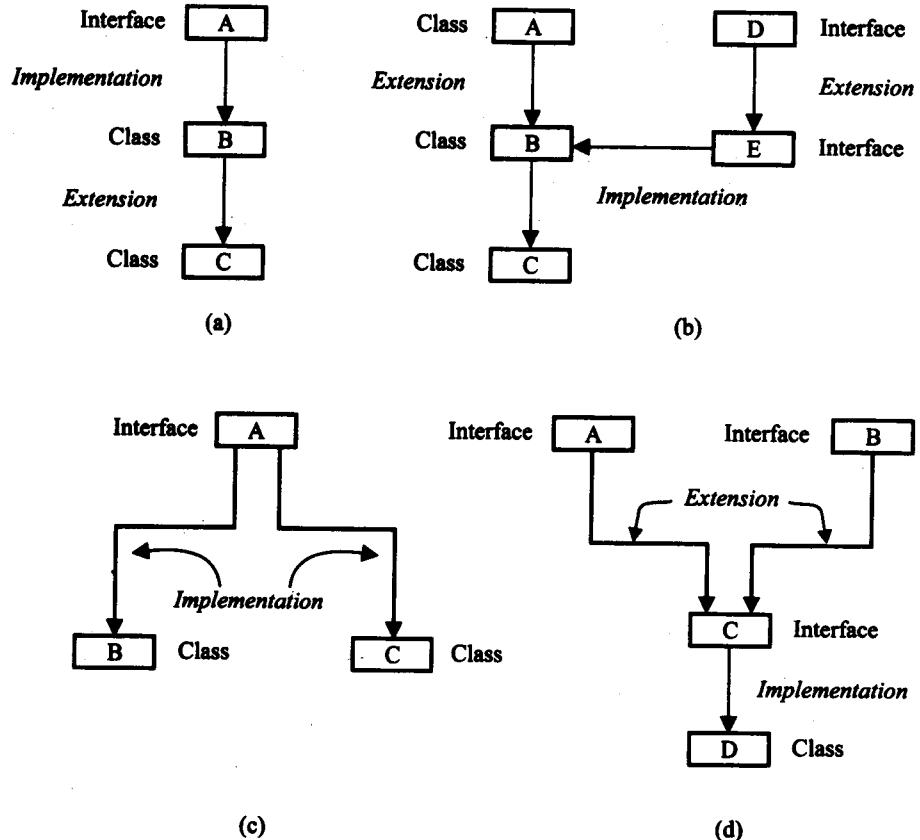
Any number of dissimilar classes can implement an interface. However,to implement the methods, we need to refer to the class objects as types of the interface rather than types of their respective classes. Note that if a class that implements an interface does not implement all the methods of the interface, then the class becomes an *abstract* class and cannot be instantiated.

**Fig. 10.1****Various forms of interface implementation****10.5****ACCESSING INTERFACE VARIABLES**

Interfaces can be used to declare a set of constants that can be used in different classes. This is similar to creating header files in C++ to contain a large number of constants. Since such interfaces do not contain methods, there is no need to worry about implementing any methods. The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value. Example:

```
interface A
{
    int m = 10;
    int n = 50;
}
class B implements A
```

**Fig. 10.1**



## **Various forms of interface implementation**

105

# ACCESSING INTERFACE VARIABLES

Interfaces can be used to declare a set of constants that can be used in different classes. This is similar to creating header files in C++ to contain a large number of constants. Since such interfaces do not contain methods, there is no need to worry about implementing any methods. The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value. Example:

```
interface A
{
    int m = 10;
    int n = 50;
}
class B implements A
```

```

{
    int x = m ;
    void methodB(int size)
    {
        .....
        .....
        if (size < n)
        .....
    }
}

```

Program 10.2 illustrates the implementation of the concept of multiple inheritance using interfaces.

### ***Program 10.2 Implementing multiple inheritance***

---

```

class Student
{
    int rollNumber;

    void getNumber(int n)
    {
        rollNumber = n;
    }

    void putNumber( )
    {
        System.out.println(~ Roll No : " + rollNumber);
    }
}

class Test extends Student
{
    float part1, part2;

    void getMarks(float ml, float .m2)
    {
        part1 = ml;
        part2 = in2;
    }
    void putMarks( )
    {
        System.out.println(Marks obtained -);
        System.out.println(Part1 a - + part1);
        System.out.println(Part2 - - + part2);
    }
}

```

---

(Continued)

---

..... • 10.2 (Continued)

---

```
    }
    interface Sports
    {
        float sportWt = 6.0F;
        void putWt();
    }

    class Results extends Test implements Sports
    {
        float total;
        public void putWt()
        {
            System.out.println("Sports Wt = " + sportWt);
        }

        void display()
        {
            total = part1 + part2 + sportWt;
            putNumber();
            putMarks();
            putWt();
            System.out.println("Total score = " + total);
        }
    }

    class Hybrid
    {
        public static void main(String args[])
        {
            Results student1 = new Results();
            student1.getNumber(1234);
            student1.getMarks(27.5F, 33.0F);
            student1.display();
        }
    }
}
```

---

Output of the Program 10.2:

```
Roll No : 1234
Marks obtained
part1 = 27.5
Part2 = 33
Sports Wt = 6
Total score = 66.5
```

10.6

## SUMMARY

Java does not support multiple inheritance. Since multiple inheritance is an important concept in OOP paradigm, Java provides an alternate way of implementing this concept. We have discussed in this chapter

- How to design an interface
  - How to extend one interface by the other
  - How to inherit an interface and
  - How to implement the concept of multiple inheritance using interfaces

The concepts discussed in this chapter will enable us to build classes using other classes available already.

## KEY TERMS

## **REVIEW QUESTIONS**

- 10.1 What is an interface?
  - 10.2 How do we tell Java that the class we are creating implements a particular interface?
  - 10.3 What is the major difference between an interface and a class?
  - 10.4 What are the similarities between interfaces and classes?
  - 10.5 Describe the various forms of implementing interfaces. Give examples of Java code for each case.
  - 10.6 Give an example where interface can be used to support multiple inheritance. Develop a standalone Java program for the example.

## **Chapter 11**

# **Packages: Putting Classes Together**



### **INTRODUCTION**

We have repeatedly stated that one of the main features of OOP is its ability to reuse the code already created. One way of achieving this is by extending the classes and implementing the interfaces we had created as discussed in Chapters 8 and 10. This is limited to reusing the classes within a program. What if we need to use classes from other programs without physically copying them into the program under development? This can be accomplished in Java by using what is known as *packages*, a concept similar to "class libraries" in other languages. Another way of achieving the reusability in Java, therefore, is to use packages ..

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes int<;> packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design ..

For most applications, we will need to use two different sets of classes, one for the internal representation of our program's data, and the other for external presentation purposes. We may have to build our own classes for handling our data and use existing class libraries for designing user interfaces. Java packages are therefore classified into two types. The first category is known as *Java system packages* and the second is known as *user defined packages*.

We shall consider both the categories of packages in this chapter and illustrate how to use them in our programs.

## SYSTEM PACKAGES

Java system provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java system. Figure 11.1 shows the functional breakdown of packages that are frequently used in the programs. Table 11.1 shows the classes that belong to each package.

**Fig. 11.1**

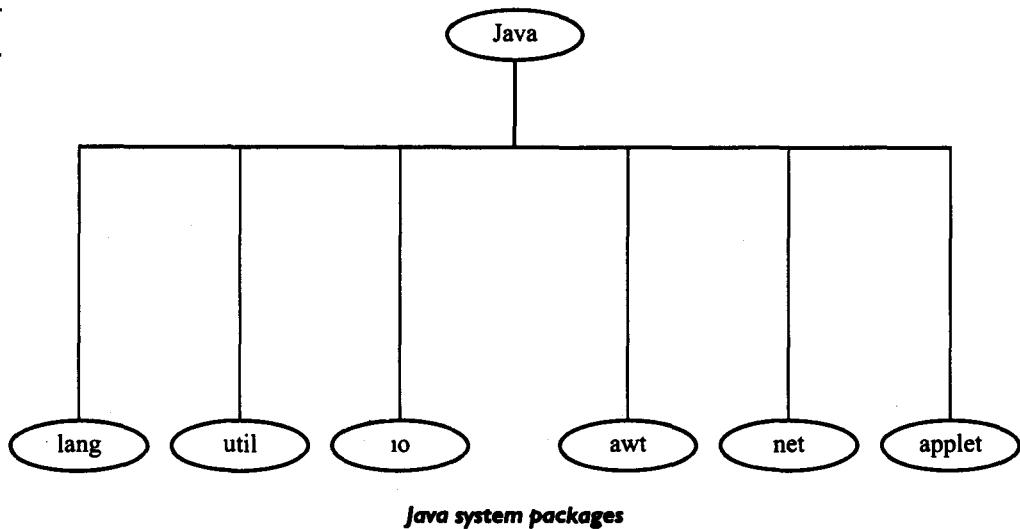


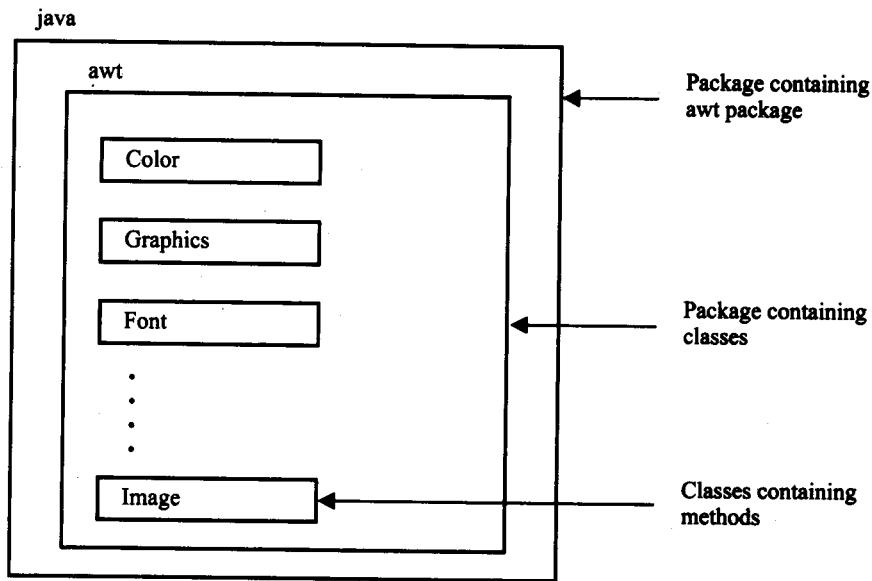
Table 11.1 Java System Packages and Their Classes

| Package name | Contents                                                                                                                                                                                                             |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| java.lang    | Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions. |
| java.util    | Language utility classes such as arrays, hash tables, random numbers, date, etc.                                                                                                                                     |
| java.io      | Input/output support classes. They provide facilities for the input and output of data.                                                                                                                              |
| java.awt     | Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.                                                                                         |
| java.net     | Classes for networking. They include classes for communicating with local computers as well as with internet servers.                                                                                                |
| java.applet  | Classes for creating and implementing applets.                                                                                                                                                                       |

## USING SYSTEM PACKAGES

The packages are organised in a hierarchical structure as illustrated in Fig. 11.2. This shows that the package named java contains the package awt, which in turn contains various classes required for implementing graphical user interface.

**Fig. 11.2**



Hierarchy represents class df JtMothwt #Hclm,..

There are two ways of accessing the classes stored in a package. The first approach is to use *the fully qualified class name* of the class that we want to use. This is done by using the package name containing the class and then appending the class name to it using the dot operator. For example, if we want to refer to the class Color in the awt package, then we may do so as follows:

```
java.awt.Colour
```

Notice that awt is a package within the package java and the hierarchy is represented by separating the levels with dots. This approach is perhaps the best and easiest one if we need to access the class only once or when we need not have to access any other classes of the package.

But, in many situations, we might want to use a class in a number of places in the program or we may like to use many of the classes contained in a package. We may achieve this easily as follows:

```
import packagename.classname;  
or  
import packagename.*;
```

These are known as *import statements* and must appear at the top of the file, before any class declarations.

The first statement allows the specified class in the specified package to be imported. For example, the statement

```
import java.awt.Color;
```

imports the class Colour and therefore the class name can now be directly used in the program. There is no need to use the package name to qualify the class.

The second statement imports every class contained in the specified package. For example, the statement

```
import java.awt.*;
```

will bring all classes of java.awt package.



## NAMING CONVENTIONS

Packages can be named using the standard Java naming rules. By convention, however, packages begin with lowercase letters. This makes it easy for users to distinguish package names from class names when looking at an explicit reference to a class. We know that all class names, again by convention, begin with an uppercase letter. For example, look at the following statement:

```
double y = java.lang.Math.sqrt(x);
```

A diagram illustrating the decomposition of a fully qualified class name. The code 'double y = java.lang.Math.sqrt(x);' is shown. Three vertical arrows point upwards from the words 'package name', 'class name', and 'method name' to the corresponding parts of the class name 'java.lang.Math.sqrt'. The 'package name' arrow points to 'java.lang', the 'class name' arrow points to 'Math', and the 'method name' arrow points to 'sqrt'.

This statement uses a fully qualified class name Math to invoke the method sqrt(). Note that methods begin with lowercase letters. Consider another example:

```
java.awt.Point pts[ ];
```

This statement declares an array of Point type objects using the fully qualified class name.

Every package name must be unique to make the best use of packages. Duplicate names will cause run-time errors. Since multiple users work on Internet, duplicate package names are unavoidable. Java designers have recognised this problem and therefore suggested a package naming convention that ensures uniqueness. This suggests the use of domain names as prefix to the preferred package names. For example:

```
cbe.psg.mypackage
```

Here cbe denotes city name and psg denotes organisation name. Remember that we can create a hierarchy of packages within packages by separating levels with dots.



## CREATING PACKAGES

We have seen in detail how Java system packages are organised and used. Now, let us see how to create our own packages. We must first declare the name of the package using the package keyword followed by a package name. This must be the first statement in a Java source file (except for comments and white spaces). Then we define a class, just as we normally define a class. Here is an example:

---

```
package firstPackage;           // package declaration
public class FirstClass         // class definition
{
    .....
        ..... (body of class)
    .....
}
```

---

Here the package name is firstPackage. The class FirstClass is now considered a part of this package. This listing would be saved as a file called FirstClass.java, and located in a directory named firstPackage. When the source file is compiled, Java will create a .class file and store it in the same directory.

Remember that the .class files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.

To recap, creating our own package involves the following steps:

1. Declare the package at the beginning of a file using the form

**package packagename;**

2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates .class file in the subdirectory.

Remember that case is significant and therefore the subdirectory name must match the package name exactly.

As pointed out earlier, Java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots. Example:

```
package firstPackage.secondPackage;
```

This approach allows us to group related classes into a package and then group related packages into a larger package. Remember to store this package in a subdirectory named EirstPackage \secondPackage.

A Java package file can have more than one class definitions. In such cases, only one of the classes may be declared public and that class name with .java extension is the source file name. When a source file with more than one class definition is compiled, Java creates independent .class files for those classes.



## ACCESSING A PACKAGE

It will be recalled that we have discussed earlier that a Java system package can be accessed either using a fully qualified class name or using a shortcut approach through the import statement. We use the import statement when there are many references to a particular package or the package name is too long and unwieldy.

The same approaches can be used to access the user-defined packages as well. The import statement can be used to search a list of packages for a particular class. The general form of import statement for searching a class is as follows:

```
import package1 [.package2] [.package3].classname;
```

Here *packagename1* is the name of the top level package, *packagename2* is the name of the package that is inside the *packagename1*, and so on. We can have any number of packages in a package hierarchy. Finally, the explicit *classname* is specified.

Note that the statement must end with a semicolon C;). The import statement should appear before any class definitions in a source file. Multiple import statements are allowed. The following is an example of importing a particular class:

```
import firstPackage.secondPackage.MyClass;
```

After defining this statement, all the members of the class MyClass can be directly accessed using the class name or its objects (as the case may be) directly without using the package name.

We can also use another approach as follows:

```
import packagename.*;
```

Here, *packagename* may denote a single package or a hierarchy of packages as mentioned earlier. The star (\*) indicates that the compiler should search this entire package hierarchy when it encounters a class name. This implies that we can access all classes contained in the above package directly.

The major drawback of the shortcut approach is that it is difficult to determine from which package a particular member came. This is particularly true when a large number of packages are imported. But the advantage is that we need not have to use long package names repeatedly in the program.



## USING A PACKAGE

Let us now consider some simple programs that will use classes from other packages. The listing below shows a package named package1 containing a single class ClassA.

```
package package1;

public class ClassA
{
    public void displayA( )
    {
        System.out.println("Class      A");
    }
}
```

---

This source file should be named ClassA.java and stored in the subdirectory package1 as stated earlier. Now compile this java file. The resultant ClassA.class will be stored in the same subdirectory.

Now consider the listing shown below:

---

```
import package1.ClassA;

class PackageTest1
{
    public static void main(String args[ ] )
    {
        ClassA objectA = new ClassA ( );
        objectA.displayA( );
    }
}
```

---

This listing shows a simple program that imports the class ClassA from the package package1. The source file should be saved as PackageTest1.java and then compiled. The source file and the compiled file would be saved in the directory of which package1 was a subdirectory. Now we can run the program and obtain the results.

During the compilation of PackageTest1.java the compiler checks for the file ClassA.class in the package1 directory for information it needs, but it does not actually include the code from ClassA.class in the file PackageTest1.class. When the PackageTest1 program is run, Java looks for the file PackageTest1.class and loads it using something called *class* loader. Now the interpreter knows that it also needs the code in the file ClassA.class and loads it as well.

Now let us consider another package named package2 containing again a single class as shown below:

---

```
package package2;
public class ClassB
{
    protected int m = 10
    public void displayB( )
    {
```

```

        System.out.println("Class      B");
        System.out.println("m      = " + m);
    }
}

```

---

As usual, the source file and the compiled file of this package are located in the subdirectory package2.

Program 11.1 shown below uses classes contained in both the packages and therefore it imports package 1 and package2. Note that we have used star instead of explicit class name in importing package2.

### **'"** **Program 11.1 Import'ng classes from other packages**

---

```

import package1.ClassA;
import package2.*;

class PackageTest2
{
    public static void main (String args [ ])
    {
        ClassA objectA=      new ClassA ( ) ;
        ClassB objectB   =  new ClassB ( );
        objectA.displayA( );
        objectB.displayB( );
    }
}

```

---

This program may be saved as PackageTest2.java, compiled and run to obtain the results. The output will be as under

```

Class A
Class B
m= 10

```

When we import multiple packages it is likely that two or more packages contain classes with identical names. Example:

---

```

package pack1;

public class Teacher
{ ....., }

public class Student
{ ....., }

```

```
package pack2;  
  
public class Courses  
{ ..... }  
  
public class Student  
{ ..... }
```

---

We may import and use these packages like:

```
import pack1.*;  
import pack2.*;  
Student student1;           II create a student object
```

Since both the packages contain the class Student, compiler cannot understand which one to use and therefore generates an error. In such instance, we have to be more explicit about which one we intend to use.

Example:

---

```
import pack1.*;  
import pack2.*;  
pack1.Student student1;      II OK  
pack2.Student student2;      II OK  
Teacher teacher1;           II No problem  
Courses course1;            II No problem
```

---

It is also possible to subclass a class that has been imported from another package. This is illustrated by Program 11.2 The output will be:

```
Class B  
m = 10  
Class C  
m = 10  
n = 20
```

Note that the variable m has been declared as protected. A subclass in another package CaQ inherit a protected member. It would not have been possible if it has been declared as either private or "default"

---

### Program 11.2 Subclassing an Imported class

---

```
II PackageTest3Java  
  
import package2.ClassB;  
  
class ClassC extends ClassB
```

---

(Continued)

## Program 11.2 (Continued)

---

```

{
    int n = 20;
    void displayC( )
    {
        System.out.println("Class      C");
        System.out.println("m      = " + m);
        System.out.println("n      = " + n);
    }
}

class PackageTest3
{
    public static void main (String args[ ])
    {
        ClassC objectC = new ClassC();
        objectC.displayB();
        objectC.displayC();
    }
}

```

---

While using packages and inheritance in a program, we should be aware of the visibility restrictions imposed by various access protection modifiers. As pointed out earlier, packages act

**Table 11.2 Access Protection**

| Access location                  | Access modifier | public | protected | friendly (default) | private protected | private |
|----------------------------------|-----------------|--------|-----------|--------------------|-------------------|---------|
| Same class                       | →               | Yes    | Yes       | Yes                | Yes               | Yes     |
| Subclass in same package         |                 | Yes    | Yes       | Yes                | Yes               | No      |
| Other classes in same package    |                 | Yes    | Yes       | Yes                | No                | No      |
| Subclass in other packages       |                 | Yes    | Yes       | No                 | Yes               | No      |
| Non-subclasses in other packages |                 | Yes    | No        | No                 | No                | No      |

— containers for classes and other packages, and classes act as containers for data and methods. Data members and methods can be declared with the access protection modifiers such as private, protected, and public as well as "default". The effect of use of these modifiers was discussed in detail in Chapter 8. For the sake of easy reference, the access protection details given in Table 8.1 are reproduced Table 11.2~



## 11.8 ADDING A CLASS TO A PACKAGE

It is simple to add a class to an existing package. Consider the following package:

---

```
pac)cage pI;  
public  ClassA  
{  
    // body of A  
}
```

---

The package pI contains one public class by name A. Suppose we want to add another class B to this package. This can be done as follows:

1. Define the class and make it public.
2. Place the package statement

```
package  pI;
```

before the class definition as follows:

```
package  pI;  
public  class  B  
{  
    // bodyofB  
}
```

3. Store this as B.java file under the directory pl.
4. Compile B.java file. This will create a B.cl88s file and place it in the directory pl.

Note that we can also add a non-public class to a package using the same procedure.

Now, the package pI will contain both the classes A and B. A statement like

```
import  pl.*;
```

will import both of them.

Remember that, since a Java source file can have only one class declared as public, we cannot put two or more public classes together in a .java file. This is because of the restriction that the file name should be same as the name of the public class With.java extension.

If we want to create a package with multiple public classes in it, we may follow the following steps:

1. Decide the name of the package.
2. Create a subdirectory with this name under the directory where main source files are stored.
3. Create classes that are to be placed in the package in separate source files and declare the package statement

```
package packagename;
```

at the top of each source file.

4. Switch to the subdirectory created earlier and compile each source file. When completed, the package would contain .class files of all the source files.



## HIDING CLASSES

When we import a package using asterisk (\*), all public classes are imported. However, we may prefer to "not import" certain classes. That is, we may like to hide these classes from accessing from outside of the package. Such classes should be declared "not public". Example:

---

```
package pl;

public class X           // public class, available outside
{
    // body of X
}

class Y                 // not public, hidden
{
    // body of Y
}
```

---

Here, the class Y which is not declared public is hidden from outside of the package pl. This class can be seen and used only by other classes in the same package. Note that a Java source file should contain only one public class and may include any number of non-public classes. We may also add a single non-public class using the procedure suggested in the previous section.

Now, consider the following code, which imports the package pl that contains classes X and Y:

---

```
import pl.*;
X objectX;           // OK; class X is available here
Y objectY;           // Not OK; Y is not available
```

---

Java compiler would generate an error message for this code because the class Y, which has not been declared public, is not imported and therefore not available for creating its objects.



## SUMMARY

In this chapter we saw the building blocks of coding in Java and high-level requirements for designing applets and application programs. Java has several levels of hierarchy for code organisation, the highest of which is the package. We have seen here

- How to create a package,
- How to add more classes to a package,
- How to access the contents of a package,
- How to protect a class from accidental access, and
- How to use Java system packages.

This chapter essentially has shown us how to organise classes into packages to better keep track of them.

### KEY TERMS

**Package, Import, public, protected, Friendly.**

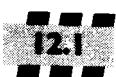
### REVIEW QUESTIONS

---

- 11.1 What is a package?
- 11.2 How do we tell Java that we want to use a particular package in a file?
- 11.3 How do we design a package? .
- 11.4 How do we add a class or an interface to a package?
- 11.5 Consider the example Program 10.2. Design a package to contain the class student and another package to contain the interface sports. Rewrite the Program 10.2 using these packages.
- 11.6 Discuss the various levels of access protection available for packages and their implications.

## **Chapter 12**

# **Multithreaded Programming**



### **INTRODUCTION**

Those who are familiar with the modern operating systems such as Windows 95 may recognize that they can execute several programs simultaneously. This ability is known as multitasking. In system's terminology, it is called *multithreading*.

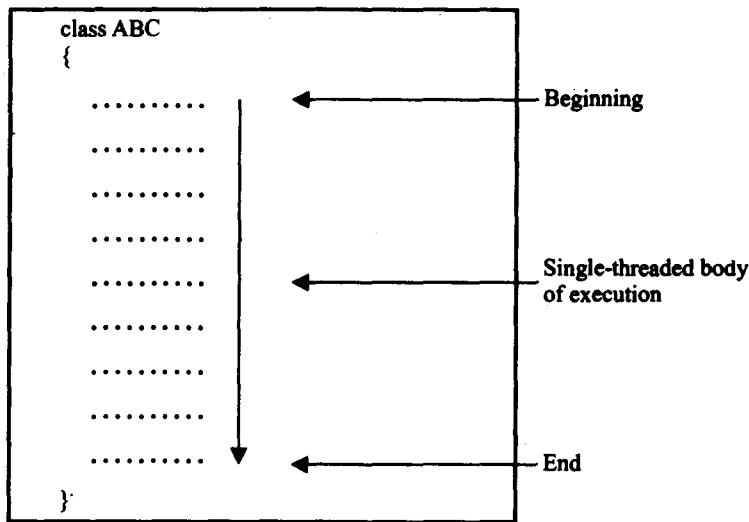
Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

In most of our computers, we have only a single processor and therefore, in reality, the processor is doing only one thing at a time. However, the processor switches between the processes so fast that it appears to human beings that all of them are being done simultaneously.

Java programs that we have seen and discussed so far contain only a single sequential flow of control. This is what happens when we execute a normal ~~~~~~. The program begins, runs through a sequence of executions, and finally ends. At any given point of time, there is only one statement under execution.

A *thread* is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially. In fact, all main programs in our earlier examples can be called *single-threaded* programs. Every program will have at least one thread as shown in Fig. 12.1.

A unique property of Java is its support for multithreading. That is, Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program (or module) known as a *thread* that runs in parallel to others as shown in Fig. 12.2. A program that contains multiple flows of control is known as *multithreaded program*. Figure 12.2 illustrates a Java program with four threads, one main and three others. The main

**Fig. 12.1**

SIn"...threaded fH'OF'Gm

thread is actually the main method module, which is designed to create and start the other three threads, namely A, B and C.

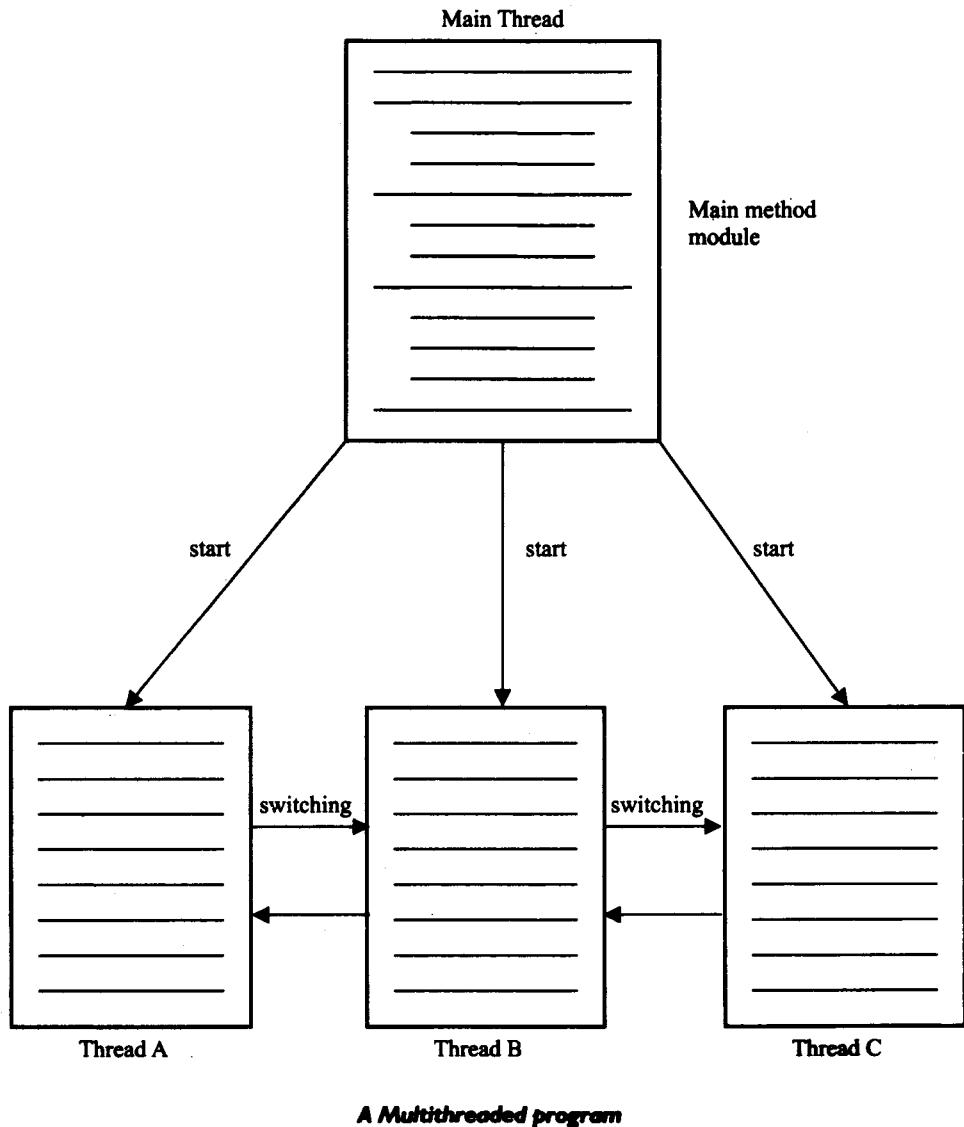
Once initiated by the main thread, the threads A, B, and C run concurrently and share the resources jointly. It is like people living in joint families and sharing certain resources among all of them. The ability of a language to support multi threads is referred to as *concurrency*. Since threads in Java are subprograms of a main application program and share the same memory space, they are known as *lightweight threads* or *lightweight processes*.

It is important to remember that 'threads running in parallel' does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Multithreading is a powerful programming tool that makes Java distinctly different from its fellow programming languages. Multithreading is useful in a number of ways. It enables programmers to do multiple things at one time. They can divide a long program (containing operations that are conceptually concurrent) into threads and execute them in parallel. For example, we can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.

Threads are extensively used in Java-enabled browsers such as HotJava. These browsers can download a file to the local computer, display a Web page in the window, output another Web page to a printer and so on.

Any application we are working on that requires two or more things to be done at the same time is probably a best one for use of threads.

**Fig. 12.2****12.2****CREATING THREADS**

Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method called `run()`. The `run()` method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behaviour can be implemented. A typical `run( )` would appear as follows:

```

public void run( )
{
    .....
    ..... (statements for implementing thread)
    .....
}

```

The run( ) method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called start( ).

A new thread can be created in two ways.

1. *By creating a thread class:* Define a class that extends Thread class and override its run() method with the code required by the thread.
2. *By converting a class to a thread:* Define a class that implements Runnable interface. The Runnable interface has only one method, run(), that is to be defined in the method with the code to be executed by the thread.

The approach to be used depends on what the class we are creating requires. If it requires to extend another class, then we have no choice but to implement the Runnable interface, since Java classes cannot have two superclasses.



## EXTENDING THE THREAD CLASS

We can make our class runnable as a thread by extending the class `java.lang.Thread`. This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the Thread class.
2. Implement the run( ) method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the start() method to initiate the thread execution.

### Declaring the Class

The Thread class can be extended as follows:

```

class MyThread extends Thread
{
    .....
    .
    .
}

```

Now we have a new type of thread `MyThread`.

## **Implementing the *run()* Method**

The run() method has been inherited by the class MyThread. We have to override this method in order to implement the code to be executed by our thread. The basic implementation of run( ) will look like this:

```
public void run( )
{
    .....
    ..... // Thread code here
    .....
}
```

When we start the new thread, Java calls the thread's run ( ) method, so it is the run ( ) where all the action takes place.

## **Starting New Thread**

To actually create and run an instance of our thread class, we must write the following:

```
MyThread aThread = new MyThread();
aThread.start(); // invokes run() method
```

The first line instantiates a new object of class MyThread. Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a *newborn* state.

The second line calls the start () method causing the thread to move into the *Runnable* state. Then, the Java runtime will schedule the thread to run by invoking its run () method. Now, the thread is said to be in the *running* state.

## **An Example of Using the Thread Class**

Program 12.1 illustrates the use of Thread class for creating and running threads in an application. The program creates three threads A, B, and C for undertaking three different tasks. The main method in the ThreadTest class also constitutes another thread which we may call the "main thread".

The main thread dies at the end of its main method. However, before it dies, it creates and starts all the three threads A, B, and C. Note the statements like

```
new < >.start();
```

in the main thread. This is just a compact way of starting a thread. This is equivalent to:

```
A threadA = new A();
threadA.start();
```

Immediately after the thread A is started, there will be two threads running in the program: the main thread and the thread A. The start( ) method returns back to the main thread immediately after invoking the run( ) method, thus allowing the main thread to start the thread B.

### **Program 12.1 Creating threads using the thread class**

---

```
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("\tFrom Thread A : i = " + i);
        }
        System.out.println("Exit from A");
    }
}

class B extends Thread
{
    public void run()
    {
        for(int j=1; j<=5; j++)
        {
            System.out.println("\tFrom Thread B : j = " + j);
        }
        System.out.println("Exit from B");
    }
}

class C extends Thread
{
    public void run()
    {
        for(int k=1; k<=5; k++)
        {
            System.out.println("\tFrom Thread C : k = " + k);
        }
        System.out.println("Exit from C");
    }
}

class ThreadTest
{
    public static void main (String args[])
    {
        new A().start();
        new B().start();
        new C().start();
    }
}
```

Output of Program 12.1 would be:

*First run*

```
From Thread A : i = 1
From Thread A : i = 2
From Thread B : j = 1
From Thread B : j = 2
From Thread C : k= 1-
From Thread C : k = 2
From Thread A : i = 3
From Thread A : i = 4
From Thread B : j = 3
From Thread B : j = 4
From Thread C : k = 3
From Thread C : k = 4
From Thread A : i = 5
Exit from A
From Thread B : j = 5
Exit from B
From Thread C : k = 5
Exit from C
```

*Second run*

```
From Thread A : i = 1
From Thread A : i = 2
From Thread C : k = 1
From Thread C : k = 2
From Thread A : i = 3
From Thread A : i = 4
From Thread B : j = 1
From Thread B : j = 2
From Thread C : k = 3
From Thread C : k = 4
From Thread A : i = 5
Exit from A
From Thread B : k = 4
From Thread B : j = 5
From Thread C : k = 5
Exit from C
From Thread B : j = 5
Exit from B-
```

Similarly, it starts C thread. By the time the main thread has reached the end of its main method, there are a total of four separate threads running in parallel.

We have simply initiated three new threads and started them. We did not hold on to them any further. They are running concurrently on their own. Note that the output from the threads are not specially sequential. They do not follow any specific order. They are running independently of one another and each executes whenever it has a chance. Remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements. Note that a second run has a different output sequence.



## STOPPING AND BLOCKING A THREAD

### Stop pin, a Thread

Whenever we want to stop a thread from running further, we may do so by calling its `stop()` method, like:

```
aThread.stop();
```

This statement causes the thread to move to the *dead* state. A thread will also move to the dead state automatically when it reaches the end of its method. The `stop()` method maybe used when the *premature death* of a thread is desired.

### Blockin, a Thread

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

|                         |                                           |
|-------------------------|-------------------------------------------|
| <code>sleep( )</code>   | // blocked for a specified time           |
| <code>suspend( )</code> | // blocked until further orders           |
| <code>wait( )</code>    | // blocked until certain condition occurs |

These methods cause the thread to go into the *blocked* (or *not Runnable*) state. The thread will return to the runnable state when the specified time is elapsed in the case of `sleep()`, the `resume()` method is invoked in the case of `suspend()`, and the `notify()` method is called in the case of `wait()`.



## LIFE CYCLE OF A THREAD

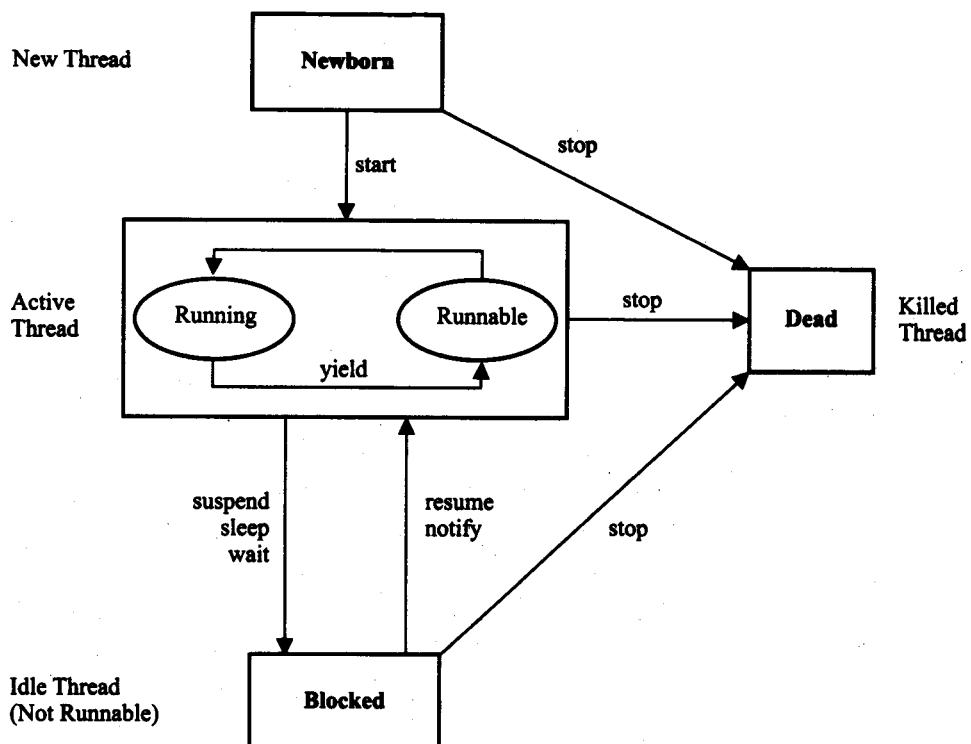
During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state

4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig. 12.3.

**Fig. 12.3**



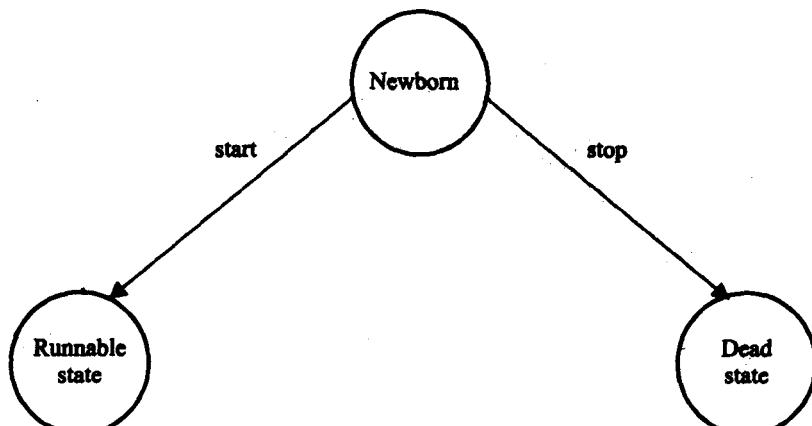
**State transition diagram of a thread**

## Newborn State

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

- Schedule it for running using `start()` method.
- Kill it using `stop()` method.

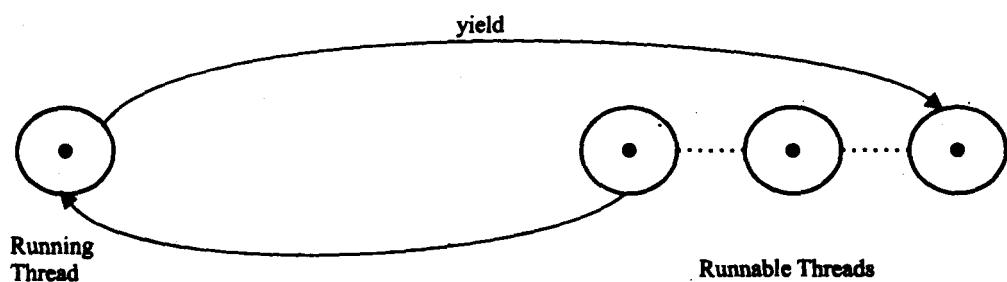
If scheduled, it moves to the runnable state (Fig. 12.4). If we attempt to use any other method at this stage, an exception will be thrown.

**Fig. 12.4****Scheduling a newborn thread**

### Runnable State

The Runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as *time-slicing*.

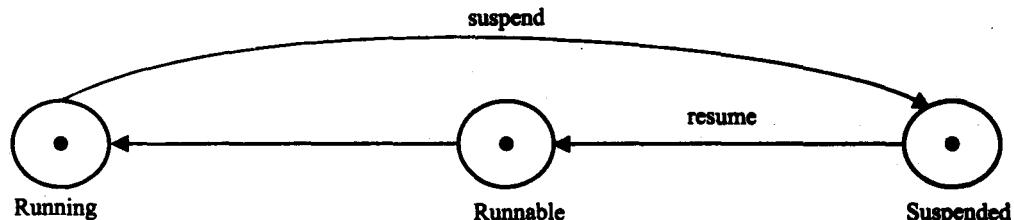
However, if we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using the `yield()` method (Fig. 12.5).

**Fig. 12.5****Relinquishing control using `yield()` method**

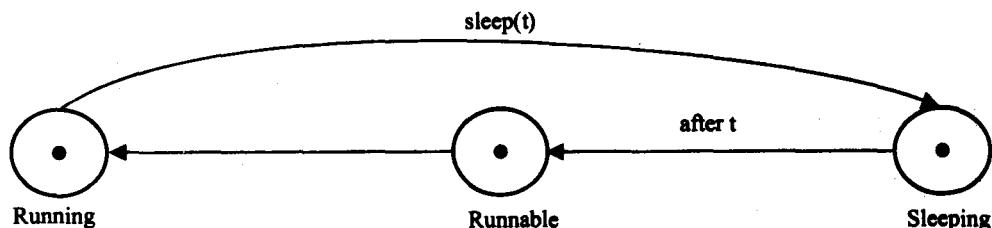
### Running State

*Running* means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

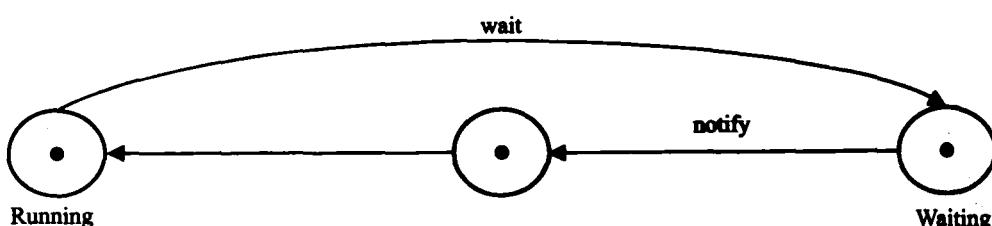
1. It has been suspended using `suspend()` method. A suspended thread can be revived by using the `resume()` method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

Fig. 12.6***Relinquishing control using suspend() method***

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method `sleep(time)` where `time` is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.

Fig. 12.7***Relinquishing control using sleep() method***

3. It has been told to wait until some event occurs. This is done using the `wait()` method. The thread can be scheduled to run again using the `notify()` method.

Fig. 12.8***Relinquishing control using wait() method***

## Dead State

A thread is said to be *blocked* when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

## Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.



## USING THREAD METHODS

We have discussed how Thread class methods can be used to control the behaviour of a thread. We have used the methods start() and run() in Program 12.1. There are also methods that can move a thread from one state to another. Program 12.2 illustrates the use of yield(), sleep(), and stop() methods. Compare the outputs of Programs 12.1 and 12.2.

---

### Program 12.2 Use of yield(), stop(), and sleep() methods

---

```
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if (i==1) yield();
            System.out.println("From Thread A : i = ~ +i");
        }
        System.out.println("Exit from A ~");
    }
}

class B extends Thread
{
    public void run()
    {
        for(int j=1; j<=5; j++)
        {
            System.out.println("From Thread B : j = ~ +j");
            if (j==3) .stop();
        }
    }
}
```

---

(Continued)

*Program 12.2 (Continued)*

---

```
        }
        System.out.println("Exit      from a");
    }
}

class C extends Thread
{
    public void run()
    {
        for(int k=1; k<=5; k++)
        {
            System.out.println("\tFrom      Thread C :  k = " +k);
            if (k==1)
                try
                {
                    sleep(1000);
                }
            catch (Exception e)
            {
            }
        }
        System.out.println("Exit      from C ");
    }
}

class ThreadMethods
{
    public static void main (String args[])
    {
        A threadA = new A();
        a threadB = new a();
        C threadC = new C();

        System.out.println("Start      thread A");
        threadA.start();

        System.out.println("Start      thread a");
        threadB.start();

        System.out.println("Start      thread C");
        threadC.start();

        System.out.println("End      of main thread");
    }
}
```

Here is the output of Program 12.2:

```

Start thread A
Start thread B
Start thread C
    From Thread B: j = 1
    From Thread B: j = 2
    From Thread A: i = 1
    From Thread A: i = 2
End of main thread
    From Thread C: k = 1
    From Thread B: j = 3
    From Thread A: i = 3
    From Thread A: i = 4
    From Thread A: i = 5
Exit from A
    From Thread C: k = 2
    From Thread C: k = 3
    From Thread C: k = 4
    From Thread C: k = 5
Exit from C

```

Program 12.2 uses the `yield()` method in thread A at the iteration  $i = 1$ . Therefore, the thread A, although started first, has relinquished its control to the thread B. The `stop()` method in thread B has killed it after implementing the for loop only three times. Note that it has not reached the end of `run()` method. The thread C started sleeping after executing the `for` loop only once. When it woke up (after 1000 milliseconds), the other two threads have already completed their runs and therefore was running alone. The main thread died much earlier than the other three threads.



## THREAD EXCEPTIONS

Note that the call to `sleep()` method is enclosed in a try block and followed by a catch block. This is necessary because the `sleep()` method throws an exception, which should be caught. If we fail to catch the exception, program will not compile.

Java run system will throw `IllegalThreadStateException` whenever we attempt to invoke a method that a thread cannot handle in the given state. For example, a sleeping thread cannot deal with the `resume()` method because a sleeping thread cannot receive any instructions. The same is true with the `suspend()` method when it is used on a blocked (Not Runnable) thread.

Whenever we call a thread method that is likely to throw an exception, we have to supply an appropriate exception handler to catch it. The catch statement may take one of the following forms:

---

```

catch (ThreadDeath e)
{
    .....
    ..... // Killed thread
}
catch (InterruptedException e)
{
    .....
    ..... // Cannot handle it in the current state
}
catch (IllegalArgumentException e)
{
    .....
    ..... // Illegal method argument
}
catch (Exception e)
{
    .....
    ..... // Any other
}

```

---

Exception handling is discussed in detail in Chapter 13.



## THREAD PRIORITY

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis.

Java permits us to set the priority of a thread using the `setPriority( )` method as follows:

`ThreadName.setPriority(intNumber);`

The `intNumber` is an integer value to which the thread's priority is set. The `Thread` class defines several `priority` constants:

|               |      |
|---------------|------|
| MIN_PRIORITY  | = 1  |
| NORM_PRIORITY | = 5  |
| MAXPRIORITY   | = 10 |

The `intNumber` may assume one of these constants or any value between 1 and 10. Note that the default setting is `NORM_PRIORITY`.

Most user-level processes should use NORM\_PRIORITY, plus or minus 1. Back-ground tasks such as network I/O and screen repainting should use a value very near to the lower limit. We should be very cautious when trying to use very high priority values. This may defeat the very purpose of using multithreads.

By assigning priorities to threads, we can ensure that they are given the attention (or lack of it) they deserve. For example, we may need to answer an input as quickly as possible. Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and executes it. For a thread of lower priority to gain control, one of the following things should happen:

1. It stops running at the end of run().
2. It is made to sleep using sleep( ).
3. It is told to wait using wait( ).

However, if another thread of a higher priority comes along, the currently running thread will be *preempted* by the incoming thread thus forcing the current thread to move to the runnable state. Remember that the highest priority thread always preempts any lower priority threads.

Program 12.3 and its output illustrate the effect of assigning higher priority to a thread. Note that although the thread A started first, the higher priority thread B has preempted it and started printing the output first. Immediately, the thread C that has been assigned the highest priority takes control over the other two threads. The thread A is the last to complete.

### Program 12.3 Use of priority in threads

---

```
class A extends Thread
{
    public void run()
    {
        System.out.println("threadA started");
        for(int i=1; i<=4; i++)
        {
            System.out.println("\tFrom Thread A : " + i);
        }
        System.out.println("Exit from A");
    }
}

class B extends Thread
{
    public void run()
    {
        System.out.println("threadB started");
        for(int j=1; j<=4; j++)
        {

```

---

(Continued)

.Program 12.3 (*Continued*)

---

```
        System.out.println("\tFrom      Thread  B :  j = " +j);
    }
    System.out.println("Exit      from  B ");
}
}

class C extends Thread
{
    public void run()
    {
        System.out.println("threadC      started");
        for(int k=1; k<=4; k++)
        {
            System.out.println("\tFrom      Thread  C :  k ..." +k);
        }
        System.out.println("Exit      from  C ");
    }
}

class ThreadPriority
{
    public static void main (String args[  ])
    {
        A threadA = new A();
        B threadB = new B();
        C threadC = new C();

        threadC.setPriority(Thread.MAX_PRIORITY);
        threadB.setPriority (threadA.getPriority() +1);
        threadA.setPriority(Thread.MIN_PRIORITY);

        System.out.println("Start      thread  A");
        threadA.start();

        System.out.println("Start      thread  B");
        threadB.start();

        System.out.println("Start      thread  C");
        threadC.start();

        System.out.println("End      of main thread-");
    }
}
```

Output of Program 12.3:

```
Start    thread    A
Start    thread    B
Start    thread    C
threadB    started
    From Thread    B : j. = 1
    From Thread    B : j = 2
threadC    started
    FroII,l Thread    C : k = 1
    From Thread    C : k = 2
    From Thread    C : k = 3
    From Thread    C : k = 4
Exit    from    C
End of main thread
    From Thread    B : j = 3
    From Thread    B : j 7 4
Exit    from    B
threadA    started
    From Thread    A : i = 1
    From Thread    A : i = 2
    From Thread    A : i = 3
    From Thread    A : i = 4
Exit    from    A
```



## SYNCHRONIZATION

So far, we have seen threads that use their own data and methods provided inside their run() methods. What happens when they try to use data and methods outside themselves? On such occasions, they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as *synchronisation*.

In case of Java, the keyword synchronized helps to solve such problems by keeping a watch on such locations. For example, the method that will read information from a file and the method that will update the same file may be declared as synchronized. Example:

```
synchronized    void    update(    )
{
    .....
    .....
        // code here is synchronized
    .....
}
```

When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock.

It is also possible to mark a block of code as synchronized as shown below:

```
synchronized (lock-object)
{
    .....
        // code here is synchronized
    .....
}
```

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

An interesting situation may occur when two or more threads are waiting to gain control of a resource. Due to some reason, the condition on which the waiting threads rely on to gain control does not happen. This results in what is known as *deadlock*. For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets hold of Method2. Because these are mutually exclusive conditions, a deadlock occurs. The code below illustrates this:

#### Thread A

```
synchronized method2( )
{
    synchronized method1( )
    {
        .....
    }
}
```

#### Thread B

```
synchronized method1( )
{
    synchronized method2( )
    {
        .....
    }
}
```



## IMPLEMENTING THE 'RUNNABLE' INTERFACE

We stated earlier that we can create threads in two ways: one by using the extended Thread class and another by implementing the Runnable interface. We have already discussed in detail how the Thread class is used for creating and running threads. In this section, we shall see how to make use of the Runnable interface to implement threads.

The Runnable interface declares the run() method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

1. Declare the class as implementing the Runnable interface.
2. Implement the run() method.
3. Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
4. Call the thread's start() method to run the thread.

Program 12.4 illustrates the implementation of the above steps. In main method, we first create an instance of X and then pass this instance as the initial value of the object threadX (an object of Thread Class). Whenever, the new thread threadX starts up, its run() method calls the run() method of the target object supplied to it. Here, the target object is nmnable. If the direct reference to the thread threadX is not required, then we may use a shortcut as shown below:

```
new Thread (new X( ) .start( ));
```

*Program 12.4 Usln, Runnabe Inter(Gce*

---

```
class X implements Runnable // Step 1
{
    public void run() // Step 2
    {
        for(int i = 1; i<=10; i++)
        {
            System.out.println("\tThreadX      :~ " +i);
        }
        System.out.println("End      of ThreadX");
    }
}

class RunnableTest
{
    public static void main (String args[ ])
    {
        X runnable = new X( );
        Thread threadX = new Thread(runnable); // Step 3
    }
}
```

---

*(Continued)*

```

        threadx.start( );
        // Step 4
        System.out.println("End      of main Thread");
    }
}

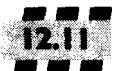
```

Output of Program 12.4:

```

End of main Thread
ThreadX : 1
ThreadX : 2
ThreadX : 3
Thr.eadX : 4
ThreadX : 5
ThreadX : 6
ThreadX : 7
ThreadX : 8
ThreadX : 9
ThreadX : 10
End of ThreadX

```



## SUMMARY

A thread is a single line of execution within a program. Multiple threads can run concurrently in a single program. A thread is created either by subclassing the `Thread` class or implementing the `Runnable` interface. We have discussed both the approaches in detail in this chapter. We have also learned the following in this chapter:

- How to synchronize threads,
- How to set priorities for threads, and
- How to control the execution of threads

Careful application of multithreading will considerably improve the execution speed of Java programs.

### KEY TERMS

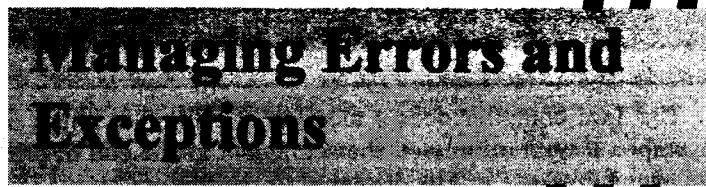
**Thread, Multithread, Multitask, Concurrency, Lightweight Processes, Time-slicing, Priority, Synchronization, Deadlock.**

## REVIEW QUESTIONS

---

- 12.1 What is a thread?
- 12.2 What is the difference between multiprocessing and multithreading? What is to be done to implement these in a program?
- 12.3 What Java interface must be implemented by all threads?
- 12.4 How do we start a thread?
- 12.5 What are the two methods by which we may stop threads?
- 12.6 What is the difference between suspending and stopping a thread?
- 12.7 How do we set priorities for threads?
- 12.8 Describe the complete life cycle of a thread.
- 12.9 What is synchronization? When do we use it?
- 12.10 Develop a simple real-life application program to illustrate the use of multi threads.

# **Chapter 13**



**13.1**

## **INTRODUCTION**

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. *Errors* are the wrongs that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.



## **TYPES OF ERRORS**

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

### **Compile-Time Errors**

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

### **Program 13.1 illustration of compile-time errors**

---

```
/* This prgram contains an error */
class Error{
    public static void main (String args[ ]){
        System.out.println("Hello Java!") // Missing;
    }
}
```

---

The Java compiler does a nice job of telling us where the errors are in the program; For example, if we have missed the semicolon at the end of print statement in Program 13.1, the following message will be displayed in the screen:

```
Error1.java :7: ';' expected
System.out.println ("Hello Java!")
^
1 error
```

We can now go to the appropriate line, correct the error, and recompile the program. Sometimes, a single error may be the source of multiple errors later in the compilation. For example, use of an undeclared variable in a number of places wUIcause a series of errors of type "undefined variable". We should generally consider the earliest errors as the major source of our problem. After we fix such an error, we should recompile the program and look for other errors.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of == operator
- And so on

Other errors we may encounter are related to directory paths. An error such as

```
javac : command not found
```

means that we have not set the path correctly. We must ensure that the path includes the directory where the Java executables are stored.

## Run-Time Errors

Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Trying to cast an instance of a class to one of its subclasses.
- Passing a parameter that is not in a valid range or value for a method.
- Trying to illegally change the state of a thread.
- Attempting to use a negative size for an array.
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting invalid string to a number.
- Accessing a character that is out of bounds of a string.
- And many more

When such errors are encountered, Java typically generates an error message and aborts the program. Program 13.2 illustrates how a run-time error causes termination of execution of the program.

### Program 13.2 Illustration of run-time errors

---

```
class Error2
{
    public static void main (String args[])
    {
        int a = 10;
        int b = 5;
        int c = 5;

        int x = a/(b-c);      // Division by zero
        System.out.println("x = " + x);

        int y = a/(b+c);
        System.out.println("y = " + y);
    }
}
```

---

Program 13.2 is syntactically correct and therefore does not cause any problem during compilation. However, while executing, it displays the following message and stops without executing further statements.

```
java.lang.ArithmetricException: / by zero  
at Error2.main(Error2.java:10)
```

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

### 13.3

## EXCEPTIONS

An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it(i.e., informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the output of Program 13.2 and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*.

The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem (*Hit* the exception).
2. Inform that an error has occurred (*Throw* the exception)
3. Receive the error information (*Catch* the exception)
4. Take corrective actions (*Handle* the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exceptions that we must watch out for catching are listed in Table 13.1.

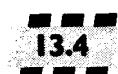
Table 13.1 Common Java Exceptions

| Exception Type                 | Cause of Exception                                                      |
|--------------------------------|-------------------------------------------------------------------------|
| ArithmetricException           | Caused by math errors such as division by zero                          |
| ArrayIndexOutOfBoundsException | Caused by bad array indexes                                             |
| ArrayStoreException            | Caused when a program tries to store the wrong type of data in an array |
| FileNotFoundException          | Caused by an attempt to access a nonexistent file                       |

(Continued)

**Table 13.1 (Continued)**

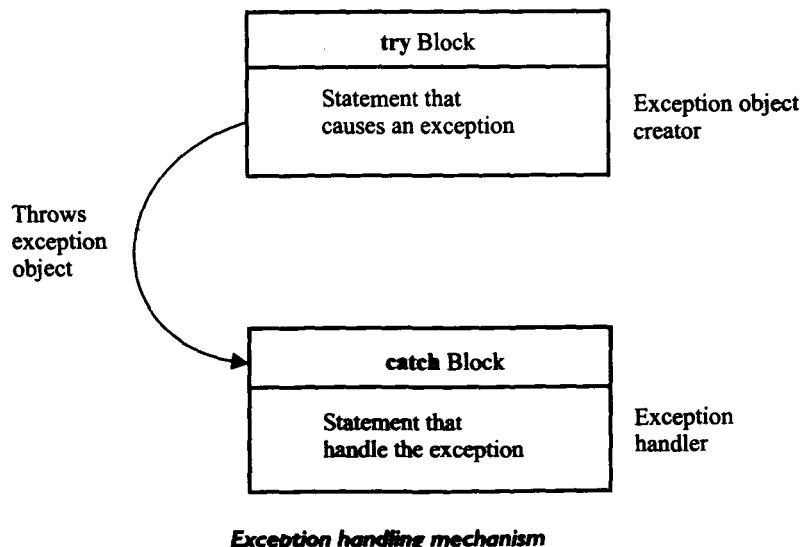
| Exception Type                  | Cause of Exception                                                                             |
|---------------------------------|------------------------------------------------------------------------------------------------|
| IOException                     | Caused by general I/O failures, such as inability to read from a file                          |
| NullPointerException            | Caused by referencing a null object                                                            |
| NumberFormatException           | Caused when a conversion between strings and number fails                                      |
| OutOfMemoryException            | Caused when there's not enough memory to allocate a new object                                 |
| SecurityException               | Caused when an applet tries to perform an action not allowed by the browser's security setting |
| StackOverflowException          | Caused when the system runs out of stack space                                                 |
| StringIndexOutOfBoundsException | Caused when a program attempts to access a nonexistent character position in a string          |



#### SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of exception handling are *throwing* an exception and *catching* it. This is illustrated in Fig. 13.1.

**Fig. 13.1**



Java uses a keyword try to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword catch "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple try and catch statements:

---

```
....  
....  
try  
{  
    statement;          // generates an exception  
}  
catch (Exception type e)  
{  
    statement;          // processes the exception  
}  
....  
....
```

---

The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every try statement should be followed by *at least* one catch statement; otherwise compilation error will occur.

Note that the catch statement works like a method definition.-The catch statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

Program 13.3 illustrates the use of try and catch blocks to handle an arithmetic exception. Note that Program 13.3 is a modified version of Program 13.2.

#### Program 13.3 Usln, try and catch for exception handlin,

---

```
class Error3  
{  
    public static void main (String args[ ])  
    {  
        int a = 10;  
        int b = 5;  
        int c = 5;  
        int x, y ;
```

---

(Continued)

---

Program 13.3 (*Continued*)

---

```
try
{
    x = a / (b-c) ;      // Exception here
}.
catch (ArithmaticException e)
{
    System.out.println("Division by zero");
}
y = a / (b+c) ;
System.out.println("y = U + y");
}
```

---

Program 13.3 displays the following output:

```
Division by zero.
y = 1
```

Note that the program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution, as if nothing has happened. Compare with the output of Program 13.2 which did not give the value of y.

Program 13.4 shows another example of using exception handling mechanism. Here, the try-catch block catches the invalid entries in the list of command line arguments.

---

**Program ,3.4 *Catch'n,'nvalid command line arguments***

---

```
class CLineInput
{
    public static void main (String args[])
    {
        int invalid = 0;                      // Number of invalid arguments
        int number, count = 0;
        for(int i = 0; i < args.length; i++)
        {
            try
            {
                number = Integer.parseInt(args[i]);
            }
            catch (NumberFormatException e)
            {
```

---

(*Continued*)

---

Program 13.4 (Continued)

---

```
        invalid = invalid + 1; // Caught an invalid number
        System.out.println("Invalid      Number: " + args[i])
        continue;               // Skip the remaining part of the loop
    }
    count = count + 1;
}
System.out.println("Valid      Numbers = " + count);
System.out.print("Invalid      Numbers = " + invalid);
}
}
```

---

Note the use of the wrapper class Integer to obtain an int number from a string:

```
number = Integer.parseInt(args[i])
```

Remember that the numbers are supplied to the program through the command line and therefore they are stored as strings in the array args[ ]. Since the above statement is placed in the try block, an exception is thrown if the string is improperly formatted and the number is not included in the count.

When we run the program with the command line:

```
java CLineInput 15 25.75 40 Java 10.5 65
```

it produces the following output:

```
Invalid      Number: 25.75
Invalid      Number: Java
Invalid      Number: 10.5
Valid      Numbers = 3
Invalid      Numbers = 3
```

## 13.5 MULTIPLE CATCH STATEMENTS

It is possible to have more than one catch statement in the catch block as illustrated below:

---

```
.....
.....
try
{
    statement ;           // generates an exception
}
catch (Exception-Type-1 e)
```

```

{
    statement;                      // processes exception type 1
}

catch (Exception-Type-2      e)
{
    statement;                      // processes exception type 2
}
-
-
-
catch (Exception-Type-N      e)
{
    statement ;                   // processes exception type N
}
-----
-----

```

---

When an exception in a try block is generated, the Java treats the multiple catch statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

Note that Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example:

```
        catch (Exception      e) ;
```

The catch statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

### **Program 13.5 Using, multiple catch blocks**

---

```

class Error4
{
    public static void main (String args[  ] )
    {
        int a[ 1 ] = {S,IO};
        int b = S;

        try
        {
            int x = a[2] / b - a[1];
        }

        catch(ArithmeicException      e)
        {

```

---

*(Continued)*

(Program 13.5 (Continued))

---

```
        System.out.println("Division      by zero");
    }

    catch (ArrayIndexOutOfBoundsException      e)
    {
        System.out.println("Array      index error");
    }

    catch(ArrayStoreException      e)
    {
        System.out.println("Wrong      data type");
    }

    int y = a[1] / a [0];
    System.out.println("y      = " + y);
}
}
```

---

Program 13.5 uses a chain of catch blocks and, when run, produces the following output:

```
Array index error
y = 2
```

Note that the array element a[2] does not exist because array a is defined to have only two elements, a(0) and a(1). Therefore, the index 2 is outside the array boundary thus causing the block

```
Catch(ArrayIndexOutOfBoundsException      e)
```

to catch and handle the error. Remaining catch blocks are skipped.



## USING finally STATEMENT

Java supports another statement known as finally statement that can be used to handle an exception that is not caught by any of the previous catch statements. finally block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:

---

```
try
{
    .....
}

try
{
    .....
}
```

---

```

finally
{
    .....
}

}

catch (....)
{
    .....
}

catch (....)
{
    .....
}

}

.
.
.

finally
{
    .....
}

```

---

When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

In Program 13.5, we may include the last two statements inside a finally block as shown below:

```

finally
{
    int y = a[1]/a[0];
    System.out.println("y = " +y);
}

```

This will produce the same output.



## THROWING OUR OWN EXCEPTIONS

There may be times when we would like to throw our own exceptions. We can do this by using the keyword `throw` as follows:

`throw new Throwable subclass;`

Examples:

```
throw new ArithmeticException();
throw new NumberFormatException();
```

Program 13.6 demonstrates the use of a user-defined subclass of Throwable class. Note that Exception is a subclass of Throwable and therefore MyException is a subclass of Throwable class. An object of a class that extends Throwable can be thrown and caught.

### **Program 13.6 Throwing our own exception**

---

```
import java.lang.Exception;

class MyException extends Exception
{
    MyException(String message)
    {
        super(message);
    }
}

class TestMyException
{
    public static void main (String args[ ])
    {
        int x = 5, y = 1000;

        try
        {
            float z = (float) x / (float) y ;
            if(z < 0.01)
            {
                throw new MyException("Number is too small");
            }
        }

        catch (MyException e)
        {
            System.out.println("Caught my exception");
            System.out.print In(e.getMessage());
        }

        finally
        {
            System.out.println("I am always here");
        }
    }
}
```

A run of program 13.6 produces:

```
Caught my exception  
Number is too small  
I am always here
```

The object e which contains the error message "Number is too small" is caught by the catch block which then displays the message using the getMessage( ) method.

Note that Program 13.6 also illustrates the use of finally block. The last line of output is produced by the finally block.

**13.8**

## USING EXCEPTIONS FOR DEBUGGING

As we have seen, the exception-handling mechanism can be used to hide errors from rest of the program. It is possible that the programmers may misuse this technique for hiding errors rather than debugging the code. Exception handling mechanism may be effectively used to locate the type and place of errors. Once we identify the errors, we must try to find out why these errors occur before we cover them up with exception handlers.

**13.9**

## SUMMARY

A good program does not produce unexpected results. We should incorporate features that could check for potential problem spots in programs and guard against program failures. The problem conditions known as exceptions in Java must be handled carefully to avoid any program failures.

In this chapter we have discussed the following:

- What exceptions are
- How to throw system exceptions
- How to define our own exceptions
- How to catch and handle different types of exceptions
- Where to use exception handling tools

We must ensure that common exceptions are handled where appropriate.

### KEY TERMS

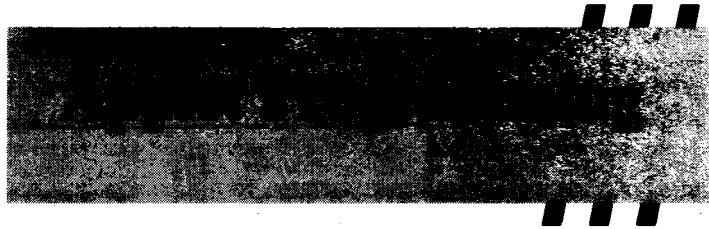
**Errors, Compile-time errors, Run-time errors, Exception, Exception handling, try, catch, throw, finally.**

## REVIEW QUESTIONS

---

- 13.1 What is an exception?
- 13.2 How do we define a try block?
- 13.3 How do we define a catch block?
- 13.4 List some of the most common types of exceptions that might occur in Java. Give examples.
- 13.5 Is it essential to catch all types of exceptions?
- 13.6 How many catch blocks can we use with one try block?
- 13.7 Create a try block that is likely to generate three types of exception and then incorporate necessary catch blocks to catch and handle them appropriately.
- 13.8 What is a finally block? When and how is it used? Give a suitable example.
- 13.9 Explain how exception handling mechanism can be used for debugging a program.
- 13.10 Define an exception called "NoMatchException" that is thrown when a string is not equal to "India". Write a program that uses this exception.

# **Chapter 14**



## **INTRODUCTION**

Applets are small Java programs that are primarily used in Internet computing. They can be transported over the Internet from one computer to another and run using the Applet Viewer or any Web browser that supports Java. An applet, like any application program, can do many things for us. It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation, and play interactive games.

Java has revolutionized the way the Internet users retrieve and use documents on the world wide network. Java has enabled them to create and use fully interactive multimedia Web documents. A web page can now contain not only a simple text or a static image but also a Java applet which, when run, can produce graphics, sounds and moving images. Java applets therefore have begun to make a significant impact on the World Wide Web.

### **Local and Remote Applets**

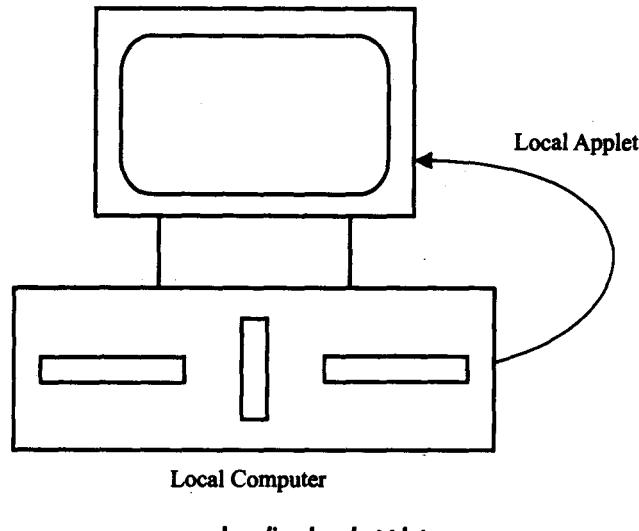
We can embed applets into Web pages in two ways. One, we can write our own applets and embed them into Web pages. Second, we can download an applet from a remote computer system and then embed it into a Web page.

An applet developed locally and stored in a local system is known as a *local applet*. When a Web page is trying to find a local applet, it does not need to use the Internet and therefore the local system does not require the Internet connection. It simply searches the directories in the local system and locates and loads the specified applet (see Fig. 14.1).

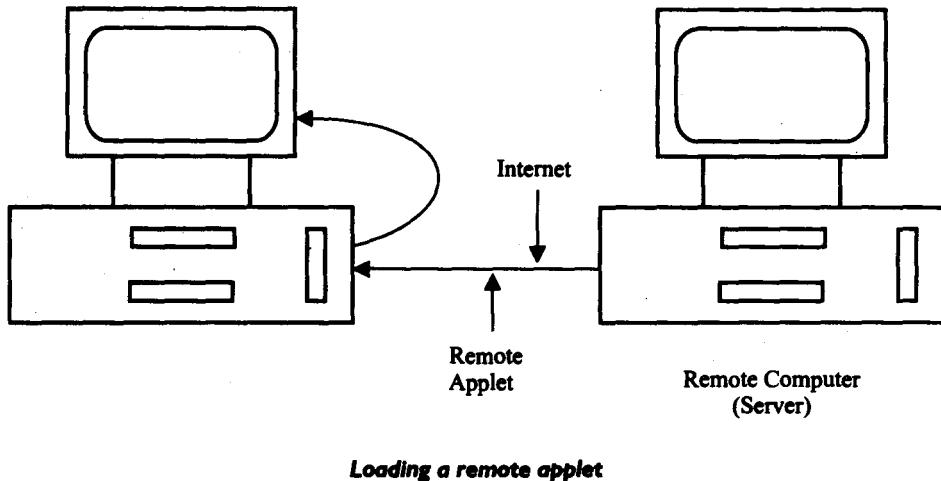
A *remote applet* is that which is developed by someone else and stored on a remote computer connected to the Internet. If our system is connected to the Internet, we can download the remote applet onto our system via the Internet and run it (see Fig. 14.2).

In order to locate and load a remote applet, we must know the applet's address on the Web. This address is known as *Uniform Resource Locator (URL)* and must be specified in the applet's HTML document as the value of the CODEBASE attribute (see Section 14.11). Example:

**Fig. 14.1**



**Fig. 14.2**



CODEBASE = <http://www.netserve.com/applets>

In the case of local applets, CODEBASE may be absent or may specify a local directory.

In this chapter we shall discuss how applets are created, how they are located in the Web documents and how they are loaded and run in the local computer.



## HOW APPLETS DIFFER FROM APPLICATIONS

Although both the applets and stand-alone applications are Java programs, there are significant differences between them. Applets are not full-featured application programs. They are usually written to accomplish a small task or a component of a task. Since they are usually designed for use on the Internet, they impose certain limitations and restrictions in their design.

- Applets do not use the `main()` method for initiating the execution of the code. Applets, when loaded, automatically call certain methods of `Applet` class to start and execute the applet code.
- Unlike stand-alone applications, applets cannot be run independently. They are run from inside a Web page using a special feature known as HTML tag.
- Applets cannot read from or write to the files in the local computer.
- Applets cannot communicate with other servers on the network.
- Applets cannot run any program from the local computer.
- Applets are restricted from using libraries from other languages such as C or C++. (Remember, Java language supports this feature through native methods).

All these restrictions and limitations are placed in the interest of security of systems. These restrictions ensure that an applet cannot do any damage to the local system.



## PREPARING TO WRITE APPLETS

Until now, we have been creating simple Java application programs with a single `main()` method that created objects, set instance variables and ran methods. Here, we will be creating applets exclusively and therefore we will need to know

- When to use applets,
- How an applets works,
- What sort of features an applet has, and
- Where to start when we first create our own applets.

First of all, let us consider the situations when we might need to use applets.

1. When we need something dynamic to be included in the display of a Web page. For example, an applet that displays daily sensitivity index would be useful on a page that lists share prices of various companies or an applet that displays a bar chart would add value to a page that contains data tables.
2. When we require some "flash" outputs. For example, applets that produce sounds, animations or some special effects would be useful when displaying certain pages.
3. When we want to create a program and make it available on the Internet for us by others on their computers.

Before we try to write applets, we must make sure that Java is installed properly and also ensure that either the Java appletviewer or a Java-enabled browser is available. The steps involved in developing and testing an applet are:

1. Building an applet code (.java file)
2. Creating an executable applet (.class file)
3. Designing a Web page using HTML tags
4. Preparing <APPLET> tag
5. Incorporating <APPLET>tag into the Web page
6. Creating HTML file
7. Testing the applet code

Each of these steps is discussed in the following sections.

## BUILDING APPLET CODE

It is essential that our applet code uses the services of two classes, namely, Applet and Graphics from the Java class library. The Applet class which is contained in the java.applet package provides life and behaviour to the applet through its methods such as init( ), start( ) and paint( ). Unlike with applications, where Java calls the main( ) method directly to initiate the execution of the program, when an applet is loaded, Java automatically calls a series of Applet class methods for starting, running, and stopping the applet code. The Applet class therefore maintains the *lifecycle* of all applet.

The paint( ) method of the Applet class, when it is called, actually displays the result of the applet code on the screen. The output may be text, graphics, or sound. The paint( ) method, which requires a Graphics object as an argument, is defined as follows:

```
public void paint(Graphics g)
```

This requires that the applet code imports the java.awt package that contains the Graphics class. All output operations of an applet are performed using the methods defined in the Graphics class. It is thus clear from the above discussions that an applet code will have a general format as shown below:

---

```
import java.awt.*;
import java.applet.*;

.....
.....
public class appletclassname extends Applet
{
    .....
    .....
```

```

public void paint (Graphics g)
{
    .....
    .....      // Applet operations code
    .....
}

.....
.....
}

```

---

The *appletclassname* is the main class for the applet. When the applet is loaded, Java creates an instance of this class, and then a series of Applet class methods are called on that instance to execute the code. Program 14.1 shows a simple HelloJava applet.

#### **Program 14.1 The HelloJava applet**

---

```

import java.awt.*;
import java.applet.*;
public class HelloJava extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString("Hello Java", 10, 100);
    }
}

```

---

The applet contains only one executable statement

```
g.drawString("Hello Java", 10, 100);
```

which, when executed, draws the string

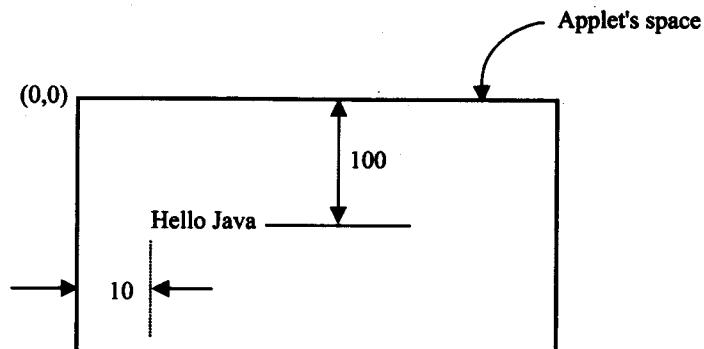
Hello Java

at the position 10, 100 (pixels) of the applet's reserved space as shown in Fig. 14.3.

Remember that the applet code in Program 14.1 should be saved with the file name HelloJava.java, in a java subdirectory. Note the public keyword for the class HelloJava. Java requires that the main applet class be declared public.

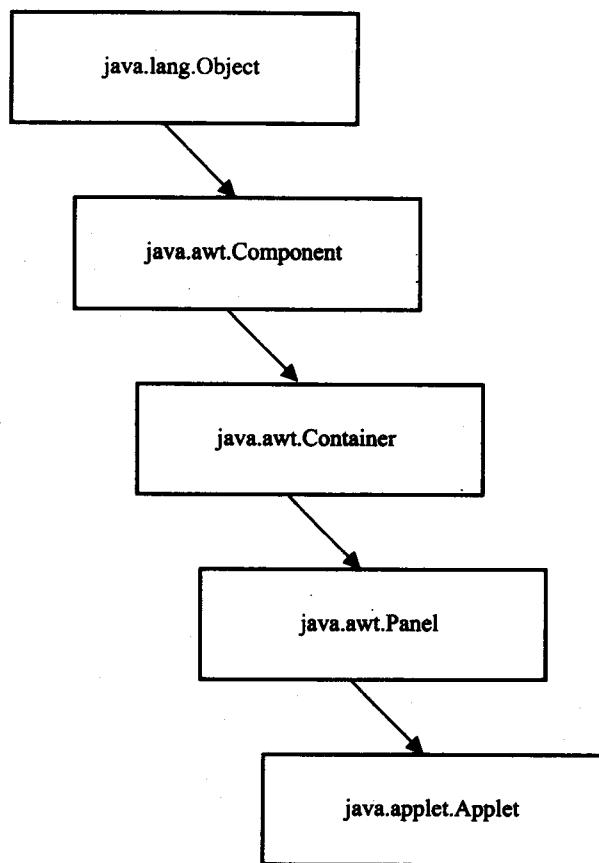
Remember that Applet class itself is a subclass of the Panel class, which is again a subclass of the Container class and so on as shown in Fig. 14.4. This shows that the main applet class inherits properties from a long chain of classes. An applet can, therefore, use variables and methods from all these classes.

Fig. 14.3



*Output of Program 14.1*

Fig. 14.4



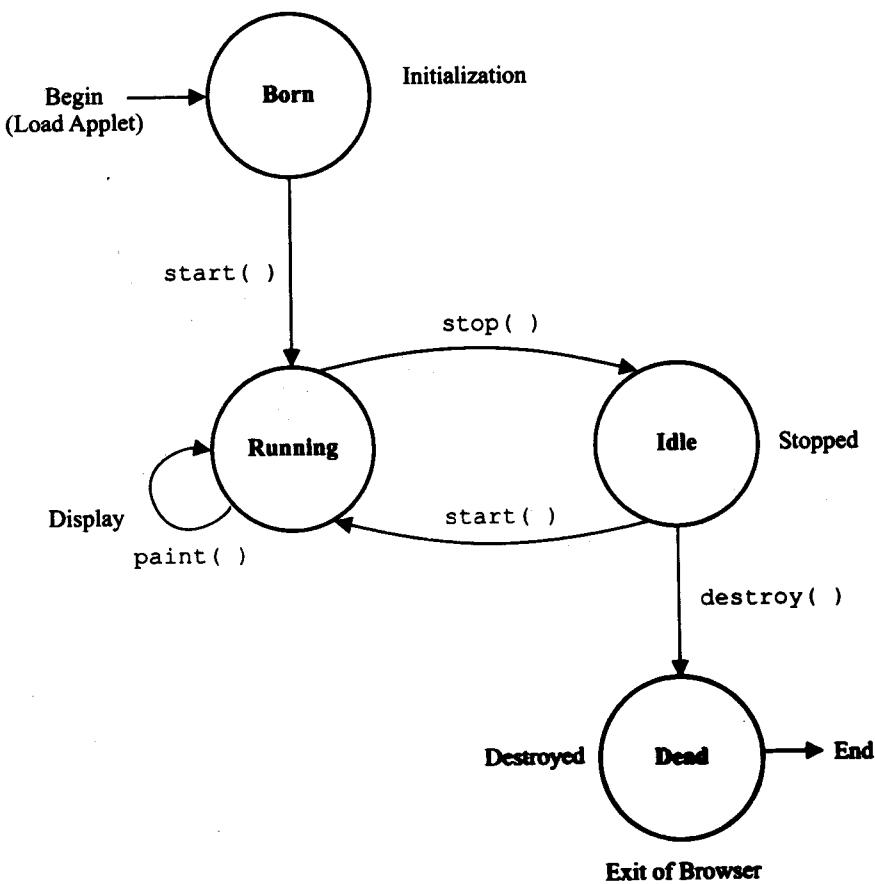
*Chain of classes inherited by Applet class*

## APPLET LIFE CYCLE

Every Java applet inherits a set of default behaviours from the **Applet** class. As a result, when an applet is loaded, it undergoes a series of changes in its state as shown in Fig. 14.5. The applet states include:

- Born or initialization state
- Running state
- Idle state
- Dead or destroyed state

**Fig. 14.5**



An applet's state transition diagram

## Initialization State

Applet enters the *initialization* state when it is first loaded. This is achieved by calling the init() method of Applet Class. The applet is born. At this stage, we may do the following, if required.

- Create objects needed by the applet
- Set up initial values
- Load images or fonts
- Set up colors

The initialization occurs only once in the applet's life cycle. To provide any of the behaviours mentioned above, we must override the init( ) method:

```
public void init( )
{
    .....
    ..... (Action)
    .....
}
```

## Running State

Applet enters the *running* state when the system calls the start( ) method of Applet Class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in "stopped" (idle) state. For example, we may leave the Web page containing the applet temporarily to another page and return back to the page. This again starts the applet running. Note that, unlike init( ) method, the start( ) method may be called more than once. We may override the start( ) method to create a thread to control the applet.

```
public void start( )
{
    .....
    ..... (Action)
    .....
}
```

## Idle or Stopped State

An applet becomes *idle* when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the stop( ) method explicitly. If we use a thread to run the applet, then we must use stop( ) method to terminate the thread. We can achieve this by overriding the stop( ) method.

```
public void stop( )
{
    .....
    ..... (Action)
    .....
}
```

## **Dead State**

An applet is said to be *dead* when it is removed from memory. This occurs automatically by invoking the `destroy()` method when we quit the browser. Like initialization, destroying stage occurs only once in the applet's life cycle. If the applet has created any resources, like threads, we may override the `destroy()` method to clean up these resources.

```
public void destroy()
{
    .....
    ..... (Action)
    .....
}
```

## **Display State**

Applet moves to the *display* state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into the running state. The `paint()` method is called to accomplish this task. Almost every applet will have a `paint()` method. Like other methods in the life cycle, the default version of `paint()` method does absolutely nothing. We must therefore override this method if we want anything to be displayed on the screen.

```
public void paint (Graphics g)
{
    .....
    ..... (Display statements)
    .....
}
```

It is to be noted that the display state is not considered as a part of the applet's life cycle. In fact, the `paint()` method is not defined in the `Applet` class. It is inherited from the `Component` class, a super class of `Applet`.

**14.6**

## **CREATING AN EXECUTABLE APPLET**

Executable applet is nothing but the `.class` file of the applet, which is obtained by compiling the source code of the applet. Compiling an applet is exactly the same as compiling an application. Therefore, we can use the Java compiler to compile the applet.

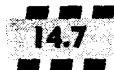
Let us consider the `HelloJava` applet created in Section 14.4.1bis. The applet has been stored in a file called `HelloJava.java`. Here are the steps required for compiling the `HelloJava` applet.

- I. Move to the directory containing the source code and type the following command:

```
javac HelloJava.java
```

2. The compiled output file called **HelloJava.class** is placed in the same directory as the source.

3. If any error message is received, then we must check for errors, correct them and compile the applet again.



## DESIGNING A WEB PAGE

Recall that Java applets are programs that reside on Web pages. In order to run a Java applet, it is first necessary to have a Web page that references that applet.

A Web page is basically made up of text and HTML tags that can be interpreted by a Web browser or an applet viewer. Like Java source code, it can be prepared using any ASCII text editor. A Web page is also known as HTMLpage or HTMLdocument. Web pages are stored using a file extension .html such as MyApplet.html. Such files are referred to as HTML files. HTML files should be stored in the same directory as the compiled code of the applets.

As pointed out earlier, Web pages include both text that we want to display and HTML tags (commands) to Web browsers. A Web page is marked by an opening HTML tag `<HTML>` and a closing HTML tag `</HTML>` and is divided into the following three major sections:

1. Comment section (Optional)
2. Head section (Optional)
3. Body section

A Web page outline containing these three sections and the opening and closing HTMLtags is illustrated in Fig. 14.6.

### Comment Section

This section contains comments about the Web page. It is important to include comments that tell us what is going on in the Web page. A comment line begins with a `<!` and ends with a `>`. Web browsers will ignore the text enclosed between them. Although comments are important, they should be kept to a minimum as they will be downloaded along with the applet. Note that comments are optional and can be included anywhere in the Web page.

### Head Section

The head section is defined with a starting `<HEAD>` tag and a closing `</HEAD>` tag. This section uSULiUycontains a title for the Web page as shown below:

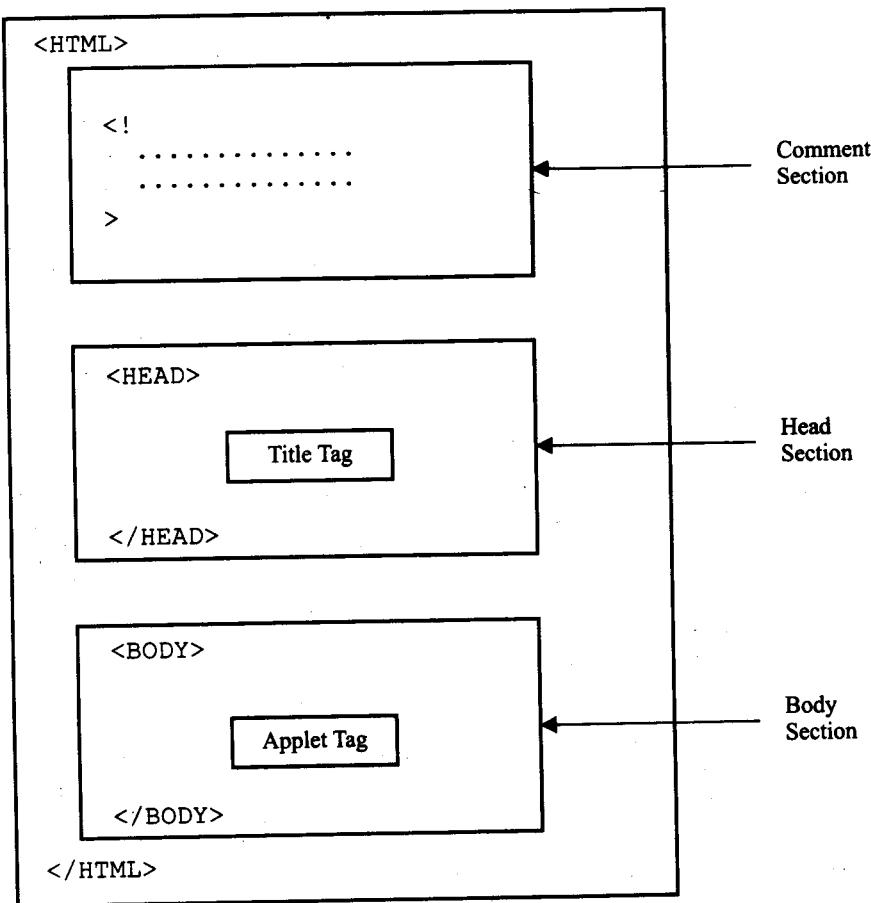
---

```
<HEAD>
    <TITLE> Welcome to Java Applets </TITLE>
</HEAD>
```

---

The text enclosed in the tags `<TITLE>`and `</TITLE>` will appear in the title bar of the Web browser when it displays the page. *The head section is also optional.*

Fig. 14.6

**A Web page template**

Note that tags `<....>` containing HTML commands usually appear in pairs such as `<HEAD>` and `</HEAD>`, and `<TITLE>` and `</TITLE>`. A slash (`/`) in a tag signifies the end of that tag section.

**Body .Section**

After the head section comes the body section. We call this as body section because this section contains the entire information about the Web page and its behaviour. We can set up many options to indicate how our page must appear on the screen (like colour, location, sound, etc.). Shown below is a simple body section:

```
<BODY>
  <CENTER>
    <H1> Welcome' to the World of Applets </H1>
  </CENTER>

  <BR>

  <APPLET ... >
  </APPLET>
</BODY>
```

---

The body shown above contains instructions to display the message

Welcome to the World of Applets

followed by the applet output on the screen. Note that the <CENTER> tag makes sure that the text is centered and <H1> tag causes the text to be of the largest size. We may use other heading tags <H2> to <H6> to reduce the size of letters in the text.

**14.8**

## APPLET TAG

Note that we have included a pair of <APPLET ... > and </APPLET> tags in the body section discussed above. The <APPLET ... > tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. The ellipsis in the tag <APPLET ..> indicates that it contains certain attributes that must be specified. The <APPLET> tag given below specifies the minimum requirements to place the HelloJava applet on a Web page:

---

```
<APPLET
  CODE = helloJava.class
  WIDTH = 400
  HEIGHT = 200 >
</APPLET >
```

---

This HTML code tells the browser to load the compiled Java applet HelloJava.class, which is in the same directory as this HTML file. And also specifies the display area for the applet output as 400 pixels width and 200 pixels height. We can make this display area appear in the centre of the screen by using the CENTER tags as shown below:

---

```
<CENTER>
  <APPLET
  .....
  .....
  ..... >
```

```
</APPLE.T>  
</CENTER>
```

---

Note that <APPLET> tag discussed above specifies three things:

1. Name of the applet
2. Width of the applet (in pixels)
3. Height of the applet (in pixels)



## ADDING APPLET TO HTML FILE

Now we can put together the various components of the Web page and create a file known as HTML file. Insert the <APPLET> tag in the page at the place where the output of the applet must appear. Following is the content of the HTML file that is embedded with the <APPLET> tag of our HelloJava applet.

---

```
<HTML>  
    <! This page includes a welcome title in the title bar and  
        also displays a welcome message. Then it specifies the  
        applet to be loaded and executed.  
    >  
  
    <HEAD>  
        <TITLE>  
            Welcome to Java Applets  
        </TITLE>  
    </HEAD>  
  
    <BODY>  
        <CENTER>  
            <H1> Welcome to the World of Applets </H1>  
        </CENTER>  
        <BR>  
        <CENTER>  
            <APPLET  
                CODB = HelloJa.. cl_.  
                WID'rII - 400  
                BBIGH'I = 200>  
            </APPu'T>  
        </CENTER>  
    </BODY>  
</HTML>
```

We must name this file as HelloJava.html and save it in the same directory as the compiled applet.

**14.10****RUNNING THE APPLET**

Now that we have created applet files as well as the HTML file containing the applet, we must have the following files in our current directory:

```
HelloJava.java  
HelloJava.class  
HelloJava.html
```

To run an applet, we require one of the following tools:

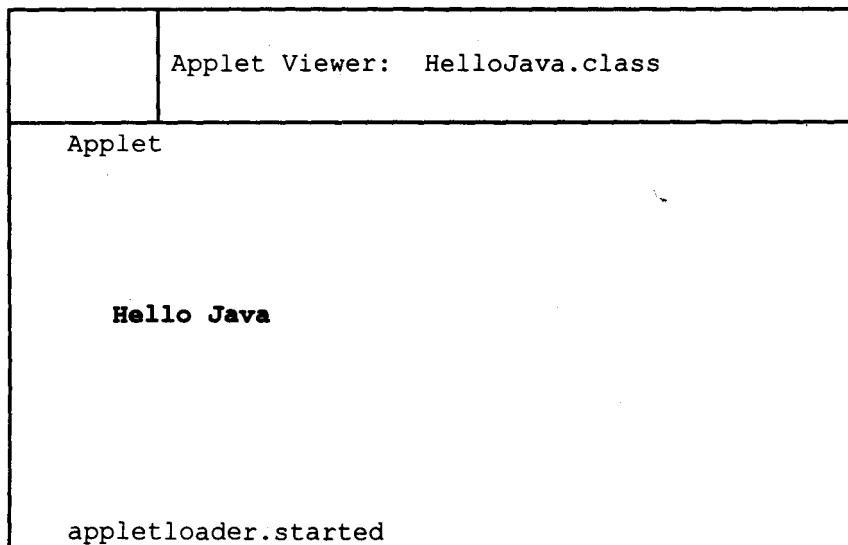
1. Java-enabled Web browser (such as HotJava or Netscape)
2. Java appletviewer

If we use a Java-enabled Web browser, we will be able to see the entire Web page containing the applet. If we use the appletviewer tool, we will only see the applet output. Remember that the appletviewer is not a full-fledged Web browser and therefore it ignores all of the HTML tags except the part pertaining to the running of the applet.

The appletviewer is available as a part of the Java Development Kit that we have been using so far. We can use it to run our applet as follows:

```
appletviewer HelloJava.html
```

Notice that the argument of the appletviewer is not the.java file or the .class file, but rather .html file. The output of our applet will be as shown in Fig. 14.7

**Fig. 14.7**

***Output of HelloJava applet by using appletviewer***



## 14.11 MORE ABOUT APPLET TAG

We have used the <APPLET> tag in its simplest form. In its simplest form, it merely creates a space of the required size and then displays the applet output in that space. The syntax of the <APPLET> tag is a little more complex and includes several attributes that can help us better integrate our applet into the overall design of the Web page. The syntax of the <APPLET> tag in full form is shown below:

---

```
<APPLET
    [ CODEBASE= codebase_URL]
    CODE = AppletFileName.class
    [ ALT = alternate_text]
    [ NAME= applet_instance_name]
    WIDTH = pixels
    HEIGHT = pixels
    [ ALIGN = alignment]
    [ VSPACE = pixels]
    [ HSPACE = pixels]
    >
    [ < PARAMNAME= name1 VALUE= value1>      ]
    [ < PARAMNAME= name2 VALUE= value2>      ]
    .....
    .....
    [ Text to be displayed in the absence of Java]
<!APPLET>
```

---

The various attributes shown inside [ ] indicate the options that can be used when integrating an applet into a Web page. Note that the minimum required attributes are:

CODE = AppletFileName.class  
 WIDTH = pixels  
 HEIGHT = pixels

Table 14.1 lists all the attributes and their meaning.

**Table 14.1 Attributes of APPLET Tag**

Attribute	Meaning
CODE=AppletFileName.class	Specifies the name <i>of</i> the applet class to be loaded. That is, the name <i>of</i> the already-compiled .class file in which the executable Java bytecode for the applet is stored. This attribute must be specified.
CODEBASE=codebase URL (Optional)	Specifies the URL <i>of</i> the directory in which the applet resides. If the applet resides in the same directory as the HTML file, then the CODEBASE attribute may be omitted entirely.
WIDTH=pixels HEIGHT=pixels	These attributes specify the width and height <i>of</i> the space on the HTML page that will be reserved for the applet.
NAME=applet_instance_name (Optional)	A name for the applet may optionally be specified so that other applets on the page may refer to this applet. This facilitates inter-applet communication.
ALIGN=alignment (Optional)	This optional attribute specifies where on the page the applet will appear. Possible values for alignment are: TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABSMIDDLE, ABSBOTTOM, TEXTTOP, and BASELINE.
HSPACE=pixels (optional)	Used only when ALIGN is set to LEFT or RIGHT, this attribute specifies the amount <i>of</i> horizontal blank space the browser should leave surrounding the applet.
VSPACE=pixels (Optional)	Used only when some vertical alignment is specified with the ALIGN attribute (TOP, BOTTOM, etc.,) VSPACE specifies the amount <i>of</i> vertical blank space the browser should leave surrounding the applet.
ALT=alternate_text (Optional)	Non-Java browsers will display this text where the applet would normally go. This attribute is optional.

We summarise below the list of things to be done for adding an applet to a HTML document:

1. Insert an <APPLET>tag at an appropriate place in the Web page.
2. Specify the name *of* the applet's .class file.
3. If the .class file is not in the current directory, use the code base parameter to specify
  - the relative path if file is on the local system, or
  - the Uniform Resource Locator (URL) of the directory containing the file if it is on a remote computer.
4. Specify the space required for display of the applet in terms of width and height in pixels.
5. Add any user-defined parameters using <PARAM> tags.
6. Add alternate HTML text to be displayed when a non-Java browser is used.
7. Close the applet declaration with the </ APPLET> tag.



## PASSING PARAMETERS TO APPLETS

We can supply user-defined parameters to an applet using `<PARAM..>` tags. Each `<PARAM..>` tag has a name attribute such as *color*, and a value attribute such as red. Inside the applet code, the applet can refer to that parameter by name to find its value. For example, we can change the colour of the text displayed to red by an applet by using a `<PARAM..>` tag as follows:

```
.<APPLET.....>
<PARAMNAME= "color"    VALUE= "red">
</APPLET>
```

Similarly, we can change the text to be displayed by an applet by supplying new text to the applet through a `<PARAM..>` tag as shown below:

```
<PARAMNAME= "text"    VALUE= "I love Java">
```

Passing parameters to an applet code using `<PARAM>` tag is something similar to passing parameters to the `main( )` method using command line arguments. To set up and handle parameters, we need to do two things:

1. Include appropriate `<PARAM...>` tags in the HTML document.
2. Provide code in the applet to parse these parameters.

Parameters are passed to an applet when it is loaded. We can define the `init( )` method in the applet to get hold of the parameters defined in the `<PARAM>` tags. This is done using the `getParameter( )` method, which takes one string argument representing the *name* of the parameter and returns a string containing the *Value* of that parameter.

Program 14.2 shows another version of HelloJava applet. Compile it so that we have a class file ready.

---

### Program 14.2 Applet HelloJavaParam

---

```
import java.awt.*;
import java.applet.*;

public class HelloJavaParam extends Applet
{
    String str;

    public void init()
    {
        str = getParameter("string");           // Receiving parameter value
        if (str == null)
            str = "Java";
        str = "Hello" + str;                  // Using the value
    }
}
```

(Continued)

---

*Program 14.2 (Continued)*

---

```
public void paint (Graphics g)
{
    g.drawString(str, 10, 100);
}
}
```

---

Now, let us create HTML file that contains this applet. Program 14.3 shows a Web page that passes a parameter whose NAME is "string" and whose VALUE is "Applet!" to the applet HelloJ avaParam.

---

**Program 14.3 The HTML file for HelloJavaParam applet**

---

```
<html>
    <! Parameterized HTML file>
    <head>
        <title> Welcome to Java Applets </title>
    </head>
    <body>
        <applet code = HelloJavaParam.class
                width = 400
                height = 200>
            <param name = "string"
                  value = "Applet!">
        </applet>
    </body>
</html>
```

---

Save this file as HelloJavaParam.html and then run the applet using the applet viewer as follows:

```
appletviewer HelloJavaParam.html
```

This will produce the result as shown in Fig. 14.8.

Now, remove the <PARAM> tag from the HTMLfile and then run the applet again. The result will be as shown in Fig. 14.9.

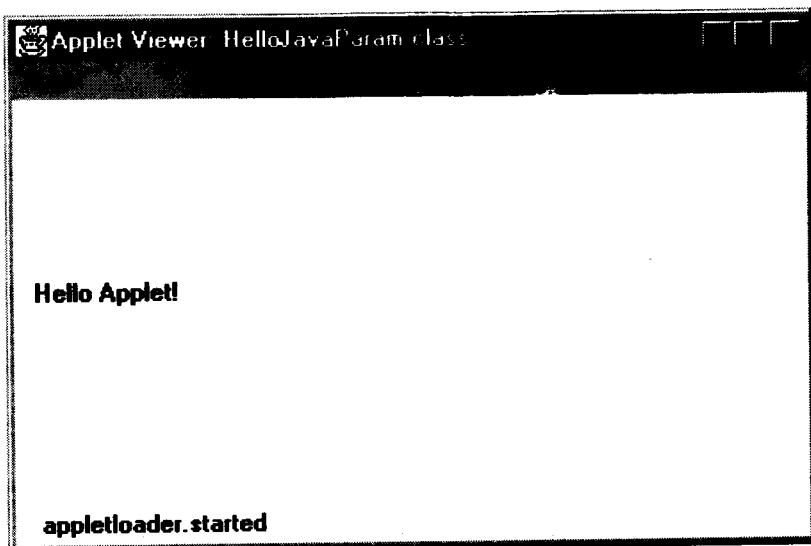
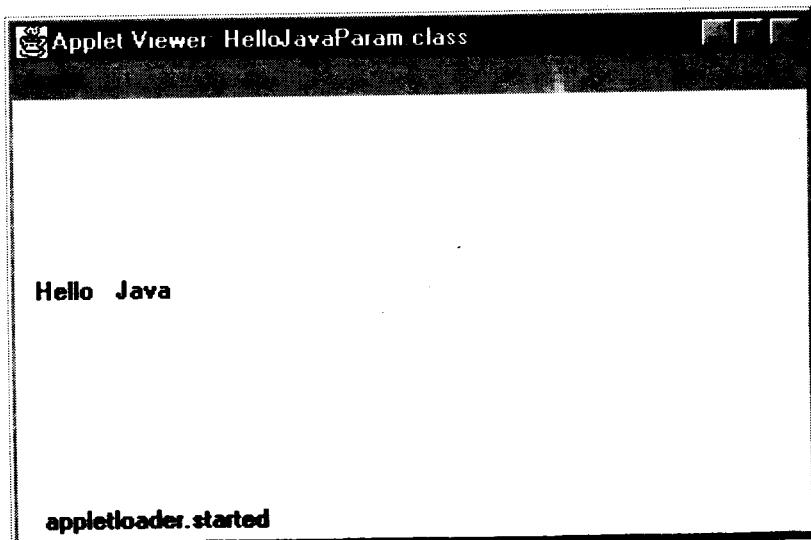


### ALIGNING THE DISPLAY

We can align the applet space using the ALIGN attribute. This attribute can have one of the nine values:

---

LEFT, RIGHT, TOP, TEXTTOP, MIDDLE,ABSMIDDLE,BASELINE,BOTTOM, ABSBOTTOM.

**Fig. 14.8***Displays of HelloJavaParam applet***Fig. 14.9***Output without PARAM tag*

For example, ALIGN = LEFT will display the applet space at the left margin of the page. All text that follows the ALIGN in the Web page will be placed to the right of the display. Program 14.4 shows a HTML file for our HelloJava applet shown in Program 14.1.

#### **Program 14.4 HTML file with ALIGN attribute**

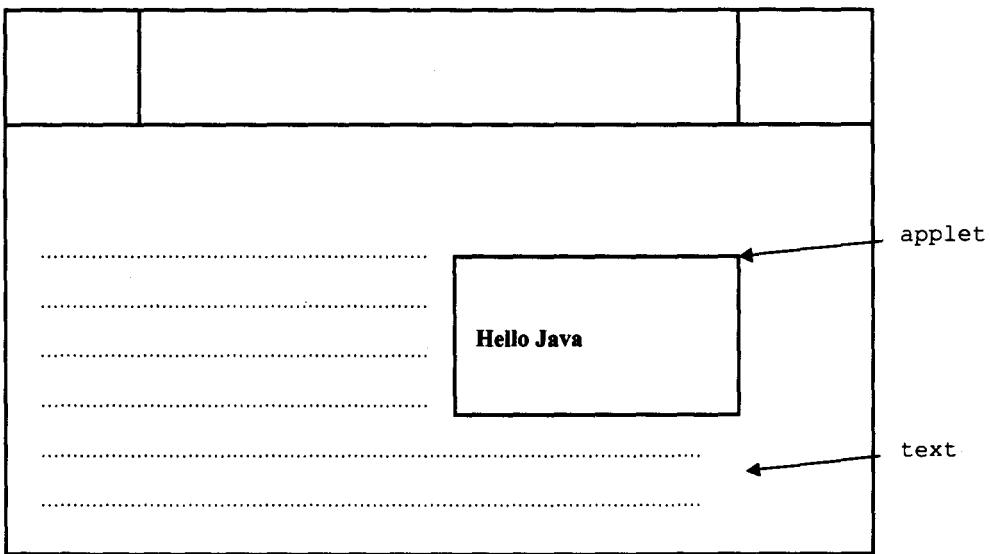
---

```
<HTML>
  <HEAD>
    <TITLE> Here is an applet </TITLE>
  </HEAD>
  <BODY>
    <APPLET CODE      = HelloJavaeClass
             WIDTH     = 400
             HEIGHT    = 200
             ALIGN     = RIGHT >
    </APPLET>
  </BODY>
</HTML>
```

---

The alignment of applet will be seen and appreciated only when we run the applet using a Java-capable browser. Figure 14.10 shows how an applet and text surrounding it might appear in a Java-capable browser. All the text following the applet appears to the left of that applet.

**Fig. 14.10**



**An applet aligned right**



## MORE ABOUT HTML TAGS

We have seen and used a few HTML tags. HTML supports a large number of tags that can be used to control the style and format of the display of Web pages. Table 14.2 lists important HTML tags and their functions.

**Table 14.2 HTML Tags and Their Functions.**

Tag	Function
<HTML> ... </HTML>	Signifies the beginning and end of a HTML file.
<HEAD> ... </HEAD>	This tag may include details about the Web page. Usually contains <TITLE> tag within it.
<TITLE> ... </TITLE>	The text contained in it will appear in the title bar of the browser.
<BODY> ... </BODY>	This tag contains the main text of the Web page. It is the place where the <APPLET> tag is declared.
<H1> ... </H1> ... <H6> ... </H6>	Header tags. Used to display headings. <H1> creates the largest font header, while <H6> creates the smallest one.
<CENTER> .. </CENTER>	Places the text contained in it at the centre of the page.
<APPLET ... >	<APPLET ... > tag declares the applet details as its attributes.
<APPLET> ... </APPLET>	May hold optionally user-defined parameters using <PARAM> Tags.
<PARAM ... >	Supplies user-defined parameters. The <PARAM> tag needs to be placed between the <APPLET> and </APPLET> tags. We can use as many different <PARAM> tags as we wish for each applet.
<B> ... </B>	Text between these tags will be displayed in bold type.
 	Line break tag. This will skip a line. Does not have an end tag.
<P>	Para tag. This tag moves us to the next line and starts a paragraph of text. No end tag is necessary.
<IMG ... >	This tag declares attributes of an image to be displayed.
<HR>	Draws a horizontal rule.
<A ... > </A>	Anchor tag used to add hyperlinks.
<FONT ... > ... </FONT>	We can change the colour and size of the text that lies in between <FONT> and </FONT> tags ~ COLOR and SIZE attributes in the tag <FONT ... >.
<! ... >	Any text starting with a <!> mark and ending with a > mark is ignored by the Web browser. We may add comments here. A comment tag may be placed anywhere in a Web page.

## **14.15 DISPLAYING NUMERICAL VALUES**

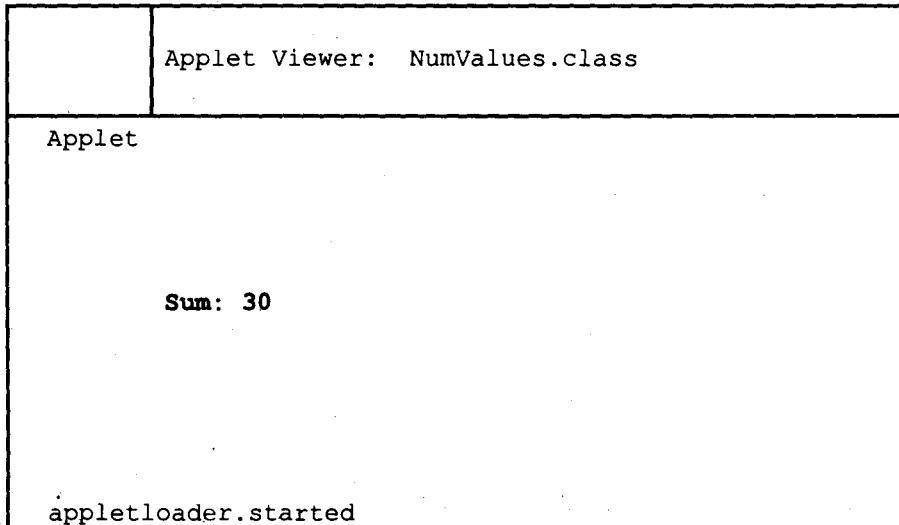
In applets, we can display numerical values by first converting them into strings and then using the `drawString( )` method of `Graphics` class. We can do this easily by calling the `ValueOf( )` method of `String` class. Program 14.5 illustrates how an applet handles numerical values.

**Program 14.5 Displaying numerical values**

```
import java.awt.*;
import java.applet.*;

public class NumValues extends Applet
{
    public void paint (Graphics g)
    {
        int value1 = 10;
        int value2 = 20;
        int sum = value1 + value2;
        String s = "sum: " + String.valueOf(sum);
        g.drawString(s, 100, 100);
    }
}
```

**Fig 14.11**

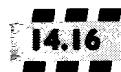


**Output of Program 14.5**

The applet Program 14.5 when run using the following HTML file displays the output as shown in Fig. 14.11.

```
<html>
<applet
    code  = NumValues.class
    width = 300
    height = 300 >
</applet>
</html>
```

---



## GETTING INPUT FROM THE USER

Applets work in a graphical environment. Therefore, applets treat inputs as text strings. We must first create an area of the screen in which user can type and edit input items (which may be any data type). We can do this by using the `TextField` class of the `applet` package. Once text fields are created for receiving input, we can type the values in the fields and edit them, if necessary.

Next step is to retrieve the items from the fields for display of calculations, if any. Remember, the text fields contain items in string form. They need to be converted to the right form, before they are used in any computations. The results are then converted back to strings for display. Program 14.6 demonstrates how these steps are implemented.

### **Program 14.6 Interactive input to an applet**

---

```
import java.awt.*;
import java.applet.*;

public class Userln extends Applet
{
    TextField  = text1, text2;

    public void init( )
    {
        text1 = new TextField(8);
        text2 = new TextField(8);
        add (text1);
        add (text2);
        text1.setText ("0");
        text2.setText ("0");
    }

    public void paint (Graphics g)
    {
```

---

(Continued)

---

**• 14.6 (Continued)**

```
int x=0,y=0,z=0;
String s1,s2,s;
g.drawString("Enter a number in each box ",10,50);
try
{
    s1 = text1.getText();
    x = Integer.parseInt(s1);
    s2 = text2.getText();
    y = Integer.parseInt(s2);
}
catch (Exception ex) { }
z = x + y;
s = String.valueOf(z);
g.drawString("THE SUM IS: " . ,10,75);
g.drawString(s,100,75);
}

public boolean action(Event event, Object obj)
{
    repaint();
    return true;
}
}
```

---

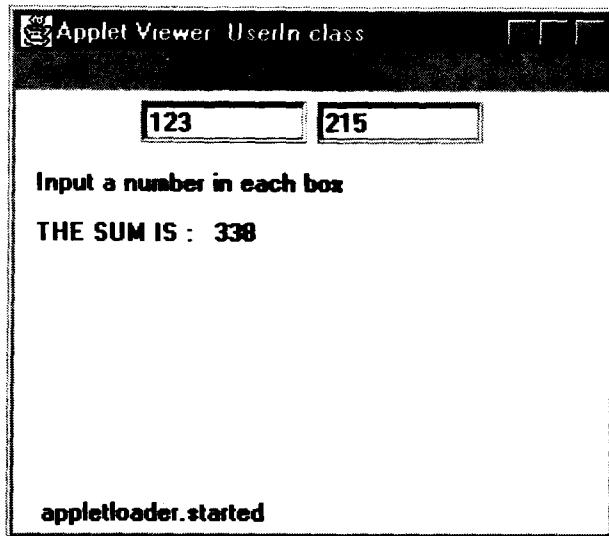
Run the applet UserIn using the following steps:

1. Type and save the program (.java file)
2. Compile the applet (.class file)
3. Write a HTML document (.html file)

```
<html>
<applet
    code = NumValues.class
    width = 300
    height = 200 >
</applet>
</html>
```

4. Use the appletviewer to display the results

When the applet is up and running, enter a number into each text field box displayed in the applet area, and then press the Return Key. Now, the applet computes the sum of these two numbers and displays the result as shown in Fig. 14.12.

**Fig. 14.12**

### ***Interactive computing with applets***

#### **Program Analysis**

The applet declares two `TextField` objects at the beginning.

```
TextField text1, text2;
```

These two objects represent text boxes where we type the numbers to be added.

Next, we override the `init( )` method to do the following:

1. To create two text field objects to hold strings (of eight character length).
2. To add the objects to the applet's display area.
3. To initialize the contents of the objects to zero.

Then comes the `paint( )` method where all the actions take place. First, three integer variables are declared, followed by three string variables. Remember, the numbers entered in the text boxes are in string form and therefore they are retrieved as strings using the `getText( )` method and then they are converted to numerical values using the `parseInt( )` method of the `Integer` class.

After retrieving and converting both the strings to integer numbers, the `paint( )` method sums them up and stores the result in the variable `z`. We must convert the numerical value in `z` to a string before we attempt to display the answer. This is done using the `ValueOf( )` method of the `String` class.

Applets are Java programs developed for use on the Internet. They provide a means to distribute interesting, dynamic, and interactive applications over the World Wide Web. We have learned the following about applets in this chapter:

- How do applets differ from applications,
- How to design applets,
- How to design a Web page using HTML tags,
- How to execute applets, and
- How to provide interactive input to applets.

**KEY TERMS**

**Applet, Local applet, Remote applet, Web page, HTML tag, APPLET tag, Applet life cycle.**

**REVIEW QUESTIONS**

---

- 14.1 What is an applet?
- 14.2 What is a local applet?
- 14.3 What is a remote applet?
- 14.4 Explain the client/server relationship as applied to Java applets.
- 14.5 How do applets differ from application programs?
- 14.6 Discuss the steps involved in developing and running a local applet.
- 14.7 Discuss the steps involved in loading and running a remote applet.
- 14.8 Describe the various sections of a Web page.
- 14.9 How many arguments can be passed to an applet using <PARAM> tags?
- 14.10 Why do applet classes need to be declared as public?
- 14.11 Describe the different stages in the life cycle of an applet. Distinguish between init() and start() methods.
- 14.12 Develop an applet that receives three numeric values as input from the user and then displays the largest of the three on the screen. Write a HTML page and test the applet.

## **Chapter 15**



**15.1**

### **INTRODUCTION**

One of the most important features of Java is its ability to draw graphics. We can write Java applets that draw lines, figures of different shapes, images, and text in different fonts and styles. We can also incorporate different colours in display.

Every applet has its own area of the screen known as *canvases*, where it creates its display. The size of an applet's space is decided by the attributes of the `<APPLET ... >` tag. A Java applet draws graphical images inside its space using the coordinate system as shown in Fig. 15.1.

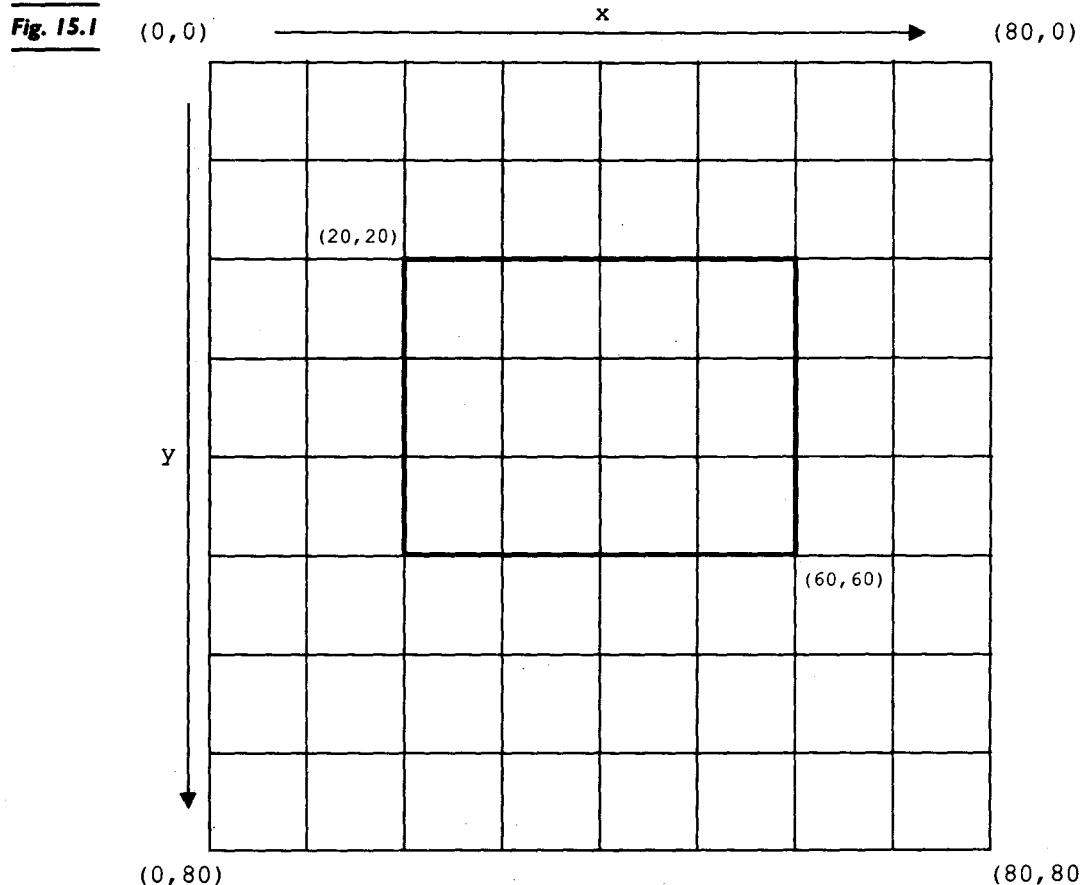
Java's coordinate system has the origin (0,0) in the upper-left corner. Positive x values are to the right, and positive y values are to the bottom. The values of coordinates x and y are in pixels.

**15.2**

### **THE GRAPHICS CLASS**

Java's Graphics class includes methods for drawing many different types of shapes, from simple lines to polygons to text in a variety of fonts. We have already seen how to display text using the `paint( )` method and a Graphics object.

To draw a shape on the screen, we may call one of the methods available in the Graphics class. Table 15.1 shows the most commonly used drawing methods in the Graphics class. All the drawing methods have arguments representing end points, corners, or starting locations of a shape as values in the applet's coordinate system. To draw a shape, we only need to use the appropriate method with the required arguments.

***Coordinate system of java*****Table 15.1 Drawing Methods of the Graphics Class**

Method	Description
clearRect( )	Erases a rectangular area of the canvas.
copyArea( )	Copies a rectangular area of the canvas to another area.
drawArc( )	Draws a hollow arc.
drawLine( )	Draws a straight line.
drawOval( )	Draws a hollow oval.
drawPolygon( )	Draws a hollow polygon.

(Continued)

Table 15.1 (*Continued*)

Method	Description
drawRect( )	Draws a hollow rectangle.
drawRoundRect( )	Draws a hollow rectangle with rounded corners.
drawString( )	Displays a text string.
fillArc( )	Draws a filled arc.
fillOval( )	Draws a filled oval.
fillPolygon( )	Draws a filled polygon.
fillRect( )	Draws a filled rectangle.
fillRoundRect( )	Draws a filled rectangle with rounded corners.
getColor( )	Retrieves the current drawing colour.
getFont( )	Retrieves the currently used font.
getFontMetrics( )	Retrieves information about the current font.
setColor( )	Sets the drawing colour.
setFont( )	Sets the font.

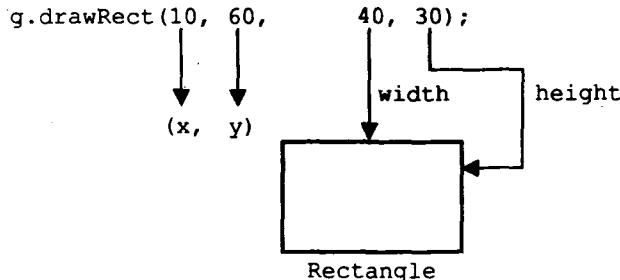
## LINES AND RECTANGLES

The simplest shape we can draw with the Graphics class is a line. The drawLine( ) method takes two pairs of coordinates, (x1, y1) and (x2, y2) as arguments and draws a line between them. For example, the following statement draws a straight line from the coordinate point (10,10) to (50,50) :

```
g.drawLine(10,10, 50,50);
```

The g is the Graphics object passed to paint( ) method.

Similarly, we can draw a rectangle using the drawRect( ) method. This method takes four arguments. The first two represent the x and y coordinates of the top left corner of the rectangle, and the remaining two represent the width and the height of the rectangle. For example, the statement



will draw a rectangle starting at (10,60) having a width of 40 pixels and a height of 30 pixels. Remember that the drawRect() method draws only the outline of a box.

We can draw a solid box by using the method fillRect(). This also takes four parameters (as drawRect) corresponding to the starting point, the width and the height of the rectangle. For example, the statement

```
g.fillRect{60,10,30,80};
```

will draw a solid rectangle starting at (60,10) with a width of 30 pixels and a height of 80 pixels.

We can also draw rounded rectangles (which are rectangles with rounded edges), using the methods drawRoundRect() and fillRoundRect(). These two methods are similar to drawRect() and fillRect() except that they take two extra arguments representing the width and height of the angle of corners. These extra parameters indicate how much of corners will be rounded. Example:

```
g.drawRoundRect{10,      100,   80,   50,   10,   10} ;  
g.fillRoundRect{20,      110,   60,   30,   5,     5} ;
```

Program 15.1 is an applet code that draws three lines, a rectangle, a filled rectangle, a rounded rectangle and a filled rounded rectangle. Note that the filled rounded one is drawn inside the rounded rectangle. Program 15.2 shows a HTML file that displays the applet. The output of the applet LineRect under appletviewer is shown in Fig. 15.2.

---

### **Program 15.1 Drawing lines and rectangles**

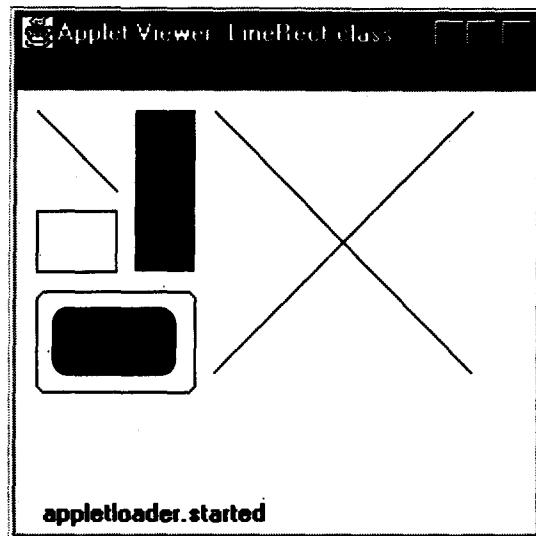
---

```
import  java.awt.*;  
import  java.applet.*;  
  
public  class  LineRect  extends  Applet  
{  
    public  void  paint  {Graphics  g)  
    {  
        g.drawLine{10,      10,   50,   50} ;  
        g.drawRect{10,      60,   40,   30} ;  
        g. fillRect  (60,  10,  30,  80) ;  
        g.drawRoundRect{10,      100,   80,   50,   10,   10} ;  
        g.fillRoundRect{20,110,      60,   30,   5,     5} ;  
        g. drawLine  (100,  10,  230,  140) ;  
        g.drawLine{100,      140,  230,  10) ;  
    }  
}
```

## **Program 15.2 LineRect.html file**

```
<APPLET  
    CODE = LineRect.class  
    WIDTH = 250  
    HEIGHT = 200>  
</APPLET>
```

**Fig. 15.2**



**Output of LineRect applet**

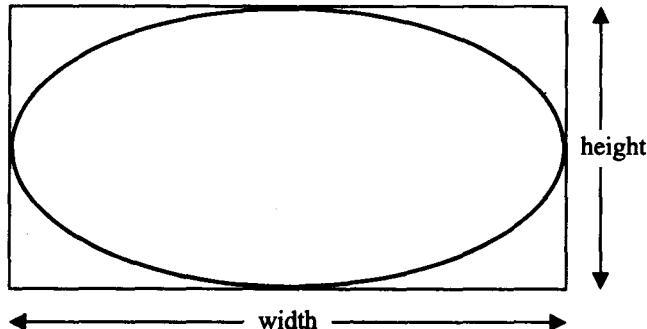


## **CIRCLES AND ELLIPSES**

The Graphics class does not have any method for circles or ellipses. However, the drawOval( ) method can be used to draw a circle or an ellipse. Ovals are just like rectangles with overly rounded corners as shown in Fig. 15.3. Note that the figure is surrounded by a rectangle that just touches the edges. The drawOval( ) method takes four arguments: the first two represent the top left corner of the imaginary rectangle and the other two represent the width and height of the oval itself. Note that if the width and height are the same, the oval becomes a circle. The oval's coordinates are actually the coordinates of an enclosing rectangle;

Like rectangle methods, the drawOval( ) method draws outline of an oval, and the fillOval( ) method draws a solid oval. The code segment shown below draws a filled circle within an oval (see Fig. 15.4).

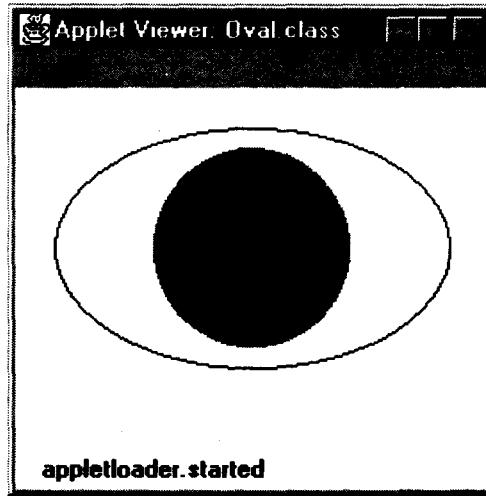
**Fig. 15.3**



**Oval within an imaginary rectangle**

```
public void paint(Graphics g)
{
    g.drawOval(20, 20, 200, 120);
    g.setColor(Color.green);
    g.fillOval(70, 30, 100, 100); // This is a circle.
}
```

**Fig. 15.4**



**A filled circle within an ellipse**

We can draw an object using a color object as follows:

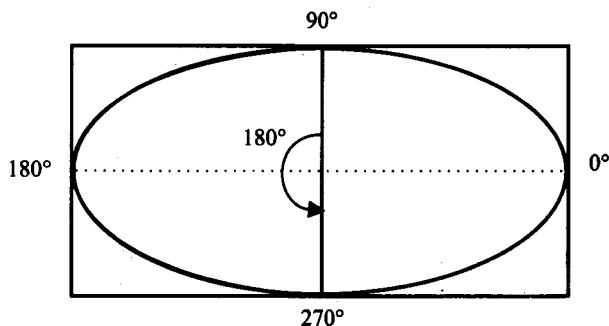
```
g.setColor(Color.green);
```

After setting the color, all drawing operations will occur in that color.

An arc is a part of an oval. In fact, we can think of an oval as a series of arcs that are connected together in an orderly manner. The `drawArc()` designed to draw arcs takes six arguments. The first four are the same as the arguments for `drawOval( )` method and the last two represent the starting angle of the arc and the number of degrees (sweep angle) around the arc.

In drawing arcs, Java actually formulates the arc as an oval and then draws only a part of it as dictated by the last two arguments. Java considers the three O'clock position as zero degree position and degrees increase in anti-clockwise direction as shown in Fig. 15.5. So, to draw an arc from 12:00 O'clock position to 6:00 O'clock position, the starting angle would be 90, and the sweep angle would be 180.

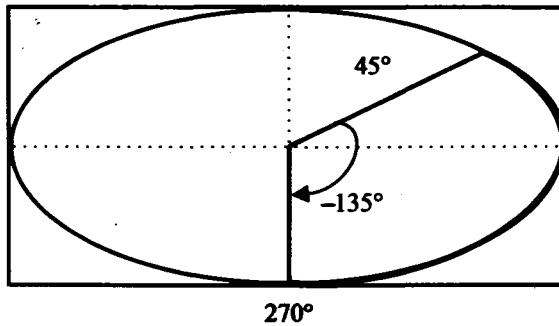
**Fig. 15.5**



**Arc as a part of an oval**

We can also draw an arc in backward direction by specifying the sweep angle as negative. For example, if the last argument is  $-135^\circ$  and the starting angle is  $45^\circ$ , then the arc is drawn as shown in Fig. 15.6.

**Fig. 15.6**



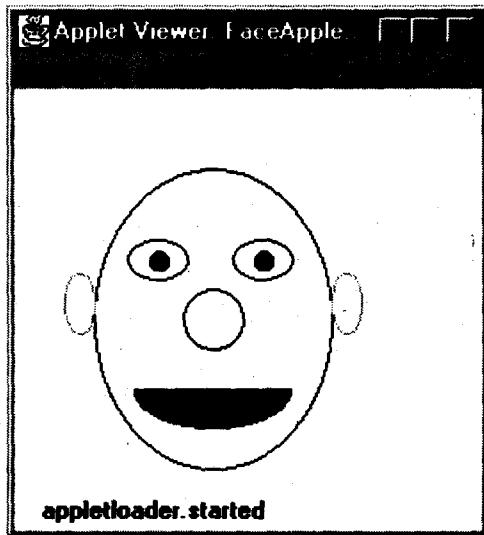
**Drawing an arc in clockwise direction**

We can use the `fillArc( )` method to fill the arc. Filled arcs are drawn as if they were sections of a pie. Instead of joining the two end points, they are joined to the centre of the oval. Program 15.3 shows an applet that draws a human face as shown in Fig. 15.7.

### **Program 15.3 Applet for drawing a human face**

```
import java.awt.*;
import java.applet.*;
public class Face extends Applet
{
    public void paint (Graphics g)
    {
        g.drawOval(40, 40, 120, 150);           II Head
        g.drawOval (57, 75, 30, 20);           II Left eye
        g.drawOval (110, 75, 30, 20);          II Right eye
        g.fillOval (68, 81, 10, 10);          II Pupil (left)
        g.fillOval (121, 81, 10, 10);         II Pupil (right)
        g.drawOval(85, 100, 30, 30);          II Nose
        g.fillArc(60,125,80,40,180,180);     II Mouth
        g.drawOval(25, 92, 15, 30);          II Left ear
        g.drawOval(160, 92, 15, 30);         II Right ear
    }
}
```

**Fig. 15.7**



**Output of the Face applet**

Polygons are shapes with many sides. A polygon may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second is the beginning of the third, and so on. This suggests that we can draw a polygon with n sides using the drawLine( ) method n times in succession. For example, the code given below will draw a polygon of three sides. The output of this code is shown in Fig. 15.8.

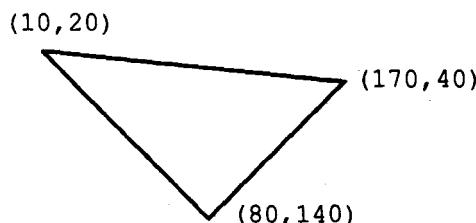
---

```
public void paint(Graphics g)
{
    g.drawLine(10, 20, 170, 40);
    g.drawLine(170, 40, 80, 140);
    g.drawLine(80, 140, 10, 20);
}
```

---

Note that the end point of the third line is the same as the starting point of the polygon.

**Fig. 15.8**



**A polygon with three sides**

We can draw polygons more conveniently using the drawPolygon( ) method of Graphics class. This method takes three arguments:

- An array of integers containing x coordinates
- An array of integers containing y coordinates
- An integer for the total number of points

It is obvious that x and y arrays should be of the same size and we must repeat the first point at the end of the array for closing the polygon. The polygon shown in Fig. 15.8 can be drawn using the drawPolygon() method as follows;

---

```
public void paint (Graphics g)
{
    int xPoints[ ] = {10, 170, 80, 10};
    int yPoints[ ] = {20, 40, 140, 20};
    int npoints [ ] = xPoints.length;
```

```
        g.drawPolygon      (xpoints,      ypoints,      nPoints);
    }
```

---

We can also draw a filled polygon by using the `fillPolygon( )` method. Program 15.4 illustrates the use of polygon methods to draw both the empty polygons and the filled polygons. Its output is shown in Fig. 15.9.

#### **Program 15.4 Drawing polygons**

---

```
import  java.awt.*;
import  java.applet.*;

public  class  Poly  extends  Applet
{
    int  x1[  ] = {20,  120,  220,  20};
    int  y1[  ] = {20,  120,  20,  20};
    int  n1 = 4;
    int  x2[  ] = {120,  220,  220,  120};
    int  y2[  ] = {120,  20,  220,  120};
    int  n2 = 4;

    public  void  paint  (Graphics  g)
    {
        g.drawPolygon(x1,      y1,  n1);
        g.fillPolygon(x2,      y2,  n2);
    }
}
```

---

Another way of calling the methods `drawPolygon( )` and `fillPolygon( )` is to use a `Polygon` object. The `Polygon` class enables us to treat the polygon as an object rather than having to deal with individual arrays. This approach involves the following steps:

1. Defining x coordinate values as an array
2. Defining y coordinate values as an array
3. Defining the number of points n
4. Creating a `Polygon` object and initializing it with the above x, y and n values.
5. Calling the method `drawPolygon( )` or `fillPoIygon( )` with the `Polygon` object as arguments

The following code illustrates these steps:

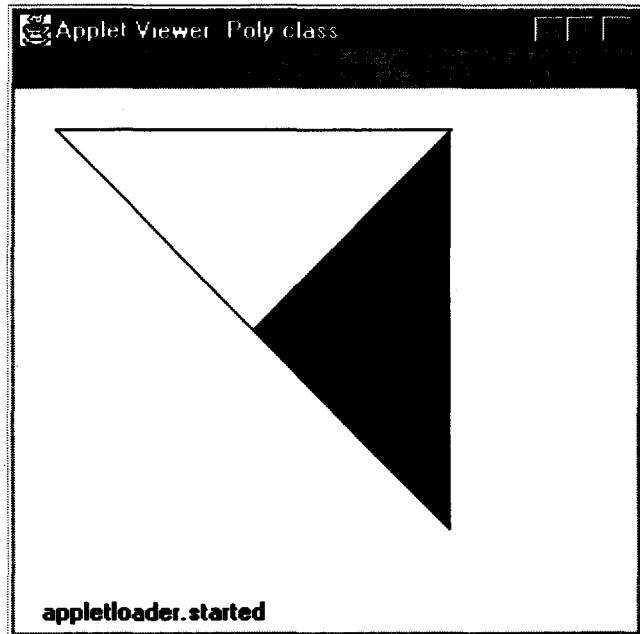
---

```
public  void  paint  (Graphics  g)
{
    int  x[  ] = {20,  120,  220,  20};
```

```

int y[ ] = {20, 120, 20, 20};
int n = x.length;
Polygon poly = new Polygon(x,y,n);
g.drawPolygon (poly);
}

```

**Fig. 15.9*****Output of Program 15.4***

The `Polygon` class is useful if we want to add points to the polygon. For example, if we have a `Polygon` object existing, then we can add a point to it as follows:

```
poly.addPoint(x,y);
```

The above code may be rewritten using the `addPoint( )` method as follows:

---

```

public void paint (Graphics g)
{
    Polygon poly = new POJ.ygon( );
    poly.addPoint(20, 20);
    poly.addPoint(120, 120);
    poly.addPoint(220, 20);
}

```

```

        poly.addPoint(20,    20);
        d.drawPolygon(poly);
    }
}

```

---

Here, we first "create" an empty polygon and then add points to it one after another. Finally, we call the drawPolygon( ) method using the poly object as an argument to complete the process of drawing the polygon.

## 15.7 LINE GRAPHS

We can design applets to draw line graphs to illustrate graphically the relationship between two variables. Consider the table of values shown as follows:

<b>X</b>	0	60	120	180	240	300	360	400
<b>Y</b>	0	120	180	260	340	340	300	180

The variation of Y, when X is increased can be shown graphically using the Program 15.5. The graph is drawn in an area 400 x 400 pixels. As we know, X increases to the right and Y increases downwards. This is different from the normal graphical systems that have their origin in the bottom left corner. In order to convert the Java coordinate system to the normal systems, we transform the Y values to N-Y where N is the height of the display area. For example, if the vertical height of the display area is 400 pixels, then the point (5, 5) would be (5, 395) in the new system. The table below shows the new values of X and Y and Program 15.5 gives the applet code that draws a line graph showing the relationship between X and Y. Figure 15.10 shows the graph.

<b>X</b>	0	60	120	180	240	300	360	420
<b>Y</b>	400	280	220	140	60	60	100	220

### **Program 15.5 Applet to draw a line graph**

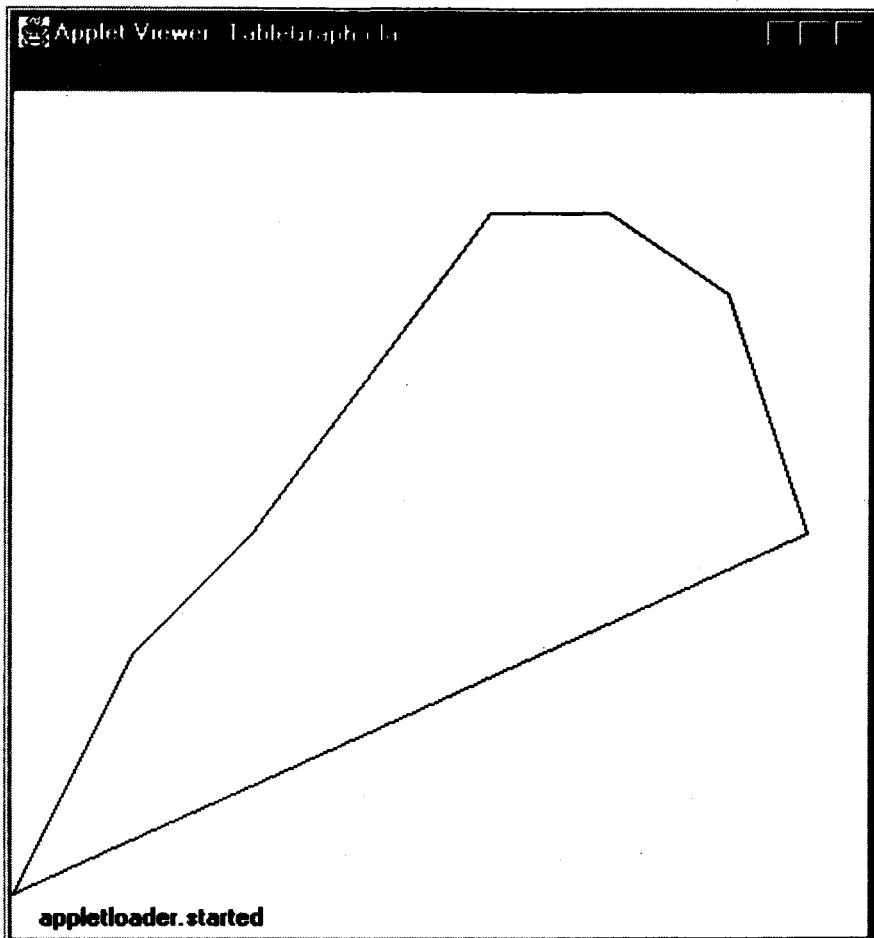
---

```

import java.awt.*;
import java.applet.*;

public class TableGraph extends Applet
{
    int x[ ] = {0, 60, 120, 180, 240, 300, 360, 400};
    int y[ ] = {400, 280, 220, 140, 60, 60, 100, 220};
    int n = x.length;
    public void paint(Graphics g)
    {
        g.drawPolygon(x,y,n);
    }
}

```

**Fig. 15.10**

A line graph

**15.8****USING CONTROL LOOPS IN APPLETS**

We can use all control structures in an applet. Program 15.6 uses a **for** loop for drawing circles repeatedly.

**Program 15.6 Using control loops in applets**

```
import java.awt.*;
import java.applet.*;
```

---

*(Continued)*

---

~ 15.6 (Continued)

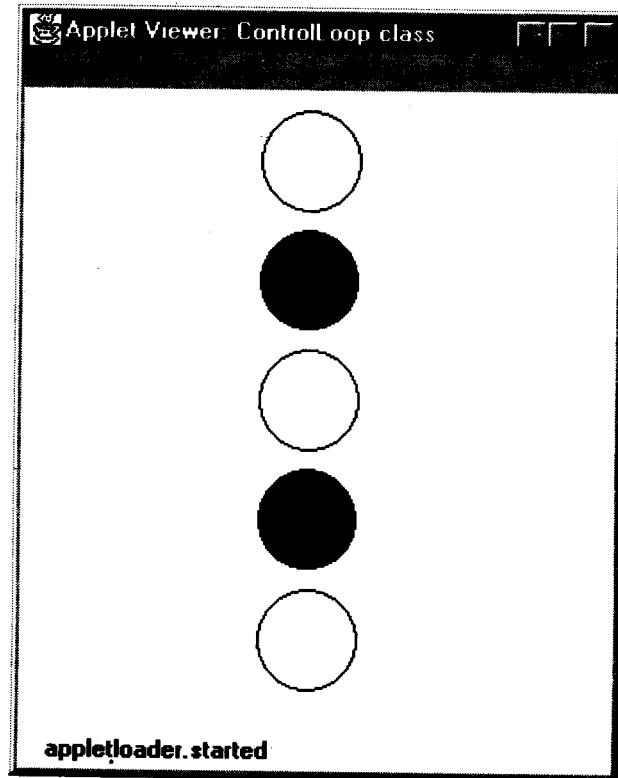
---

```
public class ControlLoop extends Applet
{
    public void paint (Graphics g)
    {
        for(int i=0; i<=4; i++)
        {
            if ((i%2) == 0)
                g.drawOval (120, i*60+10, 50, 50);
            else
                g.fillOval (120, i*60+10, 50, 50);
        }
    }
}
```

---

When we run the applet shown in Program 15.6, the for loop draws five circles as shown in Fig. 15.11.

**Fig. 15.11**



**Output of Program 15.6**

Applets can be designed to display bar charts, which are commonly used in comparative analysis of data. The table below shows the annual turnover of a company during the period 1991 to 1994. These values may be placed in a HTMLfile as PARAMattributes and then used in an applet for displaying a bar chart.

Year	1991	1992	1993	1994
Turnover (Rs. Crores)	110	150	100	170

Program 15.7 shows an applet that receives the data values from the HTMLpage shown in Program 15.8 and displays an appropriate bar chart. The method getParameter( ) is used to fetch the data values from the HTMLfile. Note that the method getParameter() returns only string values and therefore we use the wrapper class method parseInt( ) to convert strings to integer values. Figure 15.12 shows the result of Program 15.7.

### **Program 15.7 An applet for *drawing* bar charts**

---

```
import java.awt.*;
import java.applet.*;
public class BarChart extends Applet
{
    int n = 0;
    String label[];
    int value [ 1];
    public void init()
    {
        try
        {
            n = Integer.parseInt(getParameter("columns"));
            label = new String[n];
            value = new int[n];
            label[0] = getparameter("label1");
            label[1] = getParameter("label2");
            label[2] = getParameter("label3");
            label[3] = getParameter("label4");
            value [0] = Integer.parseInt (getparameter ("cl"));
            value[1] = Integer.parseInt(getParameter("fc2"));
            value[2] = Integer.parseInt(getParameter("fc3"));
        }
    }
}
```

*(Continued)*

,, \*\*\* ...".15.7 (Continued)

---

```
        value[3] = Integer.parseInt(getParameter("c4"))i
    }
    catch (NumberFormatException e). { }
}
public void paint (Graphics g)
{
    for(int i = 0; i < ni i++)
    {
        g.setColor(Color.red)i
        g.drawString(label[i], 20, i*50+30)i
        g.fillRect(50,i*50+10,value[i],40);
    }
}
```

---

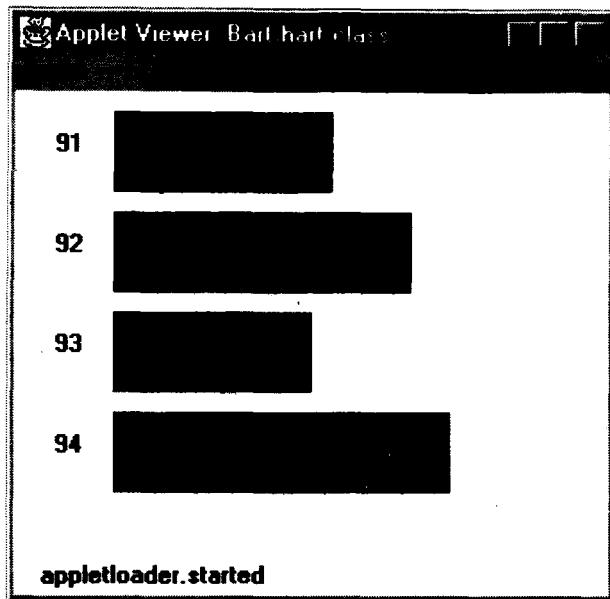
**Pro,ram 15.' HTML file for runnln, the BarChart applet**

---

```
<HTML>
<APPLET
    CODE  = BarChart.class
    WIDTH = 300
    HEIGHT = 250>

<PARAM NAME = "columns"      VALUE      "" "4">
<PARAM NAME = "c1"      VALUE      "" "110">
<PARAM NAME = "c2"      VALUE      •.• "150">
<PARAM NAME = "c3"      VALUE      = "100">
<PARAM NAME = "c4"      VALUE      = "170">
<PARAM NAME = "label1"    VALUE      •.. "91">
<PARAM NAME = "label2"    .VALUE      = "92"> .
<PARAM NAME = "label3"    VALUE      = "93">
<PARAM NAME = "label4"    VALUE      - "94">

</APPLET>
</HTML>
```

**Fig. 15.12****Barchart produced by Program 15.7**

## SUMMARY

Java's `Graphics` class supports many methods that enable us to draw many types of shapes, including lines, rectangles, ovals, and arcs. We can use these methods to enhance the appearance of outputs of applets, to draw frames around objects, and to put together simple illustrations. We have also seen how the graphics capability of Java can be used to draw line graphs and bar charts.

### KEY TERMS

Coordinate system, Graphics class, drawLine( ), drawRect( ), drawOval( ), drawArc( ), drawRoundRect( ), drawString( ), drawImage( ), Font, FontMetrics

### REVIEW QUESTIONS

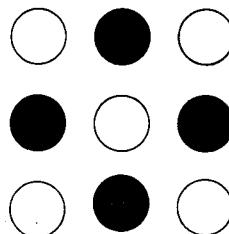
- 
- 15.1 How is Java's coordinate system organized?
  - 15.2 Describe the arguments used in the method `drawRoundRect()`.
  - 15.3 Explain the purpose of each argument used in the method `drawArc()`.

**15.4** Describe the three ways of drawing polygons.

**15.5** Write applets to draw the following shapes:

- (a) Cone
- (b) Cylinder
- (c) Cube
- (d) Square inside a circle
- (e) Circle inside a square

**15.6** Write a~ applet to display the following figure:

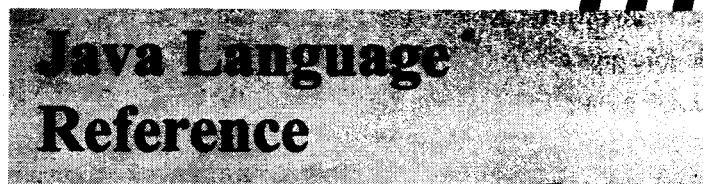


**15.7** Suggest possible improvements in the human face drawn by Program 15.3.

**15.8** Modify the Program 15.8 such that the display for each year would include the year, bar graph and the value represented by the bar as shown below



## **Appendix A**



### **Important Keywords**

---

Data-declaration keywords:

byte            int            float            char            double

Loop keywords:

do            while            for            break            continue

Conditional keywords:

if            else            switch

Exception keywords:

throw            try            catch

Structure keywords:

class            extends            interface            implements

Access keywords:

public            private            protected

---

### **Specifying Character Uterals**

---

Description or Escape Sequence	Sequen<=	Output
any character	'y'	y
backspace BS	"b"	backspace

Description or Escape Sequence	Sequence	Output
Horizontal tab HT	'\t'	tab
line feed LF	'\n'	linefeed
form feed FF	'\f'	form feed
carriage return CR	'\r'	carriareturn
double quote •	'\" ,	
single quote	'\' ,	
backslash	'\\'	\
octal bit pattern	'Oddd'	(octal value of ddd)
hex bit pattern	'Oxdd'	(hex value of dd)
Unicode character	'/dddd'	(actual Unicode character of dddd)

## Arithmetic Operators

Operator	Operation	Example
+	Addition	x+y
-	Subtraction	x-y
*	Multiplication	x*y
/	Division	x/y
%	Modulus	x%'y

## Assignment Operators

Operator	Operation	Example	Meaning
+=	add to current variable	x+=y	x=x+y
--	subtract from current variable	x-= y	x = x-y
*=	multiply by current variable	x*= y	x=x*y
/=	divide by current variable	x/=y	x = x /y

## Increment and Decrement Operators

Operator	Operation	Example	Meaning
++	increment by 1	x++	x=x+1
--	decrement by 1	x--	x = x-1

## Comparison Operators (return true or false)

Operator	Operation	Example	Meaning
<code>==</code>	Equal	<code>x==y</code>	Is x equal to y?
<code>!=</code>	Not 'equal	<code>x!=y</code>	Is x not equal to y?
<code>&lt;</code>	Less than	<code>x &lt; y--</code>	Is x less than y?
<code>&gt;</code>	Greater than	<code>x&gt;y</code>	Is x greater than y?
<code>&lt;=</code>	Less than or equal to	<code>x &lt;=y</code>	Is x less than or equal to y?
<code>&gt;=</code>	Greater than or equal to	<code>x &gt;=y</code>	Is x greater than or equal to y?

## Bitwise Operators

Operator	Operation
<code>&amp;</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>"</code>	Bitwise XOR
<code>&lt;&lt;</code>	Left shift
<code>&gt;&gt;</code>	Right shift
<code>&gt;&gt;&gt;</code>	Zero fill right shift
<code>-</code>	Bitwise complement
<code>&lt;&lt;=</code>	Left shift assignment
<code>&gt;&gt;=</code>	Right shift assignment
<code>&gt;&gt;&gt;=</code>	Zero fill right shift assignment
<code>x&amp;=y</code>	AND assignment
<code>x =y</code>	OR assignment
<code>x"=y</code>	NOT assignment

## Comment Indicators

Start	Text	End Comment
<code>I"</code>	text	<code>"I</code>
<code>I""</code>	text	<code>"I</code>
<code>II</code>	text	(everything to the end of the line is ignored by the compiler)

## ~ • Java Data Type Keywords

---

boolean char byte short int long float double-

---

## Integer Data Type Ranges

---

Type	Length	Minimum Value	Maximum Value
byte	8 bits	-128	127
Short	16 bits	-32768	32767
Int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807

---

## Unary Operators

---

Operator	Operation
-	Unary negation
-	Bitwise complement
++	Increment
--	Decrement
!	Not

---

## Operator Precedence

---

++	--	!	-	instanceof	*
/	%	+	-	<<	>>
>>	<	>	<=	>=	--
!=	&	^	&&		?:
=	op=				

---

## Control Statements

---

Statement	Example
A simple if statement	if (booleanTest) callfunction();

---

Statement	Example
A multiline if statement	<pre>if (booleanTest) {     // set of statements }</pre>
The if...else statement	<pre>if (booleanTest) {     // Trueblockstatements } else {     // Falseblockstatements }</pre>
The while statement	<pre>while (booleanTest) {     // Loopstatements }</pre>
The do...while loop	<pre>do {     // Loopstatements } while (booleanTest) ;</pre>
The switch statement	<pre>switch (expression) {     case FirstCase :         // First set of statements         break;      case SecondCase :         // Second set of statements         break;      case ThirdCase :         // Third set of statements         break;      default :         // Default statement         break; }</pre>
The for loop	<pre>for. (initialization;      condition;      increment)       statement;</pre>

## Defining Classes

---

The basic structure of defining a class is as follows:

```
Scope class ClassName [extends class]
{
    // Class implementation statements
}
```

When declaring the scope of the class, we have several options to control how other classes can access this class:

public	The class can be used by code outside of the file. Only one class in a file may have this scope. The file must be named with the class name followed by the four-letter .java extension.
private	The class can only be used within a file.
abstract	The class cannot be used by itself and must be subclassed.
final	The class cannot be used by a subclass.
synchronized	Instances of this class can be made arguments.

If a scope modifier is not used, the class is only accessible within the current file.

## Defining methods

---

A *method* is the code that acts on data inside a class and is always declared inside the class declaration. A method has the following syntax:

```
Scope ReturnType methodName(arguments)
{
    // Method implementation statements
}
```

The scope allows the programmer to control access to methods and can be one of the following:

public	The method is accessible by any system object.
protected	The method is only accessible by subclasses and the class in which it is declared.
private	The method is accessible only within current class.
final	The method cannot be overridden by any subclass.
static	The method is shared by all instances of the class.

If a method is not given a scope, it is only accessible within the scope of the current file. We can also use these scope operators when declaring variables.

## Exception- Handling

---

An exception has two parts: signalling an exception and setting up an exception handler. To signal an exception, use the try keyword. To set up an exception handler, we use the catch keyword. We use the finally keyword to specify a block of statements that will execute no matter what. To tell the system that an error has occurred, use the throw keyword.

```
try
{
    // Try this block of code and throw exception
}

catch  (Exception   e)
{
    // Handle error
}

finally
{
    // Executed no matter what happens
}
```

## General Applet Construction

---

A minimal Java Applet has the following construction:

```
/*
 *      JavaApplet.java      - Sample Applet
 *
 */

import  java.applet.*;
import  java.awt.Graphics;

public  class  JavaApplet  extends  java.applet.Applet
{

    public  void  init(    )
    {
        // Called first time applet is executed
    }

    public  void  start(    )
    {
        // Called after init() and whenever Web page is revisited
    }
}
```

```
public void stop( )
{
    // Called when Web page disappears
}

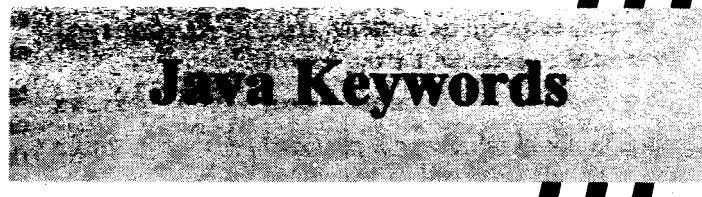
public void destroy( )
{
    // Called when applet is being removed from memory
}

public void paint (Graphics g)
{
    g.drawString("Goodbye ! ", 100, 100);
}

}
```

---

## **Appendix B**



This appendix lists the keywords in Java. They are grouped according to their meaning/function.

Group	Keyword	Meaning/Function
Class Organization	package	specifies the class in a particular source file should belong to the named package.
	import	requests the named class or classes be imported into the current application.
Class Definition	interface	defines global data and method signatures that can be shared among classes.
	class	defines a collection of related data behavior.
	extends	indicates which class to subclass.
	implements	indicates the interface for which a new class will supply methods.
Keywords for Classes and Variables	abstract	specifies the class cannot be instantiated directly.
	public	means the class, method, or variable can be accessed from anywhere.
	private	means only the class defining the method or variable can access it.

Group	Keyword	Meaning/Function
Simple Data Types	protected	means only the defining class and its subclasses can access the method or variable.
	static	specifies a class method or variable.
	synchronized	indicates only one object or class can access this variable or method at a time.
	volatile	tells the compiler this variable may change asynchronously due to threads.
	final	means this variable or method cannot be changed by subclasses.
	native	links a method to native code.
	long	is a 64-bit integer value.
	int	is a 32-bit integer value.
	short	is a 16-bit integer value.
	byte	is a <i>B-bit</i> integer value.
Values and Variables	double	is a 64-bit floating-point value.
	float	is a 32-bit floating-point value.
	char	is a 16-bit Unicode character.
	boolean	is a true or false value.
	void	indicates a method does not return a value.
Exception Handling	false	is a Boolean value.
	true	is a Boolean value.
	this	refers to the current instance in an instance method.
	super	refers to the immediate superclass in an instance method.
	null	represents a nonexistent instance.
	throw	throws an exception.
	throws	throws an exception.
	try	marks a stack so that if an exception is thrown, it will unwind to this point.

Group	Keyword	Meaning/Function
Instance Creating and Testing	catch	catches an exception.
	finally	says execute this block of code regardless of exception error handling flow.
Control Flow	new	creates new instances.
	instanceof	tests whether an instance derives from a particular class or interface.
Control Flow	switch	tests a variable.
	case	executes a particular block of code according to the value tested in the switch.
	default	means the default block of code executes if no matching case statement was found.
	break	breaks out of a particular block of code.
	continue	continues with the next iteration of a loop.
	return	returns from a method, optionally passing back a value.
	do	performs some statement or set of statements.
	if	tests for a condition and performs some action if true.
	else	performs some action if the above test was false.
	for	signifies iteration.
Not Used Yet, But Reserved	while	performs some action while a condition is true.
	byvalue	
	const	
	goto	
	cast	
	future	
	generic	

Group	Keyword	Meaning/Function
	threadsafe	
	<i>inner</i>	
	operator	
	outer	
	rest	
	var	

- **Keywords not available from C**

*auto, enum, extern, register, signed, sizeof, struct, typedef, union, unsigned.*

- **Keywords not available from C++**

*delete, friend, inline, mutable, template, using, virtual.*

## **Appendix C**

# **Differences Between Java and C/C++**

**C.1**

### **DATA TYPES**

- All Java primitive data types (char, int, short, long, byte, float, double and boolean) have specified sizes and behaviour that are machine-independent.
- Conditional expressions can only be boolean, not integral;
- Casting between data types is much more controlled in Java. Automatic conversion occurs only when there is no loss of information. All other casts must be explicit.
- Java supports special methods to convert values between class objects and primitive types.
- Composite data types are accomplished in Java using only classes. Structures and unions are not supported.
- Java does not support typedef and enum keywords.
- All non-primitive types can only be created using new operator.
- Java does not define the type modifiers auto, extern, register, signed, and unsigned.

**C.2**

### **POINTERS**

- Java does not support pointers. Similar functionality is accomplished by using implicit references to objects. Pointer arithmetic is not possible in Java.

**C.3**

### **OPERATORS**

- Java adds a new right shift operator `>>` which inserts zeros at the top end.
- The `+` operator can be used to concatenate strings.
- Operator overloading is not possible in Java.

- The `, operator` of C has been deleted.
- Java adds another operator `instanceof` to identify objects.
- The module division may be applied to float values in Java which is not permitted in C/C++.



## FUNCTIONS AND METHODS

- All functions are defined in the body of the class. There are no independent functions.
- The functions defined inside a class are known as methods.
- Although function overloading in Java works virtually identical to C++ function overloading, there are no default arguments to functions.
- No inline functions in Java.
- Java does not support variable-length argument lists to functions. All method definitions must have a specific number of arguments.
- Java requires that methods with no arguments must be declared with empty parenthesis, (or with `void` keyword).



## PREPROCESSOR

- Java does not have a preprocessor, and as such, does not support `#define` or macros.
- Constants can be created using the `final` modifier when declaring class and instance variables.
- Java programs do not use header files.



## CLASSES

- Class definitions take the similar form in Java as in C++, but there is no closing semicolon.
- There is no scope resolution operator `::` in Java.
- No forward references of classes are necessary in Java.
- No destructors in Java.
- Java has no templates.
- No nested classes in Java.
- Inheritance in Java has the same effect as in C++, but the syntax is different.
- Java does not provide direct support for multiple inheritance. We can accomplish multiple inheritance by using interfaces.
- Access specifiers (`public`, `private`, `protected` and `private protected`) are placed on each definition for each member of a class.

- A class in Java can have an access specifier to determine whether it is visible outside the file.
- There is no virtual keyword in Java. All non-static methods always use dynamic binding.
- Initialization of primitive class data member is guaranteed in Java. We can initialize them directly when we define them in the class, or we can do it in the constructor.
- We need not externally define storage for static members like we do in C++.

**C.7****STRINGS**

- Strings in C and C++ are arrays of characters, terminated by a null character. But strings in Java are objects. They are not terminated by a null. Therefore, strings are treated differently in C++ and Java.
- Strings can be concatenated using + operator.

**C.8****ARRAYS**

- Arrays are quite different in Java. Array boundaries are strictly enforced. Attempting to read past the end of an array produces an error.
- One array can be assigned to another in Java.
- Java does not support multidimensional arrays as in C and C++. However, it is possible to create arrays of arrays to represent multidimensional arrays.

**C.9****CONTROL FLOW**

- There is no goto in Java. We can, however, use labelled break and continue statements to jump out of the loops.
- The test expressions for control flow constructs return a boolean value (true or false) in Java. In C and C++, they return an integer value.
- The control variable declared in for loop is not available after the loop is exited in Java.

**C.10****COMMAND-LINE ARGUMENTS**

- The command line arguments passed from the system into a Java program differ in a couple of ways compared to that of C++ program.
- In C and C++, two arguments are passed. One specifies the number of arguments and the other is a pointer to an array of characters containing the actual arguments. In Java, a single argument containing an array of strings is passed.

- The first element in the arguments vector in C and C++ is, the name of the program itself. In Java, we do not pass the name of the program as an argument. We already know the name of the program because it is the same name as the class.



## OTHER DIFFERENCES

- Java supports multithreading.
- Java supports automatic garbage collection and makes a lot of programming problems simply vanish.
- The destructor function is replaced with a finalize function.
- Exception handling in Java is different because there are no destructors. A finally clause is always executed to perform necessary cleanup.
- Java has built-in support for comment documentation, so the source code file can also contain its own documentation.

---

## **Appendix D**



### **D.1**

#### **INTRODUCTION**

One of the unique features of Java language as compared to other high-level languages is that it allows direct manipulation of individual bits within a word. Bit-level manipulations are used in setting a particular bit or group of bits to 1 or 0. They are also used to perform certain numerical computations faster. As pointed out in Chapter 5, Java supports the following operators:

1. Bitwise logical operators
2. Bitwise shift operators
3. One's complement operator

All these operators work only on integer type operands.

### **D.2**

#### **BITWISE LOGICAL OPERATORS**

There are three logical bitwise operators. **They** are:

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise exclusive OR (^)

These are binary operators and require two integer-type operands. These operators work on their operands bit by bit starting from the least significant (i.e. the rightmost) bit, setting each bit in the result as shown in Table D.1.

**Table D.I Result of Logical Bitwise Operations**

opl	op2	opl & op2	opl   op2	opl ^ op2
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

## Bitwise AND

The bitwise AND operator is represented by a single ampersand (*Be*) and is surrounded on both sides by integer expressions. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Let us consider two variables x and y whose values are 13 and 25. The binary representation of these two variables are

```
x ---+ 0000 0000 0000 1101  
y ---+ 0000 0000 0001 1001
```

If we execute statement

```
z = x & y;
```

then the result would be:

```
z ....0000.0000 0000 1001
```

Although the resulting bit pattern represents the decimal number 9, there is no apparent connection between the decimal values of these three variables. Program D.1 shows how to use the bitwise operators.

## Program *D.1* Demonstration of bitwise operators

```
Class Bitwise  
{  
    public static void main (String args[])  
    {  
        int a=13, b=25;  
        System.out.println("a      = " + a);  
        System.out.println("b      = " + b);  
        System.out.println("a & b = " + (a & b));  
        System.out.println("a | b = " + (a | b));  
        System.out.println("a ^ b = " + (a ^ b));  
    }  
}
```

The output would be:

```
a = 13
b = 25
a & ,b = 9
al b = 29
a " b = 20
```

Bitwise ANDing is often used to test whether a particular bit is 1 or 0. For example, the following program tests whether the fourth bit of the variable flag is 1 or 0.

```
Class Bit1
{
    Static final TEST = 8;           /* represents 00....01000 */
    public static void main (String args[])
    {
        int flag;
        .....
        .....

        if ((flaq , TEST) != 0)          /* test 4th bit */
        {
            System.out.println(UFourth      bit is set \n");
        }
        .....
        .....

    }
}
```

Note that the bitwise logical operators have lower precedence than the relational operators and therefore additional parentheses are necessary as shown above.

The following program tests whether a given number is odd or even.

```
Class Bit2
{
    public static void main(String args[])
    {
        int test = 1;
        int number;

        // Input a number here
        .....
        .....

        while(number != -1)
        {
```

```

        if(number , test)
            System.out.println(Number      is odd\n\n");
        else
            System.out.println(Number      is even\n\n");
        // Input a number here
        .....
        .....
    }
}
}

```

---

### Output:

```

Input a number
20
Number is even
Input a number
9
Number is odd
Input a number
-1

```

### Bitwise OR

The bitwise OR is represented by the symbol | (vertical bar) and is surrounded by two integer operands. The result of OR operation is 1 if *at least* one of the bits has a value of 1; otherwise it is zero. Consider the variables x and y discussed above.

x	- - +	0000	0000	0000	1101
y	- - +	0000	0000	0001	1001
xly	- - +	0000	0000	0001	1101

The bitwise inclusion OR operation is often used to set a particular bit to 1 in a flag. Example:

---

```

Class Bit3
{
    final static SET = 8;
    public static void main (String args[])
    {
        int flag;
        .....
    }
}

```

```

.....
flag = flag | SET;
if ((flaq, SET) != 0)
{
    System.out.println("flaq      is _t\nIP);
}
.....
.....
}
}

```

---

The statement

`flaq = flaq | SET;`

causes the fourth bit of flag to set 1 if it is 0 and does not change it if it is already 1.

### Bitwise Exclusive OR

The bitwise *exclusive OR* is represented by the symbol `^`. The result of exclusive OR is 1 if *only one* of the bits is 1; otherwise it is 0. Consider again the same variables x and y discussed above.

x	→	0000	0000	0000	1101
y	→	0000	0000	0001	1001
x^y	→	0000	0000	0001	0100

D3

## BITWISE SHIFT OPERATORS

The shift operators are used to move bit patterns either to the left or to the right. The shift operators are represented by the symbols `<<` and `>>` and are used in the following form:

Let shift:	<code>op &lt;&lt; n</code>
Right shift:	<code>op &gt;&gt; n</code>

op is the integer expression that is to be shifted and n is the number of bit positions to be shifted.

The left-shift operation causes all the bits in the operand `ap` to be shifted to the left by `n` positions. The leftmost `n` bits in the original bit pattern will be lost and rightmost `n` bit positions that are vacated will be filled with Os.

Similarly, the right-shift operation causes all the bits in the operand `ap` to be shifted to the right by `n` positions. The rightmost `n` bits will be lost. The leftmost `n` bit positions that are vacated will be filled with zero, if the op is a *positive integer*. If the variable to be shifted is

*negative*, then the operation preserves the high-order bit of 1 and shifts only the lower 31 bits to the right.

Both the operands *op* and *n* can be constants or variables. There are two restrictions on the value of *n*. It may not be negative and it may not exceed the number of bits used to represent the left operand *op*.

Let us suppose *x* is a positive integer whose bit pattern is

0100 1001 1100 1011

then,

$x \ll 3$	= 0100 1110 0101 1000	↑	vacated positions
$x \gg 3$	= 0D00 1001 0011 1001		

Shift operators are often used for multiplication and division by powers of two.

Consider the following statement:

`x = y << 1;`

This statement shifts one bit to the left in *y* and then the result is assigned to *x*. The decimal value of *x* will be the value of *y* multiplied by 2. Similarly, the statement

`x = y >> 1;`

shifts *y* one bit to the right and assigns the result to *x*. In this case, the value of *x* will be the value of *y* divided by 2.

Java supports another shift operator `>>>` known as zero-fill-right-shift operator. When dealing with positive numbers, there is no difference between this operator and the right-shift operator. They both shift zeros into the upper bits of a number. The difference arises when dealing with negative numbers. Note that negative numbers have the high-order bit set to 1. The right-shift operator preserves the high-order bit as 1. The zero-fill-right-shift operator shifts zeros into all the upper bits, including the high-order bit, thus making a negative number into positive. Program D.2 demonstrates the use of shift operators.

## *Program D.2 Demonstration of Shift operators*

---

```
Class Shift
{
    public static void main (String args[])
    {
        int a=8, b=-8;
```

---

*(Continued)*

Program D.2 (Continued)

---

```
System.out.println("a      = " + a + "      b = " + b);
System.out.println("a      >> 2 = " + (a>>2);
System.out.println("a      << 1 = " + (a << 1);
System.out.println("a      >>> 1 = " + (a >>> 1);
System.out.println("b      >> 1 = " + (b >> 1);
System.out.println("b      >>> 1 = " + (b >>> 1);
}
}
```

---

The output would be:

```
a=8      b =-8
a>> 2 = 2
a<< 1 = 16
a>>> 1 = 4
b>>I=-4
b >>> 1 = 2147483644
```



## BITWISE COMPLEMENT OPERATORS

The complement operator - (also called the one's complement operator) is an unary operator and inverts all the bits represented by its operand. That is, Osbecome Is and Is become zero.  
Example:

```
x      = 1001 0110 1100 1011
-x     = 0110 1001 0011 0100
```

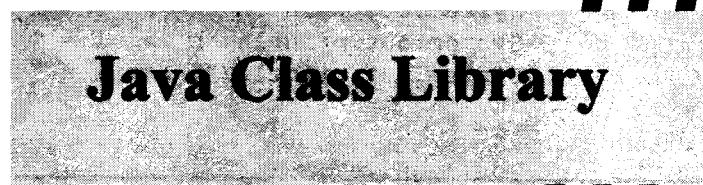
This operator is often combined with the bitwise ANDoperator to turn off a particular bit. For example, the statements

```
x      = 8;          /* 0000 0000 0000 1000 */
flag   = flag & -xi
```

would turn off the fourth bit in the variable **flq**.

---

## **Appendix E**



Java class libraries are implemented as packages, which contain groups of related classes. Along with classes, they also include interfaces, exception definitions, and error definitions. Java is composed of six standard packages grouped as follows:

### Class Libraries for Stand-alone Application Programming

1. java.lang
2. java.util
3. java.io

### Class Libraries for Applet and Network Programming

4. java.awt
5. java.applet
6. java.net

This appendix lists the interfaces and classes contained in various packages.

Table E.' iava.'" Package

---

#### Interfaces

---

Cloneable	Interface indicating that an object may be copied or cloned
Runnable	Methods for classes. that want to run as threads

---

#### Classes

---

Boolean	Object wrapper for boolean values
---------	-----------------------------------

Table E.1 (*Continued*)

## Classes

---

Character	Object wrapper for char values
Class	Run-time representations of classes
ClassLoader	Abstract behavior for handling loading of classes
Compiler	System class that gives access to the Java compiler
Double	Object wrapper for double values
Float	Object wrapper for float values
Integer	Object wrapper for int values
Long	Object wrapper for long values
Math	Utility class for math operations
Number	Abstract superclass of all number classes (Integer, Float, and so on)
Object	Generic object class, at top of inheritance hierarchy
Process	Abstract behavior for processes such as those spawned using methods in the System class
Runtime	Access to the Java runtime
SecurityManager	Abstract behavior for implementing security policies
String	Character strings
StringBuffer	Mutable strings
System	Access to Java's system-level behavior, provided in a platform independent way
Thread	Methods for managing threads and classes that run in threads
ThreadDeath	Class of object thrown when a thread is asynchronously terminated
ThreadGroup	A group of threads
Throwable	Generic exception class; all objects thrown must be a Throwable.

---

Table E.2 **jaYLutil Padraee**

## Interfaces

---

Enumeration	Methods for enumerating sets of values
Observer	Methods for enabling classes to be Observable objects

---

## Classes

---

BitSet	A set of bits
--------	---------------

---

*(Continued)*

**Classes**

---

Date	The current system date, as well as methods for generating and parsing dates
Dictionary	An abstract class that maps between keys and values (superclass of HashTable)
Hashtable	A hash table
Observable	An abstract class for observable objects
Properties	A hash table that contains behavior for setting and retrieving persistent properties of the system or a class
Random	Utilities for generating random numbers
Stack	A stack (a last-in-first-out queue)
StringTokenizer	Utilities for splitting strings into individual "token"
Vector	A growable array of Objects

---

**Table E.3 java.io Package**

---

**Interfaces**

---

DataInput	Methods for reading machine-independent typed input streams
DataOutput	Methods for writing machine-independent typed output streams
FilenameFilter	Methods for filtering file names

---

**Classes**

---

BufferedInputStream	A buffered input stream
BufferedOutputStream	A buffered output stream
ByteArrayInputStream	An input stream from a byte array
ByteArrayOutputStream	An output stream to a byte array
DataInputStream	Enables you to read primitive Java types (ints, chars, booleans, and so on) from a stream in a machine-independent way
DataOutputStream	Enables you to write primitive Java data types (ints, chars, booleans, and so on) to a stream in a machine-independent way
File	Represents a file on the host's file system

---

*(Continued)*

Table E.J (*Continued*)

## Classes

FileDescriptor	Holds onto the UNIX-like file descriptor of a file or socket
InputStream	An input stream from a file, constructed using a filename or descriptor
OutputStream	An output stream to a file, constructed using a filename or descriptor
FilterInputStream	Abstract class which provides a filter for input streams (and for adding stream functionality such as buffering)
FilterOutputStream	Abstract class which provides a filter for output streams (and for adding stream functionality such as buffering)
InputStream	An abstract class representing an input stream of bytes; the parent of all input streams in this package
LineNumberInputStream	An input stream that keeps track of line numbers
OutputStream	An abstract class representing an output stream of bytes; the parent of all output stream in this package
PipedInputStream	A piped input stream, which should be connected to a PipedOutputStream to be useful
PipedOutput Stream	A piped output stream, which should be connected to a PipedInputStream to be useful (together they provide safe communication between threads)
PrintStream	An output stream for printing (used by System.out.println( ...))
PushbackInputStream	An input stream with a 1-byte push back buffer
RandomAccessFile	Provides random access to a file, constructed from filenames, descriptors, or objects
SequenceInputStream	Converts a sequence of input streams into a single input stream
StreamTokenizer	Converts an input stream into a series of <b>individual</b> tokens
StringBufferInputStream	An input stream from a String object

**Table E.4 java.awt Package**

## Interfaces

LayoutManager	Methods for laying out containers
MenuCont.ainer	Methods for menu-related containers

(Continued)

---

**Table E.4 (Continued)**

---

**Classes**

---

BorderLayout	A layout manager for arranging items in border formation
Button	A UI pushbutton
Canvas	A canvas for drawing and performing other graphics operations
CardLayout	A layout manager for HyperCard-like metaphors
Checkbox	A checkbox
CheckboxGroup	A group of exclusive checkboxes (radio buttons)
CheckboxMenuItem	A toggle menu item
Choice	A popup menu of choices
Color	An abstract representation of a color
Component	The abstract generic class for all UI components
Container	Abstract behavior for a component that can hold other components or containers
Dialog	A window for brief interactions with users
Dimension	An object representing width and height
Event	An object representing events caused by the system or based on user input
FileDialog	A dialog for getting filenames from the local file system
FlowLayout	A layout manager that lays out objects from left to right in rows
Font	An abstract representation of a font
FontMetrics	Abstract class for holding information about a specific font's character shapes and height and width information
Frame	A top-level window with a title
Graphics	Abstract behavior for representing a graphics context, and for drawing and painting shapes and objects
GridBagConstraints	Constraints for components laid out using GridBagLayout
GridBagLayout	A layout manager that aligns components horizontally and vertically based on their values from GridBagConstraints
GridLayout	A layout manager with rows and columns; elements are added to each cell in the grid
Image	An abstract representation of a bitmap image
Insets	Distances from the outer border of the window; used to layout components

---

(Continued)

Table EA (*Continued*)

## Classes

Label	A text label for UI components
List	A scrolling list
MediaTracker	A way to keep track of the status of media objects being loaded over the Net
Menu	A menu, which can contain menu items and is a container on a menubar
MenuBar	A menubar (container for menus)
MenuComponent	The abstract superclass of all menu elements
MenuItem	An individual menu item
Panel	A container that is displayed
Point	An object representing a point (x and y coordinates)
Polygon	An object representing a set of points
Rectangle	An object representing a rectangle (x and y coordinates for the top corner, plus width and height)
Scrollbar	A UI scroll bar object
TextArea	A multiline, scrollable, editable text field
TextComponent	The superclass of all editable text components
TextField	A fixed-size editable text field
Toolkit	Abstract behavior for binding the abstract AWT classes to a platform-specific toolkit implementation
Window	A top-level window, and the superclass of the Frame and Dialog classes

**Table E.5 jaYa.awt.image Packaae**

## Interfaces

ImageConsumer	Methods for receiving image created by an ImageProducer
ImageObserver	Methods to track the loading and construction of an image
ImageProducer	Methods for producing image data received by an ImageConsumer

## Classes

ColorModel	An abstract class for <u>managing</u> color information for images
CropImageFilter	A filter for cropping images to a particular size

*(Continued)*

**Table E.5** (*Continued*)

## Classes

DirectColorModel	A specific color model for managing and translating pixel color values
FilteredImageSource	An ImageProducer that takes an image and an ImageFilter object, and produces an image for an ImageConsumer
ImageFilter	A filter that takes image data from an ImageProducer, modifies it in some way, and hands it off to an ImageConsumer
IndexColorModel	A specific color model for managing and translating color values in a fixed-color map
MemoryImageSource	An image producer that gets its image from memory; used after constructing an image by hand
PixelGrabber	An ImageConsumer that retrieves a subset of the pixels in an image
RGBImageFilter	Abstract behavior for a filter that modifies the RGB values of pixels in RGB images

**Table E.6** **java-applet** Package

## Interfaces

AppletContext  
AppletStub  
AudioClip

## Classes

Applet

**Table E.7** **java-net** Package

## Interfaces

ContentHandlerFactory  
SocketImplFactory  
URLStreamHandlerFactory

Methods for creating ContentHandler objects  
Methods for creating socket implementations (instance of the SocketImpl class)  
Methods for creating URLStreamHandler objects

(Continued)

Table E.7 (*Continued*)

---

Classes

---

ContentHandler	Abstract behavior for reading data from a URL connection and constructing the appropriate local object, based on MIME types
DatagramPacket	A datagram packet (UDP)
DatagramSocket	A datagram socket
InetAddress	An object representation of an Internet host (host name, IP address)
ServerSocket	A server-side socket
Socket	A socket
SocketImpl	An abstract class for specific socket implementations
URL	An object representation of a URL
URLConnection	Abstract behavior for a socket that can handle various Web-based protocols (http, ftp, and so on)
URLEncoder	Turns strings into x-www-form-urlencoded format
URLStreamHandler	Abstract class for managing streams to objects referenced by URLs

---

## **Appendix F**

### **Java Classes and Their Packages**

---

This appendix lists all the Java library classes in alphabetical order and indicates in which package a given class is defined. It also lists the classes that extend them.

Class	Package	Subclasses
AbstractMethodError	java.lang	Nil
AppletContext	java.applet	Nil
AppletStub	java.applet	Nil
Applet	java.applet	Nil
ArithmaticException	java.lang	Nil
ArrayIndexOutOfBoundsException	java.lang	Nil
ArrayStoreException	java.lang	Nil
AudioClip	java.applet	Nil
AWTError	java.awt	Nil
AWTException	java.awt	Nil
BitSet	java.util	Nil
Boolean	java.lang	Nil
BorderLayout	java.awt	Nil
BufferedInputStream	java.io	Nil
BufferedOutputStream	java.io	Nil
ButtonPeer	java.awt.peer	Nil
Button	java.awt	Nil
ByteArrayInputStream	java.io	Nil

Class	Package	Subclasses
ByteArrayOutputStream	java.io	Nil
CanvasPeer	java.awt.peer	Nil
Canvas	java.awt	Nil
CardLayout	java.awt	Nil
Character	java.lang	Nil
CheckboxGroup	java.awt	Nil
CheckboxMenuItemPeer	java.awt.peer	Nil
CheckboxMenuItem	java.awt	Nil
CheckboxPeer	java.awt.peer	Nil
Checkbox	java.awt	Nil
ChoicePeer	Java.awt.peer	Nil
Choice	java.awt	Nil
ClassCastException	java.lang	Nil
ClassCircularityError	java.lang	Nil
ClassFormatError	java.lang	Nil
ClassLoader	java.lang	Nil
ClassNotFoundException	java.lang	Nil
Class	java.lang	Nil
Cloneable	java.lang	Nil
CloneNotSupportedException	java.lang	Nil
ColorModel	java.awt.image	DirectColorModel, IndexColorModel
Color	java.awt	Nil
Compiler	java.lang	Nil
ComponentPeer	java.awt.peer	ButtonPeer, CanvasPeer, CheckboxPeer, ChoicePeer, ContainerPeer, LabelPeer, ListPeer, ScrollbarPeer, TextComponentPeer
Component	java.awt	Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent
ContainerPeer	java.awt.peer	PanelPeer, WindowPeer

Class	Package	Subclasses
Container	java.awt	Panel, Window
ContentHandlerFactol)'	java.net	Nil
ContentHandIer	java.net	Nil
CropImageFilter	java.awt.ima.ge	Nil
DatagramPacket	java.net	Nil
DatagramSocket	java.net	Nil
DataInputStream	java.io	Nil
DataInput	java.io	Nil
DataOutputStream	java.io	Nil
DataOutput	java.io	Nil
Date	java.util	Nil
DialogPeer	java.awt.peer	FileDialogPeer
Dialog	java.awt	FileDialog
Dictional)'	java.util	Hashtable
Dimension	java.awt	Nil
DirectColorModel	java.awt.image	Nil
Double	java.lang	Nil
EmptyStackException	java.util	Nil
Enumeration	java.util	Nil
EOFException	java.io	Nil
Error	j ava .lang	AWTError, LinkageError, ThreadDeath, VirtuaiMachineError
Event	java.awt	Nil
Exception	java.lang	AWTException, ClassNotFoundException, CloneNotSupportedException, IllegalAccessException, InstantiationException, InterruptedException, IOException, NoSuchMethodException, RuntimeException
FileDescriptor	java.io	Nil
FileDialogPeer	java.awt.peer	Nil

Class	Package	Subclasses
FileDialog	java.awt	Nil
FileInputStream	java.io	Nil
FilenameFilter	java.io	Nil
FileNotFoundException	java.io	Nil
FileOutputStream	java.io	Nil
File	java.io	Nil
FilteredImageSource	java.awt.image	Nil
FilterInputStream	java.io	BufferedInputStream, DataInputStream, LineNumberInputStream, PushbackInputStream BufferedOutputStream, DataOutputStream, PrintStream
Float	java.io	Nil
FlowLayout	java.awt	Nil
FontMetrics	java.awt	Nil
Font	java.awt	Nil
FramePeer	java.awt.peer	Nil
Frame	java.awt	Nil
Graphics	java.awt	Nil
GridBagConstraints	java.awt	Nil
GridBagLayout	java.awt	Nil
GridLayout	java.awt	Nil
Hashtable	java.util	Properties
IllegalAccessException	java.lang	Nil
IllegalAccessException	java.lang	Nil
IllegalArgumentException	java.lang	IllegalThreadStateException, NumberFormatException
IllegalMonitorStateException	java.lang	Nil
IllegalThreadStateException	java.lang	Nil
ImageConsumer	java.awt.image	Nil
ImageFilter	j *a. awt. image	Crop Image Filter , RGBImageFilter
ImageObserver	java.awt.image	Nil

Class	Package	Subclasses
ImageProducer	java.awt.image	Nil
Image	java.awt	Nil
IncompatibleClassChangeError	java.lang	AbstractMethodError, IllegalAccessError, InstantiationException, NoSuchFieldError, NoSuchMethodError
IndexColorModel	java.awt.image	Nil
IndexOutOfBoundsException	java.lang	ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException
InetAddress	java.net	Nil
InputStream	java.io	ByteArrayInputStream, FileInputStream, FilterInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream
Insets	java.awt	Nil
InstantiationException	java.lang	Nil
InstantiationException	java.lang	Nil
Integer	java.lang	Nil
InternalError	java.lang	Nil
InterruptedException	java.lang	Nil
InterruptedIOException	java.io	Nil
IOException	java.io	EOFException, FileNotFoundException, InterruptedIOException, MalformedURLException, ProtocolException, SocketException, UnknownHostException, UnknownServiceException, UTFDataFormatException
LabelPeer	java.awt.peer	Nil
Label	java.awt	Nil

Class	Package	Subclasses
LayoutManager	java.awt	Nil
LineNumberInputStream	java.io	Nil
LinkageError	java.lang	ClassCircularityError, ClassFormatError, IncompatibleClassChange Error, UnsatisfiedLinkError, VerifyError
ListPeer	java.awt.peer	Nil
List	java.awt	Nil
Long	java.lang	Nil
MalformedURLException	java.net	Nil
Math	java.lang	Nil
MediaTracker	java.awt	Nil
MemoryImageSource	java.awt.image	Nil
MenuBarPeer	java.awt.peer	Nil
Menubar	java.awt	Nil
MenuComponentPeer	java.awt.peer	CheckboxMenuItemPeer, MenuPeer
MenuComponent	java.awt	MenuBar, MenuItem
MenuContainer	java.awt	Nil
MenuItemPeer	java.awt.peer	CheckboxMenuItemPeer, MenuPeer
MenuItem	java.awt	CheckboxMenuItem, Menu
MenuPeer	java.awt.peer	Nil
Menu	java.awt	Nil
NegativeArraySizeException	java.lang	Nil
NoClassDefFoundError	java.lang	Nil
NoSuchElementException	java.util	Nil
NoSuchFieldError	java.lang	Nil
NoSuchMethodError	java.lang	Nil
NoSuchMethodException	java.lang	Nil
NullPointerException	java.lang	Nil
NumberFormatException	java.lang	Nil

Class	Package	Subclasses
Number	java.lang	Double, Float, Integer, Long
Object	java.lang	BitSet, Boolean, BorderLayout, CardLayout, Character, CheckboxGroup, Class, ClassLoader, Color; ColorModel, Compiler, Component, ContentHandler, DatagramPacket, DatagramSocket, Date, Dictionary, Dimension, Event, File, FileDescriptor, FilteredImageSource, FlowLayout, Font, FontMetrics, Graphics, GridBagConstraints, GridBagLayout, GridLayout, Image, ImageFilter, InetAddress, InputStream, Insets, Math, MediaTracker, MemoryImageSource, MenuComponent, Number, Observable, OutputStream, PeIGrabber, Point, Polygon, Process, Random, RandomAccessFile, Rectangle, Runtime, SecurityManager, ServerSocket, Socket, SocketImpl, StreamTokenizer, String, StringBuffer, StringTokenizer, System, Thread, ThreadGroup, Throwable, Toolkit, URL, URLConnection, URLEncoder, URLStreamHandler, Vector
Observable	java.util	Nil
Observer	java.util	Nil
OutOfMemoryError	java.lang	Nil
OutputStream	java.io	ByteArrayOutputStream, FileOutputStream, FilterOutputStream, PipedOutputStream
PanelPeer	. java. awt .peer	Nil
Panel	java.awt	Applet

Class	Package	Subclasses
PipedInputStream	java.io	Nil
PipedC>utputStrear.n	java.io	Nil
PixelGrabber	java.awt.image	Nil
Point	java.awt	Nil
Polygon	java.awt	Nil
PrintStream	java.io	Nil
Process	java.lang	Nil
Properties	java.util	Nil
ProtocolException	java.net	Nil
PushbackInputStream	java.io	Nil
RandomAccessFile	java.io	Nil
Randor.n	java.util	Nil
Rectangle	java.awt	Nil
RGBIr.nageFilter	java.awt.image	Nil
Runnable	java.lang	Nil
Runtir.neException	java.lang	Aritlir.neticException, ArrayStoreException, ClassCastException, Er.nptyStackException, IllegalArgur.nentException, IllegalMonitorStateException, IndexC>utC>fBoundsException, NegativeArraySizeException, NoSuchEler.nentException, NullPointerException, SecurityException
Runtir.ne	java.lang	Nil
ScrollbarPeer	java.awt.peer	Nil
Scrollbar	java.awt	Nil
SecurityException	java.lang	Nil
SecurityManager	java.lang	Nil
SequenceInputStream	java.io	Nil
SelVerSocket	java.net	Nil
SocketException	java.net	Nil

Class	Package	Subclasses
SocketImplFactory	java.net	Nil
SocketImpl	java.net	Nil
Socket	java.net	Nil
StackOverflowError	java.lang	Nil
Stack	j ava. util	Nil
StreamTokenizer	java.io	Nil
StringBufferInputStream	java.io	Nil
StringBuffer	java.lang	Nil
StringIndexOutOfBoundsException	j ava .lang	Nil
StringTokenizer	java.util	Nil
String	java.lang	Nil
System	java.lang	Nil
TextAreaPeer	java.awt.peer	Nil
TextArea	java.awt	Nil
TextComponentPeer	java.awt.peer	TextAreaPeer, TextFieldPeer
TextComponent	java.awt	TextArea, TextField
TextFieldPeer	java.awt.peer	Nil
TextField	java.awt	Nil
ThreadDeath	java.lang	Nil
ThreadGroup	java.lang	Nil
Thread	java.lang	Nil
Throwable	java.lang	Error, Exception
Toolkit	java.awt	Nil
UnknownError	java.lang	Nil
UnknownHostException	java.net	Nil
UnknownServiceException	java.net	Nil
UnsatisfiedLinkError	java.lang	Nil
URLConnection	java.net	Nil
URLEncoder	java.net	Nil
URLStreamHandlerFactory	java.net	Nil
URLStreamHandler	java.net	Nil
URL	java.net	Nil

Class	Package	Subclasses
UTFDataFormatException	java.io	Nil
Vector	java.util	Stack
VerifyError	java.lang	Nil
VirtualMachineError	java.lang	InternalError, OutOfMemoryError, StackOverflowError, UnknownError
WindowPeer	java.awt.peer	DialogPeer, FramePeer
Window	java.awt	Dialog, Frame

---

## **Appendix G**

### **Points to Remember**

**G.I**

#### **GENERAL**

1. It is important that the name of the file match the name of the class and the extension be .java.
2. All functions in Java must be members of some class.
3. Member functions are called methods in Java.
4. Creating two methods with the same name but different arguments is called method overloading.
5. Method overloading allows sets of methods with very similar purpose to be given the same name.
6. When a method makes an unqualified reference to another member of the same class, there is an implicit reference to this object.
7. Java does not provide a default constructor if the class defines a constructor of its own.
8. When present, package must be the first noncomment statement in the file.
9. The import statement must follow the package statement but must also precede all other noncomment statements.
10. The full method or class name, including the package name, must be used when two imported packages contain a method or class with the same name.
11. Due to security reasons, it is not possible to perform file I/O operations from an applet.
12. When a simple type is passed to a method, it is done by use of call-by-value. Objects are passed by use of call-by-reference.
13. It is illegal to refer to any instance variables inside of a static method.
14. All command-line arguments are passed as strings. We must therefore convert numeric values to their original forms manually.

15. A class member declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.
16. The star form of import statement may increase compile time. It will be good practice to explicitly name the classes that we want to use rather than importing whole packages.
17. Interfaces add most of the functionality & is required for many applications which would normally require the use of multiple inheritance in C++.
18. When we implement an interface method, it must be declared as public.
19. If a finally block is associated with a try, the final block will be executed upon conclusion of the try.
20. Java uses pointers (addresses) internally to store references to objects, and for elements of any array of objects. However, these pointers are not available for use by programmers.
21. We cannot overload methods with differences only in their return type.
22. When a method with the same signature occurs in both the super class and its subclass, the method in the subclass overrides the method in the super class.
23. Every constructor must invoke its super class constructor in its first statement. Otherwise, the default constructor of the super class will be called.
24. A class marked as final cannot be inherited.
25. A method marked final cannot be overridden.
26. Subclasses of an abstract class that do not provide an implementation of an abstract method; are also abstract.

**G.2*****C/C++ RELATED***

27. A Java string is not implemented as a null-terminated array of characters as it is in C and C++.
28. Most of the Java operators work much the same way as their C/C++ equivalents except for the addition of two new operators, >>> and " .
29. The comparison operators in Java return a Boolean true or false but not the integer one or zero.
30. The modulo division may be applied to floating point values in Java. This is not permitted in C/C++.
31. The control variable declared in for loop is visible only within the scope of the loop. But in C/C++, it is visible even after the loop is exited.
32. Methods cannot be declared with an explicitly void argument list, as done in C++.
33. Java methods must be defined within the class. Separate definition is not supported.

34. Unlike C/C++, Java checks the range of every subscript and generates an error message when it is violated.
35. Java does not support the destructor function. Instead, it uses the finalize method to restore the memory.
36. Java does not support multiple inheritance.
37. C++ has no equivalent to the finally block of Java.
38. Java is more strictly typed than C/C++ languages. For example, in Java, we cannot assign a floating point value to an integer (without explicit type casting).
39. Unlike C/C++ which allow the size of an integer to vary based on the execution environment, Java data types have a strictly defined range and does not change with the environment.
40. Java does not support pointers.
41. Java supports labelled break and labelled continue statements.
42. break has been designed for use only when some sort of special situation occurs. It should not be used to provide the normal means by which a loop is terminated.
43. The use of final to a variable is similar to the use of const in C/C++.
44. Overridden methods in Java are similar to virtual functions in C++.
45. Java does not have a generalized console input method that parallels the scanf in C or cin in C++.
46. Integer types are always signed in Java. There are no unsigned qualifiers.
47. Java does not define escape codes for vertical tab and the bell character.
48. In Java multidimensional arrays are created as arrays of arrays. We can define variable size arrays of arrays.

---

## **Appendix H**

### **Common Coding Errors**

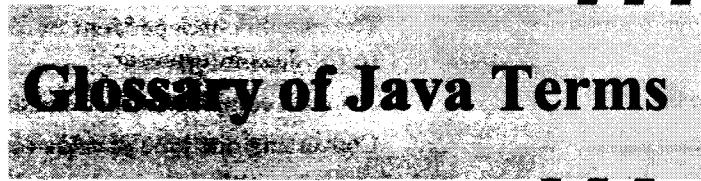
This appendix lists some common coding errors that Java programmers are likely to make.

1. File name is not identical to the class name (in both spelling and case).
2. File not ending with .Java extension for a file containing the main method class (or applet's class definition).
3. Not ending a Java statement with a semicolon.
4. Providing spaces between the symbols of the operators ==, <=, >= and !=.
5. Using operators <=, >=, and != as =<, => and !=.
6. Using the operators = in the place of ==.
7. Using the keywords as identifiers.
8. Not properly matching the braces.
- 9.. Placing a semicolon after the condition in an if statement or a while statement.
10. Using uppercase letters while writing a keyword.
11. Not initializing the variables properly.
12. Using floating point values in relational expressions.
13. Applying increment or decrement operator on an expression other than a simple variable.
14. Using commas instead of semicolons in a for header.
15. Placing a semicolon immediately after a for header.
16. Defining a method outside the braces of a class definition.
17. Not providing the return type in a method definition.
18. Not matching the type of the value returned by a method with its return type declared.
19. Placing a semicolon immediately after a method header.

20. Declaring a method parameter as a local variable inside the method.
21. Defining a method inside another method.
22. Attempting to assign a value to a final variable. -
23. Using the same variable name in both the outer and inner blocks of a program.
24. Changing only return types for method overloading.
25. Referring to an element outside the array bounds.
26. Declaring a return type for a constructor method.
27. A class trying to access a private variable of another class.
28. Using this reference explicitly in a static method.
29. Trying to access an instance variable or an instance method inside a static method.
30. Assigning an object of a superclass to a subclass reference (without a cast).
31. Not using super method call as a first statement in the subclass constructor.
32. Attempting to instantiate an object of an abstract class.
33. Not declaring explicitly a class' as abstract when it contains one or more abstract methods.
34. Using the instance variable length instead of method length() to determine the length of a string.
35. Using the method length( ) instead of the instance variable length to determine the size of an array.
36. Using string objects to access StringBuffer methods that are not members of the class String.
37. Attempting to access a character that is outside the bounds of a string.
38. Using a lower case fin TextField.
39. Trying to catch the same type of exception in two different catch blocks associated with a particular try block.
40. Placing catch (Exception e) before other catch blocks. This should be the last in the list of catch blocks.
41. Placing a catch that catches a superclass object *before* a catch that catches an object of its subclass.

---

## **Appendix I**



abstract class	A class that cannot be instantiated directly. Abstract classes exist so that subclasses can inherit variables and methods from them.
access control	A way to restrict access to classes, variables and methods.
API	Application Programming Interface. The Java API contains classes a programmer can use to build applications and applets.
applet	A Java program that is embedded in an HTML document and runs in the context of a Java-capable browser.
appletviewer	A tool created by SUN to run applets without a browser.
argument	A value that is sent to a method when the method is called.
array	A list of values of the same type. All values in an array have a common name.
ASCII	A standard set of values for representing text characters.
assignment expression	Assigns a value to a variable.
attribute	A specifier for an HTML tag (for example, code is an attribute of the <APPLET> tag)
AWT	The Abstract Windowing Toolkit, or group of classes for writing programs with graphical user interfaces.
baseclass	A class from which another class inherits functionality. In Java, a baseclass is often called a superclass.
bit	The smallest piece of data a computer understands. A bit can represent only two values, 0 or 1.
bitmap	A graphical image that is usually stored in a file.
Boolean	A value that can be either true or false.
Boolean expression	A Boolean expression evaluates to either true or false.

branching	When an execution jumps forward or backward in the program.
browser	A program used for reading, displaying, and interacting with objects on the World Wide Web.
byte	In Java, the byte is a data type, which is eight bits long.
bytecode	The machine-independent output of the Java compiler and input to the Java interpreter.
canvas	A applet component that can display graphics and text.
casting	Converting one type of value to another.
character	A value used in text. For example, the letters A-the digits 0-9 (when not used as mathematical values), spaces, and even tabs and carriage returns are all characters.
class	A collection of variables and methods that an object can have, or a template for building objects.
.class file	A file containing machine-independent Java bytecodes. The Java compiler generates .class files for the interpreter to read.
class variable	A variable allocated once per class. Class variables have global class scope and belong to the entire class instead of an instance.
client	A program that relies on services provided by another program called a server.
code	An attribute of the HTML<APPLET> tag that specifies the class to be loaded.
codebase	An attribute of the HTML<APPLET> tag that specifies the location of the classes to load.
comparison operators	Operators like == (equals) and > (greater than) that compare two expressions, giving a result of true or false.
compiler	A language translator. A program that transforms source code into another format without executing the program.
concatenate	Adding one text string to the end of another.
conditional branching	When a program jumps to a different part of a program based on a certain condition being met.
configurable applet	An applet that the user can customise by supplying different parameters when writing the applet's tag in an HTML document.
constant	A value that never changes throughout the life of a program.
constructor	A method that is used to create an instantiation of a class.
control variable	The variable that a program evaluates to determine whether or not to perform an action. Control variables are used in loops, switch statements, and other similar programming constructs.

data field	The data that is encapsulated in an object.
data type	The type of value represented by a constant, variable, or some other program object. Java data types include the integer types byte, short, int, and long; the floating-point types float and double; the character type char; and the Boolean type boolean.
deadlock	Deadlock occurs when two or more threads are waiting for resources that they can't get.
derived class	A class that inherits from a base class.
dialog box	A special pop-up window that can present important information to the user or that requests information from the user. A dialog box is an object of Java's Dialog class.
doctags	Special symbols used by the javadoc tool to document Java packages and methods.
double	In Java, the double is a data type, which is 64 bits in length.
dynamic linking	When functions called within a program are associated with the program at runtime rather than at compile time.
encapsulation	A way to contain data and methods in a class so that methods and variables may be added, changed, or deleted without requiring the code that uses the class to change.
exception	A signal that something has happened to stop normal execution of a program, usually an error.
exception handler	Code that responds to and attempts to recover from an exception.
expression	A line of program code that can be reduced to a value or that assigns a value.
extends	A keyword used to make one class a subclass of another, for example, class subclass extends superclass.
field	A data object encapsulated in a class.
final	A modifier that prevents subclass definition, makes variables constant, and prevents a subclass from overriding a method.
finalize	A method that is called when there are no further references to an object and it is no longer needed. This method releases resources and does any other necessary cleanup that Java does not handle during garbage collection.
float	In Java, the float is a data type, which is thirty-two bits long.
floating point	A value with both whole number (including zero) and fractional parts.
font	A set of characters of the same style.

frame window	A special pop-up window that can be displayed from an applet. A frame window is an object of Java's Frame class.
GIF	One type of data format for storing graphical images on disk.
GUI	Stands for Graphical User Interface. It is pronounced like "gooey"
high-level language	A computer language that isolates the programmer from the intricate details of programming a computer. Java is a high-level language.
HotJava	A Java-capable browser from Javasoft.
hspace	An attribute of the HTML<APPLET> tag that specifies the amount of horizontal space(to the left and right) between the applet and the text on the page.
HTML	Hypertext Markup Language, the language used to create Web pages.
identifier	A symbol that represents a program object.
index	The same as a subscript. Used to identify a specific array element.
infinite loop	A loop that cannot stop executing because its conditional expression can never be true.
inheritance	A property of object-oriented languages where a class inherits the methods and variables of more general classes.
initialise	Set the starting state of a program object. For example, you should initialise variables to a value before you attempt to use them in comparisons and other operations.
instance	A concrete representation of a class or object. A class can have many instances.
instance variable	A variable allocated once per instance of a class.
instantiate	To create a concrete object from a class "template". New objects are instantiated with new.
int	In Java, the int is a data type, which is 32 bits long.
integer	A whole-number value.
interface	A collection of methods and variables that other classes may implement. A class that implements an interface provides implementations for all the methods in the interface.
Internet	A huge world-spanning network of computers that can be used by anyone with a suitable connection.
Interpreter	A program that performs both language translation and program execution. java is the Java interpreter.

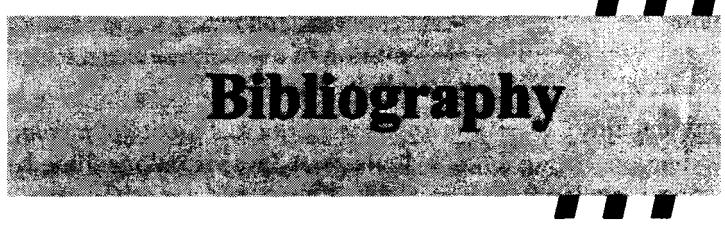
java	The program used to invoke the Java interpreter, which executes Java programs.
java file	A file containing Java source code.
javac	A command for running the Java compiler.
javac_g	A command for running a non-optimized version of the Java compiler. The javacj command can be used with debuggers, such as jdb.
Java-capable browser	A Web browser that can run Java applets. Also called a Java-enabled or Java-enhanced browser.
javadoc	A command that is used to generate API-style HTML documentation automatically.
javah	A command that can create C include files and stubs from a Java .class file. The resulting C files allow C code to access parameters passed from Java, return values to Java, and access Java class variables.
javahj	A command that can create C include files and stubs with debug information from a Java .class file.
jaap	A command that disassembles Java .class files.
JavaScript	A Java-based scripting language.
jdb	The Java debugger.
JDBC	A database access API from JavaSoft that allows developers to access databases with Java programs.
JDK	The Java Developers Kit.
literals	Values, such as a number or text string, that are written literally as part of program code. The opposite of a literal is a variable.
local applet	An applet that is stored on the local computer system, rather than somewhere else on the Internet.
logical expression	An expression that results in a value of true or false. (see Boolean eXpression)
logical operators	Operators like && (AND) and    (OR) that enable you to create logical expressions that yield true or false results.
long	In Java, the long is a data type which is 64 bits in length and can hold truly immense numbers.
loop	A program construct that enables a program to perform repetitive tasks.
method	A routine that belongs to a class.

**m** *Plot1ammilla wHhJava: A Primer*

modifier	A Java keyword that is applied to a method or variable declaration to control access, control execution, or provide additional information.
modular programming	Breaking a large program down into a number of functions, each of which performs a specific, well-defined task.
multidimensional array	An array that must be accessed using more than one subscript.
multiple inheritance	When a class simultaneously inherits methods and fields directly from more than one base class.
multitasking	Running more than one computer program concurrently (see multithread).
multithreaded	Having multiple threads of execution so that parts of a program can execute concurrently.
native methods	Methods that are declared in Java with the keyword native but are implemented in another language. Usually, native methods are written to do something that the Java API does not already do, to interact with a particular computer's hardware or operating system or to improve performance. Since native methods are not portable across platforms, applets cannot contain native methods.
nesting	When one program block is placed within another program block.
numerical expression	A combination of numbers, variables, or constants with operators.
Oak	The original name of the Java programming language.
object	An instantiation of a class.
object-oriented programming	A programming paradigm that treats program elements as objects that have data fields and functions that act on the data fields. The three main characteristics of OOP are encapsulation, inheritance, and the polymorphism.
one-dimensional array	An array that's set up like a list and requires only one subscript to identify a value contained in the array.
operator precedence	Determines the order in which mathematical operations are performed.
override	To replace a method inherited from a superclass.
package	A Java keyword used to assign the contents of a file to a package. Packages are Java's mechanism for grouping classes. Packages simplify reuse, and they are very useful for large projects.
parameter	A value that is passed to an applet or method. In the case of an applet, the parameters are defined in the HTML document, using the <PARAM> tag.

streamS	Controlled flows of data from one source to another. Java supplies several classes to create and manage streams. Classes that handle input data are derived from class <code>InputStream</code> , and classes that handle output data are derived from class <code>OutputStream</code> .
structured programming	A style of programming in which the program code is divided into logically structured chunks of code.
stub	Part of the interface between Java code and a native method. A stub allows a native method to access Java parameters, access Java class variables, and return data to Java.
subclass	A class that inherits methods and variables from another class. The statement <code>class SubClass extends SuperClass</code> means that <code>SubClass</code> is a subclass of <code>SuperClass</code> .
subscript	A subscript is a number that identifies the element of an array in which a value is stored. A subscript is sometimes called an index.
superclass	A generalization of another class. <code>X</code> is a superclass of <code>Y</code> if <code>Y</code> inherits variables and methods from <code>X</code> .
symbolic constant	A word that represents a value in a program.
synchronized	A Java keyword that prevents more than one thread from executing inside a method at once.
tag	A command in an HTML document that identifies the type of document component.
thread	A single path of execution that is a subprocess of the main process. All applications have at least one thread, which represents the main program. An application can create additional threads to handle multiple tasks concurrently.
top-down programming	A style of programming that divides tasks up into general modules. At the top level of the program are only the general tasks, whereas, as we work our way deeper into the program code, the programming becomes more and more detailed.
two-dimensional array	An array that is set up much like a table, with a specific number of columns and rows.
type cast	Convert one type of value to another.
unconditional branching	When program execution jumps to a new part of the program regardless of any conditions.
unicode	A new set of standard values for representing the symbols used in text. The unicode character set is much larger than the ASCII character set and allows for foreign-language symbols.
unsigned	A value that can only be positive. This is the opposite of signed. Unsigned numbers are not used in Java programming.

URL	Stands for Uniform Resource Locator, which is an Internet address.
variable	A value that can change as often as necessary during the execution of a program. Variables are always represented by symbolic names.
virtual machine	An abstract, logical model of a computer used to execute Java bytecodes. A Java-virtual machine has an instruction set, registers, a stack, a heap, and a method area.
VRML	Virtual Reality Modelling Language.
vspace	An attribute of the HTML<APPLET>tag that specifies the amount of vertical space above and below the applet and the text on the page.
Web browser	An application used to access the Internet's World Wide Web.
World Wide Web	The graphical part of the Internet.



## Bibliography

1. Aaron E.Walsh, *Foundations of Java Programming for the World Wide Web*, IDG Books Worldwide, 1996.
2. Anuff Ed, *The Java Sourcebook*, John Wiley & Sons, 1996.
3. Au, Edith and Dave Makower, *Java Programming Basics*, MIS Press, 1996.
4. Balagurusamy, E., *Object-Oriented Programming with C++*, TataMcGraw-Hill, 1995.
5. Balagurusamy, E., *Programming in ANSI C*, Tata McGraw-Hill, 1992.
6. Bartlett, Leslie and Simkin, *Java Programming Explorer*, Coriolis Group Books, 1996.
7. Boone, Barry, *Java Essentials for C and C++ Programmers*, Addison Wesley Developers Press, 1996.
- B. Daconta, Michael C., *Java for C/C++ Programmers*, John Wiley & Sons, 1996.
9. Davis, Stephen R., *Learn Java NOW*, Microsoft Press, 1996.
10. December, John, *Java*, Sarns.net, 1995.
11. Flanagan, D, *Java in a Nutshell*, O'Reilly & Associates, 1996.
12. Holzner, Steven, *Java Workshop Programming*, M & T Books, 1996.
13. Lemay, Laura and Charles L.Perkins, *Teach Yourself Java in 21 Days*, Sarns.net, 1996.
14. Naughton, Patrick and Herbert Schildt, *Java: The Complete Reference*, Osborne McGraw-Hill, 1996.
15. Naughton, Patrick, *The Java Handbook*, Osborne McGraw-Hill, 1996.
16. Newman, Alexander, et al., *Using Java*, Que Corporation, 1996.
17. Norton, Peter and William Stanek, *Guide to Java Programming*, Sarns.net, 1996.
18. Perry, Paul J., *Cool WebApplets with Java*, IDG Books, 1996.
19. Sarns.net, *Java Unleashed*, 1996.
20. Siyam, Karanjit S., *Inside VisualJ++*, New Riders, 1996.
21. Stout, Rick, *The World Wide Web, Complete Reference*, Osborne McGraw-Hill, 1996.

22. Tyma, Paul M., Gabriel Torok and Troy Downing, *Java Primer Plus*, Waite Group Press, 1996.
23. Walnum, Clayton, *Java by Example*, Que Corporation, 1996.

# Index

A

abstract classes 149-150  
abstract methods 149  
access modifiers 150  
    friendly 150  
    private 151  
    protected 151  
    public 150  
accessing class members 134  
accessing interface variables 184  
accessing packages 193  
adding a class to a package 198  
applet 17  
    building code for 240  
executable 245  
life-cycle of 243  
local 237  
remote 237  
parameters for 277  
    preparing for 239  
applet tag 22,248  
appletviewer 22, 270  
arithmetic 75  
    expression 66  
    floating point 68  
    integer 67  
    mixed-mode 68

operators 66

real 67

arrays 155

    creation 157

    declaration 158

    initialisation 158

    length of 159

    one-dimensional 155

    string 166

    two-dimensional 161

    variable size 164

assignment operators 72

assignment statement 54

associatively operators 81

automatic type conversion 59, 78

B

backslash character constants 49  
base class 142  
bitwise operators 74,297  
blocking a thread 208  
boolean type data 53  
bottom-up programming 1  
break statement 123  
branching 88  
bytecode .12,22,42

C 2,12,15-17  
 C++ 2,12,15-17  
 casting 58,79  
 character constant 49  
 character type data 53  
 child class 142  
 class variable 56,139  
 classes 3,129  
     abstract 149  
     graphics 265  
     math 82  
     string 165  
     stringBuffer 165,167  
     vector 169  
     wrapper 171  
 client 26  
 codebase attribute 237  
 command line arguments 43  
 compiler 12,26,40  
 compiling a program 40  
 conditional branching 88  
 conditional operators 74,106  
 constants 47  
     backslash characters 49  
     character 49  
     integer 47  
     real 48  
     string 49  
     symbolic 57  
 constructors 137  
 continue statement 123  
 control statements 88  
     break 123  
     continue 123  
     do 114  
     for 116  
     if 88,90  
     switch 102  
     while 113  
 control visibility 150



conventions of naming packages 191  
 creating a program 40  
 creating an array 157  
 creating objects 133  
 creating packages 192  
 creating threads 203



data abstraction 4  
 data hiding 4  
 data types 50  
     boolean type 53  
     character types 53  
     floating point 52  
     integer types 51  
 deadlocks 219  
 declaration of variable 53  
 decrement operators 73  
 default values 63  
 defining interfaces 179  
 derived class 142  
 do statement 114  
 documentation comment 32  
 dot operators 75,139  
 dynamic binding 6,15



else if ladder 98  
 empty statement 38,120  
 encapsulation 4,129  
 entry controlled loop 111-112  
 errors 224  
     compile time 224  
     run time 225  
 exceptions 226  
 executable applet 245  
 exit controlled loop 111-112  
 expressions 75  
     evaluation of 76

mixed-mode 68  
type conversion in 78  
extending a thread 204  
extending interfaces 181



fields 129  
final classes 149  
final method 148  
final variables 148  
finalizer methods 149  
floating point arithmetic 68  
floating point data 52  
for statement 116  
friendly access 150  
  
giving values to variables 54  
graphics 263  
arcs 269  
bar charts 277  
circles 267  
class 263-264  
ellipse 267  
lines 265-266, 274  
polygons 271  
rectangles 265-266  
graphics class 263  
green project 12  
guarding statement 38



hiding classes 199  
hierarchical inheritance 146  
HotJava 13, 20  
HTMLdocuments 237,246

HTMLfile 248-249  
HTMLtag 277  
Hypertext Mark-up Language 18, 26



identifiers 36  
if statement 88, 90  
iLelse statement 92  
implementing interfaces 182  
import statement 33  
increment operators 73  
index number 155  
infinite loop 111  
inheritance 5,129,130,142,179  
    hierarchical 146  
    multilevel 145  
    multiple 179  
inheritance path 145  
initialisation 55  
initialisation of arrays 158  
instance methods 139  
instanceof operator 75  
instance variable 56, 139  
integer constants 47  
    decimal 47  
    hexadecimal 48  
    octal 48  
integer type data 51  
integer arithmetic 67  
interfaces 179  
    accessing variables 184  
    defining 179  
    extending 181  
    implementing 182  
interface statement 33, 179  
Internet 17  
Internet Explorer 21  
interpreter 13,26,41  
iteration statement 38

java 22  
Java character set 35  
java class library 304  
Java code 22  
java language packages 189  
java standard library 23, 304  
java statements 38  
Java tokens 33  
java.applet 189  
jawa.awt 189  
java.io 189  
java.lang 189  
java.net 189  
java.util 189  
javac 22  
javadoc 22  
javah 22  
javap 22  
jdb 22  
jump statement 38  
jumps in loops 122  
  
keywords 35,289  
  
labelled loops 123  
labeled statement 38  
length of arrays 158  
life-cycle of an applet 243  
life.cycle of a thread 208  
lightweight threads 202  
literals 37  
local variable 56  
  
machine code 42  
machine neutral 42  
mathematical functions 82  
member variables 130  
methods 2, 129, 130  
    abstract 149  
    finalizer 149  
    graphics 263  
    instance 139  
    nesting of 141  
    overloading of 138  
    overriding of 147  
    string 166  
    vector 169  
        final 148  
mixed-mode arithmetic 68  
modifiability 58  
modifiers 150  
modular programming 1  
multitask 201  
multithreaded 15  
multilevel inheritance 145  
multiple inheritance 179  
multithreaded 201  
  
NaN 53  
narrowing 59  
native methods 15  
nesting of blocks 56  
nesting of loops 121  
nesting of methods 141  
Netscape Navigator 21

**null** statement 120



Oak 12

object 2

object-oriented language 2,14

object-oriented paradigm 2

object-oriented programming 1,3

objects 129, 133

one-dimensional arrays 155

operators 37

    arithmetic 66

    assignment 72

    associatively of 81

    bitwise 74

    conditional 74,106

    decrement 73

    dot 75, 139

    increment 73

    logical 71

    relational 69

    shorthand 72

    ternary 74,106

    instanceof 75

    precedence of 76, 81

overloading of methods 138

overriding methods 147



package 14,32,188

    user-defined 193

    naming conventions 191

    statement 32

    accessing 193.

    creation 192

    systems 189

    using 193

    java.applet 189

    jawa.awt 189

java.io 189

java.lang 189

java.net 189

java.util 189

parameters 131

parameters to applets 277

pareht class -142

platform-neutral 13-14

polørpmssm 6,129,138

precedence operator 76,81

peparing for applets 239

private access 151

program blocks 56

protected access 151

public access 150



read statement 55

real arithmetic 67

real constants 48

relational operators 69

reusability 5

runnable interface 204,220

running a program 41



scope of variables 56

selection statement 38

separators 37

server 26

shorthand operators 72

source file 40

statements 38

    brake 123

    continue 123

    do 114

    for 116

    if 88, 90

    switch 102

while 113  
static members 139  
stopping a thread 208  
string array 166  
string class 165  
string constants 49  
string methods 166  
StringBuffer class 165,167  
strings 164  
structured programming 1  
subclass 6, 142  
subscript 155  
Sun Micro Systems 12  
super class 142  
switch statement 102  
symbolic constants 57  
synchronisation 218  
synchronisation statement 38  
system packages 189

ternary operators 74,106  
thread 201  
thread blocking 208  
thread class 204  
thread creation 203  
thread exceptions 214  
thread extension 204  
thread life.cycle 208  
thread methods 217  
thread priority 215  
thread stopping 208  
top.down programming 1  
two.dimentional arrays 161  
type conversion 59  
type conversion in expression 78

unicode 35  
uniform resource locator 237  
user defined package 193  
using packages 193

variables 50  
    class 56, 139  
    declaration of 53  
    giving values to 54  
    local 56  
    scope of 56  
    final 148  
    member 130  
    instance 56, 139

vectors 169  
vector methods 170  
virtual machine 42  
visibility control 150  
visibili~ modifiers 150

web applets 26  
web browser 13,19,21  
    " web page 246  
    web server 21  
    while statement 113  
widening 59  
World Wide Web 12,17  
wrapper classes 171